# CONTENTS

1 Introduction 3

2 Built-in Functions 5

3 Built-in Constants 23
  3.1 Constants added by the site module 23

4 Built-in Types 25
  4.1 Truth Value Testing 25
  4.2 Boolean Operations — and, or, not 25
  4.3 Comparisons 26
  4.4 Numeric Types — int, float, complex 26
  4.5 Iterator Types 32
  4.6 Sequence Types — list, tuple, range 32
  4.7 Text Sequence Type — str 37
  4.8 Binary Sequence Types — bytes, bytearray, memoryview 45
  4.9 Set Types — set, frozenset 54
  4.10 Mapping Types — dict 56
  4.11 Context Manager Types 60
  4.12 Other Built-in Types 60
  4.13 Special Attributes 63

5 Built-in Exceptions 65
  5.1 Base classes 65
  5.2 Concrete exceptions 66
  5.3 Warnings 70
  5.4 Exception hierarchy 71

6 Text Processing Services 73
  6.1 string — Common string operations 73
  6.2 re — Regular expression operations 82
  6.3 difflib — Helpers for computing deltas 98
  6.4 textwrap — Text wrapping and filling 107
  6.5 unicodedata — Unicode Database 110
  6.6 stringprep — Internet String Preparation 112
  6.7 readline — GNU readline interface 113
  6.8 rlcompleter — Completion function for GNU readline 116

7 Binary Data Services 117
  7.1 struct — Interpret bytes as packed binary data 117
  7.2 codecs — Codec registry and base classes 121

8 Data Types 137
  8.1 datetime — Basic date and time types 137
### 8.2 calendar
- General calendar-related functions

### 8.3 collections
- Container datatypes

### 8.4 collections.abc
- Abstract Base Classes for Containers

### 8.5 heapq
- Heap queue algorithm

### 8.6 bisect
- Array bisection algorithm

### 8.7 array
- Efficient arrays of numeric values

### 8.8 weakref
- Weak references

### 8.9 types
- Dynamic type creation and names for built-in types

### 8.10 copy
- Shallow and deep copy operations

### 8.11 pprint
- Data pretty printer

### 8.12 reprlib
- Alternate `repr()` implementation

### 9 Numeric and Mathematical Modules

#### 9.1 numbers
- Numeric abstract base classes

#### 9.2 math
- Mathematical functions

#### 9.3 cmath
- Mathematical functions for complex numbers

#### 9.4 decimal
- Decimal fixed point and floating point arithmetic

#### 9.5 fractions
- Rational numbers

### 10 Functional Programming Modules

#### 10.1 itertools
- Functions creating iterators for efficient looping

#### 10.2 functools
- Higher-order functions and operations on callable objects

### 11 File and Directory Access

#### 11.1 os.path
- Common pathname manipulations

#### 11.2 fileinput
- Iterate over lines from multiple input streams

#### 11.3 stat
- Interpreting `stat()` results

#### 11.4 filecmp
- File and Directory Comparisons

#### 11.5 tempfile
- Generate temporary files and directories

#### 11.6 glob
- Unix style pathname pattern expansion

#### 11.7 fnmatch
- Unix filename pattern matching

#### 11.8 linecache
- Random access to text lines

#### 11.9 shutil
- High-level file operations

#### 11.10 macpath
- Mac OS 9 path manipulation functions

### 12 Data Persistence

#### 12.1 pickle
- Python object serialization

#### 12.2 copyreg
- Register pickle support functions

#### 12.3 shelve
- Python object persistence

#### 12.4 marshal
- Internal Python object serialization

#### 12.5 dbm
- Interfaces to Unix “databases”

#### 12.6 sqlite3
- DB-API 2.0 interface for SQLite databases

### 13 Data Compression and Archiving

#### 13.1 zlib
- Compression compatible with `gzip`

#### 13.2 gzip
- Support for `gzip` files

#### 13.3 bz2
- Support for `bzip2` compression

#### 13.4 lzma
- Compression using the LZMA algorithm

#### 13.5 zipfile
- Work with ZIP archives

#### 13.6 tarfile
- Read and write tar archive files

### 14 File Formats

#### 14.1 csv
- CSV File Reading and Writing

#### 14.2 configparser
- Configuration file parser

#### 14.3 metrc
- Metrc file processing

#### 14.4 xdrlib
- Encode and decode XDR data

#### 14.5 plistlib
- Generate and parse Mac OS X .plist files
# Cryptographic Services

15.1 hashlib — Secure hashes and message digests ........................................... 383
15.2 hmac — Keyed-Hashing for Message Authentication ................................. 385

# Generic Operating System Services

16.1 os — Miscellaneous operating system interfaces ...................................... 387
16.2 io — Core tools for working with streams ............................................. 421
16.3 time — Time access and conversions ..................................................... 431
16.4 argparse — Parser for command-line options, arguments and sub-commands .... 438
16.5 optparse — Parser for command line options ......................................... 465
16.6 getopt — C-style parser for command line options .................................. 489
16.7 logging — Logging facility for Python .................................................... 491
16.8 logging.config — Logging configuration ................................................. 505
16.9 logging.handlers — Logging handlers .................................................... 514
16.10 getpass — Portable password input .................................................... 524
16.11 curses — Terminal handling for character-cell displays ............................. 525
16.12 curses.textpad — Text input widget for curses programs .......................... 540
16.13 curses.ascii — Utilities for ASCII characters ........................................ 541
16.14 curses.panel — A panel stack extension for curses .................................. 543
16.15 platform — Access to underlying platform’s identifying data ....................... 544
16.16 errno — Standard errno system symbols ............................................. 548
16.17 ctypes — A foreign function library for Python .................................... 553

# Concurrent Execution

17.1 threading — Thread-based parallelism .................................................. 583
17.2 multiprocessing — Process-based parallelism ....................................... 594
17.3 The concurrent package ............................................................................ 643
17.4 concurrent.futures — Launching parallel tasks ...................................... 643
17.5 subprocess — Subprocess management .................................................... 647
17.6 sched — Event scheduler ......................................................................... 660
17.7 queue — A synchronized queue class ..................................................... 661
17.8 select — Waiting for I/O completion ..................................................... 663
17.9 dummy_threading — Drop-in replacement for the threading module ............ 668
17.10 _thread — Low-level threading API .................................................... 669
17.11 _dummy_thread — Drop-in replacement for the _thread module ............... 670

# Interprocess Communication and Networking

18.1 socket — Low-level networking interface .............................................. 673
18.2 ssl — TLS/SSL wrapper for socket objects .............................................. 688
18.3 asyncore — Asynchronous socket handler ............................................. 704
18.4 asynchat — Asynchronous socket command/response handler .................... 708
18.5 signal — Set handlers for asynchronous events ...................................... 710
18.6 mmap — Memory-mapped file support ..................................................... 715

# Internet Data Handling

19.1 email — An email and MIME handling package ...................................... 719
19.2 json — JSON encoder and decoder ....................................................... 761
19.3 mailcap — Mailcap file handling ............................................................ 768
19.4 mailbox — Manipulate mailboxes in various formats .................................. 768
19.5 mimetypes — Map filenames to MIME types .......................................... 784
19.6 base64 — RFC 3548: Base16, Base32, Base64 Data Encodings .................... 787
19.7 binhex — Encode and decode binhex4 files ............................................ 789
19.8 binascii — Convert between binary and ASCII ........................................ 789
19.9 quopri — Encode and decode MIME quoted-printable data ....................... 791
19.10 uu — Encode and decode uuencode files ............................................. 792

# Structured Markup Processing Tools

20.1 html — HyperText Markup Language support ........................................ 793
20.2 html.parser — Simple HTML and XHTML parser .................................... 793
### 21 Internet Protocols and Support

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>webbrowser</td>
<td>Convenient Web-browser controller</td>
<td>845</td>
</tr>
<tr>
<td>cgi</td>
<td>Common Gateway Interface support</td>
<td>845</td>
</tr>
<tr>
<td>cgitb</td>
<td>Traceback manager for CGI scripts</td>
<td>847</td>
</tr>
<tr>
<td>wsgiref</td>
<td>WSGI Utilities and Reference Implementation</td>
<td>853</td>
</tr>
<tr>
<td>urllib</td>
<td>URL handling modules</td>
<td>854</td>
</tr>
<tr>
<td>urllib.request</td>
<td>Extensible library for opening URLs</td>
<td>862</td>
</tr>
<tr>
<td>urllib.response</td>
<td>Response classes used by urllib</td>
<td>878</td>
</tr>
<tr>
<td>urllib.parse</td>
<td>Parse URLs into components</td>
<td>878</td>
</tr>
<tr>
<td>urllib.error</td>
<td>Exception classes raised by urllib.request</td>
<td>884</td>
</tr>
<tr>
<td>urllib.robotparser</td>
<td>Parser for robots.txt</td>
<td>884</td>
</tr>
<tr>
<td>http</td>
<td>HTTP modules</td>
<td>885</td>
</tr>
<tr>
<td>http.client</td>
<td>HTTP protocol client</td>
<td>885</td>
</tr>
<tr>
<td>ftpclient</td>
<td>FTP protocol client</td>
<td>891</td>
</tr>
<tr>
<td>poplib</td>
<td>POP3 protocol client</td>
<td>895</td>
</tr>
<tr>
<td>imaplib</td>
<td>IMAP4 protocol client</td>
<td>897</td>
</tr>
<tr>
<td>nntplib</td>
<td>NNTP protocol client</td>
<td>903</td>
</tr>
<tr>
<td>smtplib</td>
<td>SMTP protocol client</td>
<td>909</td>
</tr>
<tr>
<td>smtplib</td>
<td>SMTP Server</td>
<td>914</td>
</tr>
<tr>
<td>telnetlib</td>
<td>Telnet client</td>
<td>916</td>
</tr>
<tr>
<td>uuid</td>
<td>UUID objects according to RFC 4122</td>
<td>918</td>
</tr>
<tr>
<td>socketserver</td>
<td>A framework for network servers</td>
<td>921</td>
</tr>
<tr>
<td>http.server</td>
<td>HTTP servers</td>
<td>928</td>
</tr>
<tr>
<td>http.cookies</td>
<td>HTTP state management</td>
<td>932</td>
</tr>
<tr>
<td>http.cookiejar</td>
<td>Cookie handling for HTTP clients</td>
<td>935</td>
</tr>
<tr>
<td>xmlrpc</td>
<td>XMLRPC server and client modules</td>
<td>943</td>
</tr>
<tr>
<td>xmlrpc.client</td>
<td>XML-RPC client access</td>
<td>943</td>
</tr>
<tr>
<td>xmlrpc.server</td>
<td>Basic XML-RPC servers</td>
<td>950</td>
</tr>
<tr>
<td>ipaddress</td>
<td>IPv4/IPv6 manipulation library</td>
<td>954</td>
</tr>
</tbody>
</table>

### 22 Multimedia Services

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>audioop</td>
<td>Manipulate raw audio data</td>
<td>965</td>
</tr>
<tr>
<td>aifc</td>
<td>Read and write AIFF and AIFC files</td>
<td>968</td>
</tr>
<tr>
<td>sunau</td>
<td>Read and write Sun AU files</td>
<td>970</td>
</tr>
<tr>
<td>wave</td>
<td>Read and write WAV files</td>
<td>972</td>
</tr>
<tr>
<td>chunk</td>
<td>Read IFF chunked data</td>
<td>974</td>
</tr>
<tr>
<td>colorsys</td>
<td>Conversions between color systems</td>
<td>975</td>
</tr>
<tr>
<td>imghdr</td>
<td>Determine the type of an image</td>
<td>976</td>
</tr>
<tr>
<td>sndhdr</td>
<td>Determine type of sound file</td>
<td>977</td>
</tr>
<tr>
<td>ossaudiodev</td>
<td>Access to OSS-compatible audio devices</td>
<td>977</td>
</tr>
</tbody>
</table>

### 23 Internationalization

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>gettext</td>
<td>Multilingual internationalization services</td>
<td>983</td>
</tr>
<tr>
<td>locale</td>
<td>Internationalization services</td>
<td>991</td>
</tr>
</tbody>
</table>

### 24 Program Frameworks

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>

iv
24.1 turtle — Turtle graphics .................................................... 999
24.2 cmd — Support for line-oriented command interpreters ............... 1031
24.3 shlex — Simple lexical analysis ........................................ 1036

25 Graphical User Interfaces with Tk ........................................ 1041
25.1 tkinter — Python interface to Tcl/Tk .................................. 1041
25.2 tkinter.ttk — Tk themed widgets ....................................... 1051
25.3 tkinter.tix — Extension widgets for Tk ............................... 1066
25.4 tkinter.scrolledtext — Scrolled Text Widget ......................... 1071
25.5 IDLE ................................................................. 1071
25.6 Other Graphical User Interface Packages .............................. 1074

26 Development Tools ....................................................... 1077
26.1 pydoc — Documentation generator and online help system ............ 1077
26.2 doctest — Test interactive Python examples .......................... 1078
26.3 unittest — Unit testing framework .................................... 1099
26.4 unittest.mock — mock object library ................................ 1121
26.5 unittest.mock — getting started ...................................... 1151
26.6 2to3 — Automated Python 2 to 3 code translation .................. 1169
26.7 test — Regression tests package for Python ........................... 1173
26.8 test.support — Utilities for the Python test suite .................... 1175
26.9 venv — Creation of virtual environments .............................. 1180

27 Debugging and Profiling .................................................. 1189
27.1 bdb — Debugger framework ............................................. 1189
27.2 faulthandler — Dump the Python traceback ........................... 1193
27.3 pdb — The Python Debugger ........................................... 1195
27.4 The Python Profilers .................................................... 1200
27.5 timeit — Measure execution time of small code snippets ............. 1207
27.6 trace — Trace or track Python statement execution ................. 1211

28 Python Runtime Services ................................................. 1215
28.1 sys — System-specific parameters and functions ....................... 1215
28.2 sysconfig — Provide access to Python’s configuration information 1227
28.3 builtins — Built-in objects ............................................ 1230
28.4 __main__ — Top-level script environment ............................. 1231
28.5 warnings — Warning control .......................................... 1231
28.6 contextlib — Utilities for with-statement contexts .................. 1236
28.7 abc — Abstract Base Classes .......................................... 1243
28.8 atexit — Exit handlers ................................................ 1247
28.9 traceback — Print or retrieve a stack traceback ....................... 1248
28.10 __future__ — Future statement definitions ........................... 1251
28.11 gc — Garbage Collector interface .................................... 1253
28.12 inspect — Inspect live objects ....................................... 1255
28.13 site — Site-specific configuration hook ............................. 1265
28.14 fpectl — Floating point exception control .......................... 1268
28.15 distutils — Building and installing Python modules ................. 1269

29 Custom Python Interpreters .............................................. 1271
29.1 code — Interpreter base classes ..................................... 1271
29.2 codeop — Compile Python code ....................................... 1273

30 Importing Modules ....................................................... 1275
30.1 imp — Access the import internals .................................... 1275
30.2 zipimport — Import modules from Zip archives ....................... 1279
30.3 pkgutil — Package extension utility .................................. 1280
30.4 modulefinder — Find modules used by a script ....................... 1283
30.5 runpy — Locating and executing Python modules ..................... 1284
30.6 importlib — An implementation of import ............................ 1286
While *reference-index* describes the exact syntax and semantics of the Python language, this library reference manual describes the standard library that is distributed with Python. It also describes some of the optional components that are commonly included in Python distributions.

Python’s standard library is very extensive, offering a wide range of facilities as indicated by the long table of contents listed below. The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs.

The Python installers for the Windows platform usually include the entire standard library and often also include many additional components. For Unix-like operating systems Python is normally provided as a collection of packages, so it may be necessary to use the packaging tools provided with the operating system to obtain some or all of the optional components.

In addition to the standard library, there is a growing collection of several thousand components (from individual programs and modules to packages and entire application development frameworks), available from the Python Package Index.
The “Python library” contains several different kinds of components.

It contains data types that would normally be considered part of the “core” of a language, such as numbers and lists. For these types, the Python language core defines the form of literals and places some constraints on their semantics, but does not fully define the semantics. (On the other hand, the language core does define syntactic properties like the spelling and priorities of operators.)

The library also contains built-in functions and exceptions — objects that can be used by all Python code without the need of an import statement. Some of these are defined by the core language, but many are not essential for the core semantics and are only described here.

The bulk of the library, however, consists of a collection of modules. There are many ways to dissect this collection. Some modules are written in C and built into the Python interpreter; others are written in Python and imported in source form. Some modules provide interfaces that are highly specific to Python, like printing a stack trace; some provide interfaces that are specific to particular operating systems, such as access to specific hardware; others provide interfaces that are specific to a particular application domain, like the World Wide Web. Some modules are available in all versions and ports of Python; others are only available when the underlying system supports or requires them; yet others are available only when a particular configuration option was chosen at the time when Python was compiled and installed.

This manual is organized “from the inside out”: it first describes the built-in data types, then the built-in functions and exceptions, and finally the modules, grouped in chapters of related modules. The ordering of the chapters as well as the ordering of the modules within each chapter is roughly from most relevant to least important.

This means that if you start reading this manual from the start, and skip to the next chapter when you get bored, you will get a reasonable overview of the available modules and application areas that are supported by the Python library. Of course, you don’t have to read it like a novel — you can also browse the table of contents (in front of the manual), or look for a specific function, module or term in the index (in the back). And finally, if you enjoy learning about random subjects, you choose a random page number (see module random) and read a section or two. Regardless of the order in which you read the sections of this manual, it helps to start with chapter Built-in Functions, as the remainder of the manual assumes familiarity with this material.

Let the show begin!
The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

| abs() | dict() | help() | min() | setattr() |
| all() | dir()  | hex()  | next() | slice()  |
| any() | divmod() | id() | object() | sorted() |
| ascii() | enumerate() | input() | oct() | staticmethod() |
| bin() | eval() | int() | open() | str() |
| bool() | exec() | isinstance() | ord() | sum() |
| bytearray() | filter() | issubclass() | pow() | super() |
| bytes() | float() | iter() | print() | tuple() |
| callable() | format() | len() | property() | type() |
| chr() | frozenset() | list() | range() | vars() |
| classmethod() | getattr() | locals() | repr() | zip() |
| compile() | globals() | map() | reversed() | __import__() |
| complex() | hasattr() | max() | round() |
| delattr() | hash() | memoryview() | set() |

**abs(x)**
Return the absolute value of a number. The argument may be an integer or a floating point number. If the argument is a complex number, its magnitude is returned.

**all(iterable)**
Return True if all elements of the *iterable* are true (or if the iterable is empty). Equivalent to:

```python
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

**any(iterable)**
Return True if any element of the *iterable* is true. If the iterable is empty, return False. Equivalent to:

```python
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

**ascii(object)**
As `repr()`, return a string containing a printable representation of an object, but escape the non-ASCII characters in the string returned by `repr()` using \x, \u or \U escapes. This generates a string similar to that returned by `repr()` in Python 2.
bin \((x)\)
Convert an integer number to a binary string. The result is a valid Python expression. If \(x\) is not a Python `int` object, it has to define an `__index__()` method that returns an integer.

bool \([\{x\}]\)
Convert a value to a Boolean, using the standard truth testing procedure. If \(x\) is false or omitted, this returns `False`; otherwise it returns `True`. Class `bool` is also a class, which is a subclass of `int` (see Numeric Types — `int`, `float`, `complex`). Class `bool` cannot be subclassed further. Its only instances are `False` and `True` (see Boolean Values).

bytearray \([\{source[, encoding[, errors ]]\}]\)
Return a new array of bytes. The `bytearray` type is a mutable sequence of integers in the range 0 <= \(x\) < 256. It has most of the usual methods of mutable sequences, described in Mutable Sequence Types, as well as most methods that the `bytes` type has, see Bytes and Bytearray Operations.

The optional `source` parameter can be used to initialize the array in a few different ways:

- •If it is a `string`, you must also give the `encoding` (and optionally, `errors`) parameters; `bytearray()` then converts the string to bytes using `str.encode()`.
- •If it is an `integer`, the array will have that size and will be initialized with null bytes.
- •If it is an object conforming to the `buffer` interface, a read-only buffer of the object will be used to initialize the bytes array.
- •If it is an `iterable`, it must be an iterable of integers in the range 0 <= \(x\) < 256, which are used as the initial contents of the array.

Without an argument, an array of size 0 is created.

See also Binary Sequence Types — `bytes`, `bytearray`, `memoryview` and Bytearray Objects.

bytes \([\{source[, encoding[, errors ]]\}]\)
Return a new “bytes” object, which is an immutable sequence of integers in the range 0 <= \(x\) < 256. `bytes` is an immutable version of `bytearray` – it has the same non-mutating methods and the same indexing and slicing behavior.

Accordingly, constructor arguments are interpreted as for `bytearray()`.

Bytes objects can also be created with literals, see `strings`.

See also Binary Sequence Types — `bytes`, `bytearray`, `memoryview`, `Bytes`, and Bytes and Bytearray Operations.

callable \((object)\)
Return `True` if the `object` argument appears callable, `False` if not. If this returns true, it is still possible that a call fails, but if it is false, calling `object` will never succeed. Note that classes are callable (calling a class returns a new instance); instances are callable if their class has a `__call__()` method. New in version 3.2: This function was first removed in Python 3.0 and then brought back in Python 3.2.

chr \((i)\)
Return the string representing a character whose Unicode codepoint is the integer \(i\). For example, `chr(97)` returns the string ‘a’. This is the inverse of `ord()`. The valid range for the argument is from 0 through 1,114,111 (0x10FFFF in base 16). `ValueError` will be raised if \(i\) is outside that range.

classmethod \((function)\)
Return a class method for `function`.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```python
class C:
    @classmethod
    def f(cls, arg1, arg2, ...): ...
```

The `@classmethod` form is a function decorator – see the description of function definitions in `function` for details.

6 Chapter 2. Built-in Functions
It can be called either on the class (such as \texttt{C.f()}) or on an instance (such as \texttt{C().f()}). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see \texttt{staticmethod()} in this section.

For more information on class methods, consult the documentation on the standard type hierarchy in \texttt{types}.

\begin{tcl}
\texttt{compile (source, filename, mode, flags=0, dont_inherit=False, optimize=-1)}
\end{tcl}

Compile the \texttt{source} into a code or AST object. Code objects can be executed by \texttt{exec()} or \texttt{eval()}. \texttt{source} can either be a string or an AST object. Refer to the \texttt{ast} module documentation for information on how to work with AST objects.

The \texttt{filename} argument should give the file from which the code was read; pass some recognizable value if it wasn’t read from a file (‘\texttt{<string>’} is commonly used).

The \texttt{mode} argument specifies what kind of code must be compiled; it can be ‘\texttt{exec}’ if \texttt{source} consists of a sequence of statements, ‘\texttt{eval}’ if it consists of a single expression, or ‘\texttt{single}’ if it consists of a single interactive statement (in the latter case, expression statements that evaluate to something other than \texttt{None} will be printed).

The optional arguments \texttt{flags} and \texttt{dont_inherit} control which future statements (see PEP 236) affect the compilation of \texttt{source}. If neither is present (or both are zero) the code is compiled with those future statements that are in effect in the code that is calling compile. If the \texttt{flags} argument is given and \texttt{dont_inherit} is not (or is zero) then the future statements specified by the \texttt{flags} argument are used in addition to those that would be used anyway. If \texttt{dont_inherit} is a non-zero integer then the \texttt{flags} argument is it – the future statements in effect around the call to compile are ignored.

Future statements are specified by bits which can be bitwise ORed together to specify multiple statements. The bitfield required to specify a given feature can be found as the \texttt{compiler_flag} attribute on the \_Feature instance in the \texttt{__future__} module.

The argument \texttt{optimize} specifies the optimization level of the compiler; the default value of \texttt{-1} selects the optimization level of the interpreter as given by \texttt{-O} options. Explicit levels are 0 (no optimization; \texttt{__debug__} is true), 1 (asserts are removed, \texttt{__debug__} is false) or 2 (docstrings are removed too).

This function raises \texttt{SyntaxError} if the compiled source is invalid, and \texttt{TypeError} if the source contains null bytes.

\begin{note}
When compiling a string with multi-line code in ‘\texttt{single}’ or ‘\texttt{eval}’ mode, input must be terminated by at least one newline character. This is to facilitate detection of incomplete and complete statements in the \texttt{code} module.
\end{note}

\begin{note}
Changed in version 3.2: Allowed use of Windows and Mac newlines. Also input in ‘\texttt{exec}’ mode does not have to end in a newline anymore. Added the \texttt{optimize} parameter.
\end{note}

\begin{tcl}
\texttt{complex ([real[, imag]])}
\end{tcl}

Create a complex number with the value \texttt{real + imag*j} or convert a string or number to a complex number. If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If \texttt{imag} is omitted, it defaults to zero and the function serves as a numeric conversion function like \texttt{int()} and \texttt{float()}. If both arguments are omitted, returns \texttt{0j}.

\begin{note}
When converting from a string, the string must not contain whitespace around the central + or - operator. For example, \texttt{complex(’1+2j’)} is fine, but complex(’1 + 2j’) raises \texttt{ValueError}.
\end{note}

The complex type is described in \texttt{Numeric Types — int, float, complex}.

\begin{tcl}
\texttt{delattr (object, name)}
\end{tcl}

This is a relative of \texttt{setattr()}. The arguments are an object and a string. The string must be the name
of one of the object’s attributes. The function deletes the named attribute, provided the object allows it. For example, `delattr(x, ‘foobar’)` is equivalent to `del x.foobar`.

```python
dict(**kwarg)
dict(mapping, **kwarg)
dict(iterable, **kwarg)
```

Create a new dictionary. The `dict` object is the dictionary class. See `dict` and `Mapping Types — dict` for documentation about this class.

For other containers see the built-in `list`, `set`, and `tuple` classes, as well as the `collections` module.

```python
dir([object])
```

Without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object.

If the object has a method named `__dir__()`, this method will be called and must return the list of attributes. This allows objects that implement a custom `__getattr__()` or `__getattribute__()` function to customize the way `dir()` reports their attributes.

If the object does not provide `__dir__()`, the function tries its best to gather information from the object’s `__dict__` attribute, if defined, and from its type object. The resulting list is not necessarily complete, and may be inaccurate when the object has a custom `__getattr__()`.

The default `dir()` mechanism behaves differently with different types of objects, as it attempts to produce the most relevant, rather than complete, information:

- If the object is a module object, the list contains the names of the module’s attributes.
- If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases.
- Otherwise, the list contains the object’s attributes’ names, the names of its class’s attributes, and recursively of the attributes of its class’s base classes.

The resulting list is sorted alphabetically. For example:

```python
>>> import struct
>>> dir()  # show the names in the module namespace
[ '__builtins__', '__name__', 'struct']
>>> dir(struct)  # show the names in the struct module
['Struct', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__initializing__', '__loader__', '__name__', '__package__',
 'clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape:
...     def __dir__(self):
...         return ['area', 'perimeter', 'location']

s = Shape()
>>> dir(s)
[ 'area', 'location', 'perimeter']
```

**Note:** Because `dir()` is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.

```python
divmod(a, b)
```

Take two (non complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using integer division. With mixed operand types, the rules for binary arithmetic operators apply. For integers, the result is the same as `(a // b, a % b)`. For floating point numbers the result is `(q, a % b)`, where `q` is usually `math.floor(a / b)` but may be 1 less than that. In any case `q +`
b + a % b is very close to a, if a % b is non-zero it has the same sign as b, and 0 <= abs(a % b) < abs(b).

**enumerate**( iterable, start=0)

Return an enumerate object. iterable must be a sequence, an iterator, or some other object which supports iteration. The **__next__()** method of the iterator returned by **enumerate()** returns a tuple containing a count (from start which defaults to 0) and the values obtained from iterating over iterable.

```python
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']

>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]

>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

Equivalent to:

```python
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
    n += 1
```

**eval**( expression, globals=None, locals=None)

The arguments are a string and optional globals and locals. If provided, **globals** must be a dictionary. If provided, **locals** can be any mapping object.

The **expression** argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the **globals** and **locals** dictionaries as global and local namespace. If the **globals** dictionary is present and lacks '__builtins__', the current **globals** are copied into **globals** before **expression** is parsed. This means that **expression** normally has full access to the standard **builtins** module and restricted environments are propagated. If the **locals** dictionary is omitted it defaults to the **globals** dictionary. If both dictionaries are omitted, the expression is executed in the environment where **eval()** is called. The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:

```python
>>> x = 1
>>> eval('x+1')
2
```

This function can also be used to execute arbitrary code objects (such as those created by **compile()**). In this case pass a code object instead of a string. If the code object has been compiled with 'exec' as the **mode** argument, **eval()**'s return value will be None.

Hints: dynamic execution of statements is supported by the **exec()** function. The **globals()** and **locals()** functions returns the current global and local dictionary, respectively, which may be useful to pass around for use by **eval()** or **exec()**.

See **ast.literal_eval()** for a function that can safely evaluate strings with expressions containing only literals.

**exec**( object[, globals[, locals ]])

This function supports dynamic execution of Python code. **object** must be either a string or a code object. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs). ¹ If it is a code object, it is simply executed. In all cases, the code that’s executed is expected to be valid as file input (see the section “File input” in the Reference Manual). Be aware that the **return** and **yield** statements may not be used outside of function definitions even within the context of code passed to the **exec()** function. The return value is None.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only **globals** is provided, it must be a dictionary, which will be used for both the global and the local variables. If **globals**

---

¹ Note that the parser only accepts the Unix-style end of line convention. If you are reading the code from a file, make sure to use newline conversion mode to convert Windows or Mac-style newlines.
and `locals` are given, they are used for the global and local variables, respectively. If provided, `locals` can be any mapping object. Remember that at module level, globals and locals are the same dictionary. If `exec` gets two separate objects as `globals` and `locals`, the code will be executed as if it were embedded in a class definition.

If the `globals` dictionary does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module `builtins` is inserted under that key. That way you can control what builtins are available to the executed code by inserting your own `__builtins__` dictionary into `globals` before passing it to `exec()`.

**Note:** The built-in functions `globals()` and `locals()` return the current global and local dictionary, respectively, which may be useful to pass around for use as the second and third argument to `exec()`.

**Note:** The default `locals` act as described for function `locals()` below: modifications to the default `locals` dictionary should not be attempted. Pass an explicit `locals` dictionary if you need to see effects of the code on `locals` after function `exec()` returns.

### filter (function, iterable)

Construct an iterator from those elements of `iterable` for which `function` returns true. `iterable` may be either a sequence, a container which supports iteration, or an iterator. If `function` is `None`, the identity function is assumed, that is, all elements of `iterable` that are false are removed.

Note that `filter(function, iterable)` is equivalent to the generator expression `{item for item in iterable if function(item)} if function is not None and {item for item in iterable if item} if function is None.

See `itertools.filterfalse()` for the complementary function that returns elements of `iterable` for which `function` returns false.

### float ([x])

Convert a string or a number to floating point.

If the argument is a string, it should contain a decimal number, optionally preceded by a sign, and optionally embedded in whitespace. The optional sign may be `+` or `-`; a `+` sign has no effect on the value produced. The argument may also be a string representing a NaN (not-a-number), or a positive or negative infinity. More precisely, the input must conform to the following grammar after leading and trailing whitespace characters are removed:

\[
\begin{align*}
\text{sign} & ::= \ "\+" \mid \ "\-" \\
\text{infinity} & ::= \ "\text{Infinity}" \mid \ "\text{inf}\" \\
\text{nan} & ::= \ "\text{nan}\" \\
\text{numeric\_value} & ::= \ \text{floatnumber} \mid \ \text{infinity} \mid \ \text{nan} \\
\text{numeric\_string} & ::= \ [\text{sign}] \ \text{numeric\_value}
\end{align*}
\]

Here `floatnumber` is the form of a Python floating-point literal, described in `floating`. Case is not significant, so, for example, “inf”, “Inf”, “INFINITY” and “iNfINity” are all acceptable spellings for positive infinity.

Otherwise, if the argument is an integer or a floating point number, a floating point number with the same value (within Python’s floating point precision) is returned. If the argument is outside the range of a Python float, an `OverflowError` will be raised.

For a general Python object `x`, `float(x)` delegates to `x.__float__()`.

If no argument is given, 0.0 is returned.

Examples:

```python
>>> float(‘+1.23’)
1.23
>>> float(‘-12345\n’)
```

---

Chapter 2. Built-in Functions
-12345.0
>>> float('1e-003')
0.001
>>> float('+1E6')
1000000.0
>>> float('-Infinity')
-inf

The float type is described in *Numeric Types — int, float, complex*.

**format** *(value[, format_spec]*)

Convert a *value* to a “formatted” representation, as controlled by *format_spec*. The interpretation of *format_spec* will depend on the type of the *value* argument, however there is a standard formatting syntax that is used by most built-in types: *Format Specification Mini-Language*.

The default *format_spec* is an empty string which usually gives the same effect as calling *str(value)*.

A call to *format(value, format_spec)* is translated to *type(value).*format.__format__(format_spec)* which bypasses the instance dictionary when searching for the value’s __format__() method. A *TypeError* exception is raised if the method is not found or if either the *format_spec* or the return value are not strings.

**frozenset** *(iterable]*)

Return a new *frozenset* object, optionally with elements taken from *iterable*. *frozenset* is a built-in class. See *frozenset* and *Set Types — set, frozenset* for documentation about this class.

For other containers see the built-in *set, list, tuple*, and *dict* classes, as well as the *collections* module.

**getattr** *(object, name[, default]*)

Return the value of the named attribute of *object*. *name* must be a string. If the string is the name of one of the object’s attributes, the result is the value of that attribute. For example, *getattr(x, ’foobar’)* is equivalent to *x.foobar*. If the named attribute does not exist, *default* is returned if provided, otherwise *AttributeError* is raised.

**globals** *(*)

Return a dictionary representing the current global symbol table. This is always the dictionary of the current module (inside a function or method, this is the module where it is defined, not the module from which it is called).

**hasattr** *(object, name)*

The arguments are an object and a string. The result is True if the string is the name of one of the object’s attributes, False if not. (This is implemented by calling *getattr(object, name)* and seeing whether it raises an *AttributeError* or not.)

**hash** *(object)*

Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).

**help** *(object]*)

Invoke the built-in help system. (This function is intended for interactive use.) If no argument is given, the interactive help system starts on the interpreter console. If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console. If the argument is any other kind of object, a help page on the object is generated.

This function is added to the built-in namespace by the *site* module.
hex(x)
Convert an integer number to a hexadecimal string. The result is a valid Python expression. If x is not a
Python int object, it has to define an __index__() method that returns an integer.

Note: To obtain a hexadecimal string representation for a float, use the float.hex() method.

id(object)
Return the “identity” of an object. This is an integer which is guaranteed to be unique and constant for this
object during its lifetime. Two objects with non-overlapping lifetimes may have the same id() value.

CPython implementation detail: This is the address of the object in memory.

input([prompt])
If the prompt argument is present, it is written to standard output without a trailing newline. The function
then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When
EOF is read, EOFError is raised. Example:

>>> s = input('--> ')
--> Monty Python’s Flying Circus
>>> s
"Monty Python’s Flying Circus"

If the readline module was loaded, then input() will use it to provide elaborate line editing and
history features.

int(x=0)
int(x, base=10)
Convert a number or string x to an integer, or return 0 if no arguments are given. If x is a number, return
x. __int__() . For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing
an integer literal in radix base. Optionally, the literal can be preceded by + or - (with no space in between)
and surrounded by whitespace. A base-n literal consists of the digits 0 to n-1, with a to z (or A to Z) having
values 10 to 35. The default base is 10. The allowed values are 0 and 2-36. Base-2, -8, and -16 literals
can be optionally prefixed with 0b/0B, 0o/0O, or 0x/0X, as with integer literals in code. Base 0 means to
interpret exactly as a code literal, so that the actual base is 2, 8, 10, or 16, and so that int(’010’, 0)
is not legal, while int(’010’, 8) is, as well as int(’010’, 2).

The integer type is described in Numeric Types — int, float, complex.

instance(object, classinfo)
Return true if the object argument is an instance of the classinfo argument, or of a (direct, indirect or virtual)
subclass thereof. If object is not an object of the given type, the function always returns false. If classinfo
is not a class (type object), it may be a tuple of type objects, or may recursively contain other such tuples (other
sequence types are not accepted). If classinfo is not a type or tuple of types and such tuples, a TypeError
exception is raised.

issubclass(class, classinfo)
Return true if class is a subclass (direct, indirect or virtual) of classinfo. A class is considered a subclass of
itself. classinfo may be a tuple of class objects, in which case every entry in classinfo will be checked. In
any other case, a TypeError exception is raised.

iter(object[, sentinel])
Return an iterator object. The first argument is interpreted very differently depending on the presence of the
second argument. Without a second argument, object must be a collection object which supports the iteration
protocol (the __iter__() method), or it must support the sequence protocol (the __getitem__() method with integer arguments starting at 0). If it does not support either of those protocols, TypeError
is raised. If the second argument, sentinel, is given, then object must be a callable object. The iterator
created in this case will call object with no arguments for each call to its __next__() method; if the
value returned is equal to sentinel, StopIteration will be raised, otherwise the value will be returned.

See also Iterator Types.
One useful application of the second form of `iter()` is to read lines of a file until a certain line is reached. The following example reads a file until the `readline()` method returns an empty string:

```python
with open('mydata.txt') as fp:
    for line in iter(fp.readline, ''):
        process_line(line)
```

`len(s)`
Return the length (the number of items) of an object. The argument may be a sequence (string, tuple or list) or a mapping (dictionary).

`list([iterable])`
Rather than being a function, `list` is actually a mutable sequence type, as documented in *Lists* and *Sequence Types — list, tuple, range*.

`locals()`
Update and return a dictionary representing the current local symbol table. Free variables are returned by `locals()` when it is called in function blocks, but not in class blocks.

**Note:** The contents of this dictionary should not be modified; changes may not affect the values of local and free variables used by the interpreter.

`map(function, iterable, ...)`
Return an iterator that applies `function` to every item of `iterable`, yielding the results. If additional `iterable` arguments are passed, `function` must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. For cases where the function inputs are already arranged into argument tuples, see `itertools.starmap()`.

`max(iterable, *[key])`

`max(arg1, arg2, *args, *[key])`
Return the largest item in an iterable or the largest of two or more arguments.

If one positional argument is provided, `iterable` must be a non-empty iterable (such as a non-empty string, tuple or list). The largest item in the iterable is returned. If two or more positional arguments are provided, the largest of the positional arguments is returned.

The optional keyword-only `key` argument specifies a one-argument ordering function like that used for `list.sort()`.

If multiple items are maximal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as `sorted(iterable, key=keyfunc, reverse=True)[0]` and `heapq.nlargest(1, iterable, key=keyfunc)`.

`memoryview(obj)`
Return a “memory view” object created from the given argument. See *Memory Views* for more information.

`min(iterable, *[key])`

`min(arg1, arg2, *args, *[key])`
Return the smallest item in an iterable or the smallest of two or more arguments.

If one positional argument is provided, `iterable` must be a non-empty iterable (such as a non-empty string, tuple or list). The smallest item in the iterable is returned. If two or more positional arguments are provided, the smallest of the positional arguments is returned.

The optional keyword-only `key` argument specifies a one-argument ordering function like that used for `list.sort()`.

If multiple items are minimal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as `sorted(iterable, key=keyfunc)[0]` and `heapq.nsmallest(1, iterable, key=keyfunc)`.
next (iterator[, default])

Retrieve the next item from the iterator by calling its __next__() method. If default is given, it is returned if the iterator is exhausted, otherwise StopIteration is raised.

object()

Return a new featureless object. object is a base for all classes. It has the methods that are common to all instances of Python classes. This function does not accept any arguments.

Note: object does not have a __dict__, so you can’t assign arbitrary attributes to an instance of the object class.

oct (x)

Convert an integer number to an octal string. The result is a valid Python expression. If x is not a Python int object, it has to define an __index__() method that returns an integer.

open (file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)

Open file and return a corresponding file object. If the file cannot be opened, an OSError is raised.

file is either a string or bytes object giving the pathname (absolute or relative to the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed, unless closefd is set to False.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to ‘r’ which means open for reading in text mode. Other common values are ‘w’ for writing (truncating the file if it already exists), ‘x’ for exclusive creation and ‘a’ for appending (which on some Unix systems, means that all writes append to the end of the file regardless of the current seek position). In text mode, if encoding is not specified the encoding used is platform dependent: locale.getpreferredencoding(False) is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave encoding unspecified.) The available modes are:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘r’</td>
<td>open for reading (default)</td>
</tr>
<tr>
<td>‘w’</td>
<td>open for writing, truncating the file first</td>
</tr>
<tr>
<td>‘x’</td>
<td>open for exclusive creation, failing if the file already exists</td>
</tr>
<tr>
<td>‘a’</td>
<td>open for writing, appending to the end of the file if it exists</td>
</tr>
<tr>
<td>‘b’</td>
<td>binary mode</td>
</tr>
<tr>
<td>‘t’</td>
<td>text mode (default)</td>
</tr>
<tr>
<td>‘+’</td>
<td>open a disk file for updating (reading and writing)</td>
</tr>
<tr>
<td>‘U’</td>
<td>universal newlines mode (for backwards compatibility; should not be used in new code)</td>
</tr>
</tbody>
</table>

The default mode is ‘r’ (open for reading text, synonym of ‘rt’). For binary read-write access, the mode ‘w+b’ opens and truncates the file to 0 bytes. ‘r+b’ opens the file without truncation.

As mentioned in the Overview, Python distinguishes between binary and text I/O. Files opened in binary mode (including ‘b’ in the mode argument) return contents as bytes objects without any decoding. In text mode (the default, or when ‘t’ is included in the mode argument), the contents of the file are returned as str, the bytes having been first decoded using a platform-dependent encoding or using the specified encoding if given.

Note: Python doesn’t depend on the underlying operating system’s notion of text files; all the processing is done by Python itself, and is therefore platform-independent.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size in bytes of a fixed-size chunk buffer. When no buffering argument is given, the default buffering policy works as follows:

• Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device’s “block size” and falling back on io.DEFAULT_BUFFER_SIZE. On many systems, the buffer will typically be 4096 or 8192 bytes long.
• "Interactive" text files (files for which `isatty()` returns True) use line buffering. Other text files use the policy described above for binary files.

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent (whatever `locale.getpreferredencoding()` returns), but any encoding supported by Python can be used. See the `codecs` module for the list of supported encodings.

decoding is an optional string that specifies how encoding and decoding errors are to be handled–this cannot be used in binary mode. A variety of standard error handlers are available, though any error handling name that has been registered with `codecs.register_error()` is also valid. The standard names are:

- `'strict'` to raise a `ValueError` exception if there is an encoding error. The default value of `None` has the same effect.
- `'ignore'` ignores errors. Note that ignoring encoding errors can lead to data loss.
- `'replace'` causes a replacement marker (such as `?'`) to be inserted where there is malformed data.
- `'surrogateescape'` will represent any incorrect bytes as code points in the Unicode Private Use Area ranging from U+DC80 to U+DCFF. These private code points will then be turned back into the same bytes when the `surrogateescape` error handler is used when writing data. This is useful for processing files in an unknown encoding.
- `'xmlcharrefreplace'` is only supported when writing to a file. Characters not supported by the encoding are replaced with the appropriate XML character reference `&#nnn;`.
- `'backslashreplace'` (also only supported when writing) replaces unsupported characters with Python’s backslashed escape sequences.

newline controls how universal newlines mode works (it only applies to text mode). It can be `None`, `''`, `’\n’`, `’\r’`, and `’\r\n’`. It works as follows:

- When reading input from the stream, if `newline` is `None`, universal newlines mode is enabled. Lines in the input can end in `’\n’`, `’\r’`, or `’\r\n’`, and these are translated into `’\n’` before being returned to the caller. If it is `''`, universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.

- When writing output to the stream, if `newline` is `None`, any `’\n’` characters written are translated to the system default line separator, `os.linesep`. If `newline` is `''` or `’\n’`, no translation takes place. If `newline` is any of the other legal values, any `’\n’` characters written are translated to the given string.

If `closefd` is `False` and a file descriptor rather than a filename was given, the underlying file descriptor will be kept open when the file is closed. If a filename is given `closefd` has no effect and must be `True` (the default).

A custom opener can be used by passing a callable as `opener`. The underlying file descriptor for the file object is then obtained by calling `opener` with `(file, flags)`. `opener` must return an open file descriptor (passing `os.open` as `opener` results in functionality similar to passing `None`).

The following example uses the `dir_fd` parameter of the `os.open()` function to open a file relative to a given directory:

```python
>>> import os
>>> dir_fd = os.open('somedir', os.O_RDONLY)
>>> def opener(path, flags):
...     return os.open(path, flags, dir_fd=dir_fd)
...
>>> with open('spamspam.txt', 'w', opener=opener) as f:
...     print('This will be written to somedir/spamspam.txt', file=f)
...
>>> os.close(dir_fd)  # don't leak a file descriptor
```
The Python Library Reference, Release 3.3.3

Changed in version 3.3: The opener parameter was added. The ‘x’ mode was added. The type of file object returned by the open() function depends on the mode. When open() is used to open a file in a text mode (‘w’, ‘r’, ‘wt’, ‘rt’, etc.), it returns a subclass of io.TextIOBase (specifically io.TextIOWrapper). When used to open a file in a binary mode with buffering, the returned class is a subclass of io.BufferedIOBase. The exact class varies: in read binary mode, it returns a io.BufferedReader; in write binary and append binary modes, it returns a io.BufferedWriter, and in read/write mode, it returns a io.BufferedRandom. When buffering is disabled, the raw stream, a subclass of io.RawIOBase, io.FileIO, is returned.

See also the file handling modules, such as fileinput, io (where open() is declared), os, os.path, tempfile, and shutil. Changed in version 3.3: IOError used to be raised, it is now an alias of OSError. FileExistsError is now raised if the file opened in exclusive creation mode (‘x’) already exists.

ord(c)
Given a string representing one Unicode character, return an integer representing the Unicode code point of that character. For example, ord(‘a’) returns the integer 97 and ord(‘†’) returns 8224. This is the inverse of chr().

pow(x, y, z)
Return x to the power y; if z is present, return x to the power y, modulo z (computed more efficiently than pow(x, y) % z). The two-argument form pow(x, y) is equivalent to using the power operator: x**y.

The arguments must have numeric types. With mixed operand types, the coercion rules for binary arithmetic operators apply. For int operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, 10**2 returns 100, but 10**-2 returns 0.01. If the second argument is negative, the third argument must be omitted. If z is present, x and y must be of integer types, and y must be non-negative.

print(*objects, sep='\ ', end='\ ', file=sys.stdout, flush=False)
Print objects to the stream file, separated by sep and followed by end. sep, end and file, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like str() does and written to the stream, separated by sep and followed by end. Both sep and end must be strings; they can also be None, which means to use the default values. If no objects are given, print() will just write end.

The file argument must be an object with a write(string) method; if it is not present or None, sys.stdout will be used. Whether output is buffered is usually determined by file, but if the flush keyword argument is true, the stream is forcibly flushed. Changed in version 3.3: Added the flush keyword argument.

property (fget=None, fset=None, fdel=None, doc=None)
Return a property attribute.

fget is a function for getting an attribute value, likewise fset is a function for setting, and fdel a function for del’ing, an attribute. Typical use is to define a managed attribute x:

class C:
    def __init__(self):
        self._x = None
    
def getx(self):
        return self._x
    def setx(self, value):
        self._x = value
    def delx(self):
        del self._x
    x = property(getx, setx, delx, "I’m the ’x’ property.")

If then c is an instance of C, c.x will invoke the getter, c.x = value will invoke the setter and del c.x the deleter.
If given, doc will be the docstring of the property attribute. Otherwise, the property will copy fget’s docstring (if it exists). This makes it possible to create read-only properties easily using property() as a decorator:

```python
class Parrot:
    def __init__(self):
        self._voltage = 100000

@property
def voltage(self):
    """Get the current voltage.""
    return self._voltage
```

turns the voltage() method into a “getter” for a read-only attribute with the same name.

A property object has getter, setter, and deleter methods usable as decorators that create a copy of the property with the corresponding accessor function set to the decorated function. This is best explained with an example:

```python
class C:
    def __init__(self):
        self._x = None

@property
def x(self):
    """I’m the ‘x’ property.""
    return self._x

@x.setter
def x(self, value):
    self._x = value

@x.deleter
def x(self):
    del self._x
```

This code is exactly equivalent to the first example. Be sure to give the additional functions the same name as the original property (x in this case.)

The returned property also has the attributes fget, fset, and fdel corresponding to the constructor arguments.

```python
range(stop)
range(start, stop[, step])
```

Rather than being a function, range is actually an immutable sequence type, as documented in Ranges and Sequence Types — list, tuple, range.

```python
repr(object)
```

Return a string containing a printable representation of an object. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to `eval()`, otherwise the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object. A class can control what this function returns for its instances by defining a `__repr__()` method.

```python
reversed(seq)
```

Return a reverse iterator. seq must be an object which has a `__reversed__()` method or supports the sequence protocol (the `__len__() method and the `__getitem__() method with integer arguments starting at 0).

```python
round(number[, ndigits])
```

Return the floating point value number rounded to ndigits digits after the decimal point. If ndigits is omitted, it defaults to zero. Delegates to number.__round__(ndigits).
For the built-in types supporting `round()`, values are rounded to the closest multiple of 10 to the power minus `ndigits`; if two multiples are equally close, rounding is done toward the even choice (so, for example, both `round(0.5)` and `round(-0.5)` are 0, and `round(1.5)` is 2). The return value is an integer if called with one argument, otherwise of the same type as `number`.

**Note:** The behavior of `round()` for floats can be surprising: for example, `round(2.675, 2)` gives 2.67 instead of the expected 2.68. This is not a bug: it’s a result of the fact that most decimal fractions can’t be represented exactly as a float. See [tut-fp-issues](https://docs.python.org/) for more information.

```python
set([iterable])
```

Return a new `set` object, optionally with elements taken from `iterable`. `set` is a built-in class. See `set` and `Set Types — set, frozenset` for documentation about this class.

For other containers see the built-in `frozenset, list, tuple, and dict` classes, as well as the `collections` module.

```python
setattr(object, name, value)
```

This is the counterpart of `getattr()`. The arguments are an object, a string and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, `setattr(x, 'foobar', 123)` is equivalent to `x.foobar = 123`.

```python
slice(stop)
slice(start, stop[, step])
```

Return a `slice` object representing the set of indices specified by `range(start, stop, step)`. The `start` and `step` arguments default to `None`. Slice objects have read-only data attributes `start, stop` and `step` which merely return the argument values (or their default). They have no other explicit functionality; however they are used by Numerical Python and other third party extensions. Slice objects are also generated when extended indexing syntax is used. For example: `a[start:stop:step]` or `a[start:stop, i]`. See `itertools.islice()` for an alternate version that returns an iterator.

```python
sorted(iterable[, key[, reverse]])
```

Return a new sorted list from the items in `iterable`.

Has two optional arguments which must be specified as keyword arguments.

- `key` specifies a function of one argument that is used to extract a comparison key from each list element: `key=str.lower`. The default value is `None` (compare the elements directly).
- `reverse` is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

Use `functools.cmp_to_key()` to convert an old-style `cmp` function to a `key` function.

For sorting examples and a brief sorting tutorial, see [Sorting HowTo](https://docs.python.org/).

```python
staticmethod(function)
```

Return a static method for `function`.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

```python
class C:
    @staticmethod
    def f(arg1, arg2, ...): ...
```

The `@staticmethod` form is a function decorator — see the description of function definitions in `function` for details.

It can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). The instance is ignored except for its class.

Static methods in Python are similar to those found in Java or C++. Also see `classmethod()` for a variant that is useful for creating alternate class constructors.
For more information on static methods, consult the documentation on the standard type hierarchy in `types`.

```python
str(object='')
```

Return a `str` version of `object`. See `str()` for details.

```python
str(object=b'', encoding='utf-8', errors='strict')
```

`str` is the built-in string class. For general information about strings, see `Text Sequence Type — str`.

```python
sum(iterable[, start])
```

Sums `start` and the items of an `iterable` from left to right and returns the total. `start` defaults to 0. The `iterable`'s items are normally numbers, and the start value is not allowed to be a string.

For some use cases, there are good alternatives to `sum()`. The preferred, fast way to concatenate a sequence of strings is by calling `.join(sequence)`. To add floating point values with extended precision, see `math.fsum()`. To concatenate a series of iterables, consider using `itertools.chain()`.

```python
super([type[, object-or-type]])
```

Return a proxy object that delegates method calls to a parent or sibling class of `type`. This is useful for accessing inherited methods that have been overridden in a class. The search order is same as that used by `getattr()` except that the `type` itself is skipped.

The `__mro__` attribute of the `type` lists the method resolution search order used by both `getattr()` and `super()`. The attribute is dynamic and can change whenever the inheritance hierarchy is updated.

If the second argument is omitted, the super object returned is unbound. If the second argument is an object, `isinstance(obj, type)` must be true. If the second argument is a type, `issubclass(type2, type)` must be true (this is useful for classmethods).

There are two typical use cases for `super`. In a class hierarchy with single inheritance, `super` can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable. This use closely parallels the use of `super` in other programming languages.

The second use case is to support cooperative multiple inheritance in a dynamic execution environment. This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance. This makes it possible to implement “diamond diagrams” where multiple base classes implement the same method. Good design dictates that this method have the same calling signature in every case (because the order of calls is determined at runtime, because that order adapts to changes in the class hierarchy, and because that order can include sibling classes that are unknown prior to runtime).

For both use cases, a typical superclass call looks like this:

```python
class C(B):
    def method(self, arg):
        super().method(arg)  # This does the same thing as:
        # super(C, self).method(arg)
```

Note that `super()` is implemented as part of the binding process for explicit dotted attribute lookups such as `super().__getitem__(name)`. It does so by implementing its own `__getattribute__()` method for searching classes in a predictable order that supports cooperative multiple inheritance. Accordingly, `super()` is undefined for implicit lookups using statements or operators such as `super()[name]`.

Also note that, aside from the zero argument form, `super()` is not limited to use inside methods. The two argument form specifies the arguments exactly and makes the appropriate references. The zero argument form only works inside a class definition, as the compiler fills in the necessary details to correctly retrieve the class being defined, as well as accessing the current instance for ordinary methods.

For practical suggestions on how to design cooperative classes using `super()`, see `guide to using super()`.

```python
tuple([iterable])
```

Rather than being a function, `tuple` is actually an immutable sequence type, as documented in `Tuples` and `Sequence Types — list, tuple, range`.

```python
type(object)
```
The Python Library Reference, Release 3.3.3

**type**(name, bases, dict)

With one argument, return the type of an object. The return value is a type object and generally the same object as returned by `object.__class__`.

The `isinstance()` built-in function is recommended for testing the type of an object, because it takes subclasses into account.

With three arguments, return a new type object. This is essentially a dynamic form of the `class` statement. The `name` string is the class name and becomes the `__name__` attribute; the `bases` tuple itemizes the base classes and becomes the `__bases__` attribute; and the `dict` dictionary is the namespace containing definitions for class body and becomes the `__dict__` attribute. For example, the following two statements create identical `type` objects:

```python
>>> class X:
...     a = 1
...
>>> X = type('X', (object,), dict(a=1))
```

See also `Type Objects`.

**vars**(object)

Return the `__dict__` attribute for a module, class, instance, or any other object with a `__dict__` attribute.

Objects such as modules and instances have an updateable `__dict__` attribute; however, other objects may have write restrictions on their `__dict__` attributes (for example, classes use a dictproxy to prevent direct dictionary updates).

Without an argument, `vars()` acts like `locals()`. Note, the locals dictionary is only useful for reads since updates to the locals dictionary are ignored.

**zip**(iterables)

Make an iterator that aggregates elements from each of the iterables.

Returns an iterator of tuples, where the `i`-th tuple contains the `i`-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted. With a single iterable argument, it returns an iterator of 1-tuples. With no arguments, it returns an empty iterator. Equivalent to:

```python
def zip(*iterables):
    # zip('ABCD', 'xy') --> Ax By
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
                return
            result.append(elem)
        yield tuple(result)
```

The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into `n`-length groups using `zip(*[iter(s)]*n)`.

`zip()` should only be used with unequal length inputs when you don’t care about trailing, unmatched values from the longer iterables. If those values are important, use `itertools.zip_longest()` instead.

`zip()` in conjunction with the `*` operator can be used to unzip a list:

```python
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> list(zipped)
```
[(1, 4), (2, 5), (3, 6)]

>>> x2, y2 = zip(*zip(x, y))
>>> x == list(x2) and y == list(y2)
True

`__import__`(name, globals=None, locals=None, fromlist=(), level=0)

**Note:** This is an advanced function that is not needed in everyday Python programming, unlike `importlib.import_module()`. This function is invoked by the `import` statement. It can be replaced (by importing the `builtins` module and assigning to `builtins.__import__`) in order to change semantics of the `import` statement, but doing so is strongly discouraged as it is usually simpler to use import hooks (see PEP 302) to attain the same goals and does not cause issues with code which assumes the default import implementation is in use. Direct use of `__import__()` is also discouraged in favor of `importlib.import_module()`.

The function imports the module `name`, potentially using the given `globals` and `locals` to determine how to interpret the name in a package context. The `fromlist` gives the names of objects or submodules that should be imported from the module given by `name`. The standard implementation does not use its `locals` argument at all, and uses its `globals` only to determine the package context of the `import` statement.

`level` specifies whether to use absolute or relative imports. 0 (the default) means only perform absolute imports. Positive values for `level` indicate the number of parent directories to search relative to the directory of the module calling `__import__()` (see PEP 328 for the details).

When the `name` variable is of the form `package.module`, normally, the top-level package (the name up till the first dot) is returned, not the module named by `name`. However, when a non-empty `fromlist` argument is given, the module named by `name` is returned.

For example, the statement `import spam` results in bytecode resembling the following code:

```python
spam = __import__('spam', globals(), locals(), [], 0)
```

The statement `import spam.ham` results in this call:

```python
spam = __import__('spam.ham', globals(), locals(), [], 0)
```

Note how `__import__()` returns the toplevel module here because this is the object that is bound to a name by the `import` statement.

On the other hand, the statement `from spam.ham import eggs, sausage as saus` results in

```python
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], 0)
eggs = _temp.eggs
saus = _temp.sausage
```

Here, the `spam.ham` module is returned from `__import__()`). From this object, the names to import are retrieved and assigned to their respective names.

If you simply want to import a module (potentially within a package) by name, use `importlib.import_module()`. Changed in version 3.3: Negative values for `level` are no longer supported (which also changes the default value to 0).
A small number of constants live in the built-in namespace. They are:

**False**

The false value of the `bool` type. Assignments to `False` are illegal and raise a `SyntaxError`.

**True**

The true value of the `bool` type. Assignments to `True` are illegal and raise a `SyntaxError`.

**None**

The sole value of the type `NoneType`. `None` is frequently used to represent the absence of a value, as when default arguments are not passed to a function. Assignments to `None` are illegal and raise a `SyntaxError`.

**NotImplemented**

Special value which can be returned by the “rich comparison” special methods (`__eq__()`, `__lt__()`, and friends), to indicate that the comparison is not implemented with respect to the other type.

**Ellipsis**

The same as `. . .`. Special value used mostly in conjunction with extended slicing syntax for user-defined container data types.

**__debug__**

This constant is true if Python was not started with an `-O` option. See also the `assert` statement.

---

**Note:** The names `None`, `False`, `True` and `__debug__` cannot be reassigned (assignments to them, even as an attribute name, raise `SyntaxError`), so they can be considered “true” constants.

### 3.1 Constants added by the `site` module

The `site` module (which is imported automatically during startup, except if the `-S` command-line option is given) adds several constants to the built-in namespace. They are useful for the interactive interpreter shell and should not be used in programs.

- **`quit(code=None)`**
- **`exit(code=None)`**
  - Objects that when printed, print a message like “Use quit() or Ctrl-D (i.e. EOF) to exit”, and when called, raise `SystemExit` with the specified exit code.
- **`copyright`**
- **`license`**
- **`credits`**
  - Objects that when printed, print a message like “Type license() to see the full license text”, and when called, display the corresponding text in a pager-like fashion (one screen at a time).
The following sections describe the standard types that are built into the interpreter.

The principal built-in types are numerics, sequences, mappings, classes, instances and exceptions.

Some collection classes are mutable. The methods that add, subtract, or rearrange their members in place, and don’t return a specific item, never return the collection instance itself but None.

Some operations are supported by several object types; in particular, practically all objects can be compared, tested for truth value, and converted to a string (with the repr() function or the slightly different str() function). The latter function is implicitly used when an object is written by the print() function.

4.1 Truth Value Testing

Any object can be tested for truth value, for use in an if or while condition or as operand of the Boolean operations below. The following values are considered false:

- None
- False
- zero of any numeric type, for example, 0, 0.0, 0j.
- any empty sequence, for example, "", (), [].
- any empty mapping, for example, {}.
- instances of user-defined classes, if the class defines a __bool__() or __len__() method, when that method returns the integer zero or bool value False.¹

All other values are considered true — so objects of many types are always true.

Operations and built-in functions that have a Boolean result always return 0 or False for false and 1 or True for true, unless otherwise stated. (Important exception: the Boolean operations or and and always return one of their operands.)

4.2 Boolean Operations — and, or, not

These are the Boolean operations, ordered by ascending priority:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x or y</td>
<td>if x is false, then y, else x</td>
<td>(1)</td>
</tr>
<tr>
<td>x and y</td>
<td>if x is false, then x, else y</td>
<td>(2)</td>
</tr>
<tr>
<td>not x</td>
<td>if x is false, then True, else False</td>
<td>(3)</td>
</tr>
</tbody>
</table>

¹ Additional information on these special methods may be found in the Python Reference Manual (customization).
Notes:

1. This is a short-circuit operator, so it only evaluates the second argument if the first one is `False`.
2. This is a short-circuit operator, so it only evaluates the second argument if the first one is `True`.
3. `not` has a lower priority than non-Boolean operators, so `not a == b` is interpreted as `not (a == b)`, and `a == not b` is a syntax error.

### 4.3 Comparisons

There are eight comparison operations in Python. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily; for example, `x < y <= z` is equivalent to `x < y` and `y <= z`, except that `y` is evaluated only once (but in both cases `z` is not evaluated at all when `x < y` is found to be false).

This table summarizes the comparison operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>strictly less than</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>less than or equal</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>strictly greater than</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>greater than or equal</td>
</tr>
<tr>
<td><code>==</code></td>
<td>equal</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>not equal</td>
</tr>
<tr>
<td><code>is</code></td>
<td>object identity</td>
</tr>
<tr>
<td><code>is not</code></td>
<td>negated object identity</td>
</tr>
</tbody>
</table>

Objects of different types, except different numeric types, never compare equal. Furthermore, some types (for example, function objects) support only a degenerate notion of comparison where any two objects of that type are unequal. The `<`, `<=`, `>` and `>=` operators will raise a `TypeError` exception when comparing a complex number with another built-in numeric type, when the objects are of different types that cannot be compared, or in other cases where there is no defined ordering.

Non-identical instances of a class normally compare as non-equal unless the class defines the `__eq__()` method.

Instances of a class cannot be ordered with respect to other instances of the same class, or other types of object, unless the class defines enough of the methods `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()` (in general, `__lt__()` and `__eq__()` are sufficient, if you want the conventional meanings of the comparison operators).

The behavior of the `is` and `is not` operators cannot be customized; also they can be applied to any two objects and never raise an exception.

Two more operations with the same syntactic priority, `in` and `not in`, are supported only by sequence types (below).

### 4.4 Numeric Types — `int`, `float`, `complex`

There are three distinct numeric types: `integers`, `floating point numbers`, and `complex numbers`. In addition, Booleans are a subtype of integers. Integers have unlimited precision. Floating point numbers are usually implemented using `double` in C; information about the precision and internal representation of floating point numbers for the machine on which your program is running is available in `sys.float_info`. Complex numbers have a real and imaginary part, which are each a floating point number. To extract these parts from a complex number `z`, use `z.real` and `z.imag`. (The standard library includes additional numeric types, `fractions` that hold rationals, and `decimal` that hold floating-point numbers with user-definable precision.)

Numbers are created by numeric literals or as the result of built-in functions and operators. Unadorned integer literals (including hex, octal and binary numbers) yield integers. Numeric literals containing a decimal point or an exponent sign yield floating point numbers. Appending ‘`j`’ or ‘`J`’ to a numeric literal yields an imaginary
number (a complex number with a zero real part) which you can add to an integer or float to get a complex number with real and imaginary parts.

Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the “narrower” type is widened to that of the other, where integer is narrower than floating point, which is narrower than complex. Comparisons between numbers of mixed type use the same rule. ² The constructors int(), float(), and complex() can be used to produce numbers of a specific type.

All numeric types (except complex) support the following operations, sorted by ascending priority (operations in the same box have the same priority; all numeric operations have a higher priority than comparison operations):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
<th>Full documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>x + y</td>
<td>sum of x and y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x - y</td>
<td>difference of x and y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x * y</td>
<td>product of x and y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x / y</td>
<td>quotient of x and y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x // y</td>
<td>floored quotient of x and y</td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>x % y</td>
<td>remainder of x / y</td>
<td>(2)</td>
<td></td>
</tr>
<tr>
<td>-x</td>
<td>x negated</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+x</td>
<td>x unchanged</td>
<td></td>
<td></td>
</tr>
<tr>
<td>abs(x)</td>
<td>absolute value or magnitude of x</td>
<td></td>
<td>abs()</td>
</tr>
<tr>
<td>int(x)</td>
<td>x converted to integer</td>
<td>(3)</td>
<td>int()</td>
</tr>
<tr>
<td>float(x)</td>
<td>x converted to floating point</td>
<td>(4)</td>
<td>float()</td>
</tr>
<tr>
<td>complex(re, im)</td>
<td>a complex number with real part re, imaginary part im. im</td>
<td>(6)</td>
<td>complex()</td>
</tr>
<tr>
<td>c.conjugate()</td>
<td>conjugate of the complex number c</td>
<td>(6)</td>
<td></td>
</tr>
<tr>
<td>divmod(x, y)</td>
<td>the pair (x // y, x % y)</td>
<td>(2)</td>
<td>divmod()</td>
</tr>
<tr>
<td>pow(x, y)</td>
<td>x to the power y</td>
<td>(5)</td>
<td>pow()</td>
</tr>
<tr>
<td>x ** y</td>
<td>x to the power y</td>
<td>(5)</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. Also referred to as integer division. The resultant value is a whole integer, though the result’s type is not necessarily int. The result is always rounded towards minus infinity: 1//2 is 0, (-1)//2 is -1, 1//-2 is -1, and (-1)//(-2) is 0.

2. Not for complex numbers. Instead convert to floats using abs() if appropriate.

3. Conversion from floating point to integer may round or truncate as in C; see functions math.floor() and math.ceil() for well-defined conversions.

4. float also accepts the strings “nan” and “inf” with an optional prefix “+” or “-” for Not a Number (NaN) and positive or negative infinity.

5. Python defines pow(0, 0) and 0 ** 0 to be 1, as is common for programming languages.

6. The numeric literals accepted include the digits 0 to 9 or any Unicode equivalent (code points with the Nd property).

See http://www.unicode.org/Public/6.0.0/ucd/extracted/DerivedNumericType.txt for a complete list of code points with the Nd property.

All numbers.Real types (int and float) also include the following operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>math.trunc(x)</td>
<td>x truncated to Integral</td>
<td></td>
</tr>
<tr>
<td>round(x[, n])</td>
<td>x rounded to n digits, rounding half to even. If n is omitted, it defaults to 0.</td>
<td></td>
</tr>
<tr>
<td>math.floor(x)</td>
<td>the greatest integral float &lt;= x</td>
<td></td>
</tr>
<tr>
<td>math.ceil(x)</td>
<td>the least integral float &gt;= x</td>
<td></td>
</tr>
</tbody>
</table>

For additional numeric operations see the math and cmath modules.

² As a consequence, the list [1, 2] is considered equal to [1.0, 2.0], and similarly for tuples.
4.4.1 Bitwise Operations on Integer Types

Bitwise operations only make sense for integers. Negative numbers are treated as their 2’s complement value (this assumes a sufficiently large number of bits that no overflow occurs during the operation).

The priorities of the binary bitwise operations are all lower than the numeric operations and higher than the comparisons; the unary operation ~ has the same priority as the other unary numeric operations (+ and −).

This table lists the bitwise operations sorted in ascending priority (operations in the same box have the same priority):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>bitwise or of x and y</td>
</tr>
<tr>
<td>x ^ y</td>
<td>bitwise exclusive or of x and y</td>
<td></td>
</tr>
<tr>
<td>x &amp; y</td>
<td>bitwise and of x and y</td>
<td></td>
</tr>
<tr>
<td>x &lt;&lt; n</td>
<td>x shifted left by n bits</td>
<td></td>
</tr>
<tr>
<td>x &gt;&gt; n</td>
<td>x shifted right by n bits</td>
<td></td>
</tr>
<tr>
<td>~x</td>
<td>the bits of x inverted</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. Negative shift counts are illegal and cause a `ValueError` to be raised.
2. A left shift by n bits is equivalent to multiplication by $\text{pow}(2, \ n)$ without overflow check.
3. A right shift by n bits is equivalent to division by $\text{pow}(2, \ n)$ without overflow check.

4.4.2 Additional Methods on Integer Types

The int type implements the `numbers.Integral abstract base class`. In addition, it provides one more method:

```python
int.bit_length()
```

Return the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros:

```python
>>> n = -37
>>> bin(n)
'\-0b100101'
>>> n.bit_length()
6
```

More precisely, if x is nonzero, then x.bit_length() is the unique positive integer k such that $2^{*(k-1)} <= \text{abs}(x) < 2^{*k}$. Equivalently, when \text{abs}(x) is small enough to have a correctly rounded logarithm, then $k = 1 + \text{int}(\log(\text{abs}(x), \ 2))$. If x is zero, then x.bit_length() returns 0.

Equivalent to:

```python
def bit_length(self):
    s = bin(self)  # binary representation: bin(-37) --> '\-0b100101'
    s = s.lstrip('\-0b')  # remove leading zeros and minus sign
    return len(s)  # len('100101') --> 6
```

New in version 3.1.

```python
int.to_bytes(length, byteorder, *, signed=False)
```

Return an array of bytes representing an integer.

```python
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
```
The integer is represented using length bytes. An `OverflowError` is raised if the integer is not representable with the given number of bytes.

The `byteorder` argument determines the byte order used to represent the integer. If `byteorder` is "big", the most significant byte is at the beginning of the byte array. If `byteorder` is "little", the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value.

The `signed` argument determines whether two’s complement is used to represent the integer. If `signed` is False and a negative integer is given, an `OverflowError` is raised. The default value for `signed` is False. New in version 3.2.

```python
>>> x = 1000
>>> x.to_bytes((x.bit_length() // 8) + 1, byteorder='little')
b'\xe8\x03'
```

New in version 3.2.

The argument `bytes` must either be a `bytes-like object` or an iterable producing bytes.

Two methods support conversion to and from hexadecimal strings. Since Python’s floats are stored internally as binary numbers, converting a float to or from a `decimal` string usually involves a small rounding error. In contrast,

```python
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
65536
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

4.4.3 Additional Methods on Float

The float type implements the `numbers.Real abstract base class`. float also has the following additional methods.

```python
float.as_integer_ratio()
```

Return a pair of integers whose ratio is exactly equal to the original float and with a positive denominator. Raises `OverflowError` on infinities and a `ValueError` on NaNs.

```python
float.is_integer()
```

Return True if the float instance is finite with integral value, and False otherwise:

```python
>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```
hexadecimal strings allow exact representation and specification of floating-point numbers. This can be useful when debugging, and in numerical work.

```python
float.hex()
```

Return a representation of a floating-point number as a hexadecimal string. For finite floating-point numbers, this representation will always include a leading 0x and a trailing p and exponent.

```python
classmethod float.fromhex(s)
```

Class method to return the float represented by a hexadecimal string `s`. The string `s` may have leading and trailing whitespace.

Note that `float.hex()` is an instance method, while `float.fromhex()` is a class method.

A hexadecimal string takes the form:

```
[sign] ['0x'] integer ['.' fraction] ['p' exponent]
```

where the optional `sign` may be either + or -. `integer` and `fraction` are strings of hexadecimal digits, and `exponent` is a decimal integer with an optional leading sign. Case is not significant, and there must be at least one hexadecimal digit in either the integer or the fraction. This syntax is similar to the syntax specified in section 6.4.4.2 of the C99 standard, and also to the syntax used in Java 1.5 onwards. In particular, the output of `float.hex()` is usable as a hexadecimal floating-point literal in C or Java code, and hexadecimal strings produced by C’s %a format character or Java’s `Double.toHexString` are accepted by `float.fromhex()`.

Note that the exponent is written in decimal rather than hexadecimal, and that it gives the power of 2 by which to multiply the coefficient. For example, the hexadecimal string `0x3.a7p10` represents the floating-point number `(3 + 10./16 + 7./16**2) * 2.0**10`, or `3740.0`:

```python
>>> float.fromhex('0x3.a7p10')
3740.0
```

Applying the reverse conversion to `3740.0` gives a different hexadecimal string representing the same number:

```python
>>> float.hex(3740.0)
'0x1.d380000000000p+11'
```

### 4.4.4 Hashing of numeric types

For numbers `x` and `y`, possibly of different types, it’s a requirement that `hash(x) == hash(y)` whenever `x == y` (see the `__hash__()` method documentation for more details). For ease of implementation and efficiency across a variety of numeric types (including `int`, `float`, `decimal.Decimal` and `fractions.Fraction`) Python’s hash for numeric types is based on a single mathematical function that’s defined for any rational number, and hence applies to all instances of `int` and `fractions.Fraction`, and all finite instances of `float` and `decimal.Decimal`. Essentially, this function is given by reduction modulo $P$ for a fixed prime $P$. The value of $P$ is made available to Python as the modulus attribute of `sys.hash_info`.

**CPython implementation detail:** Currently, the prime used is $P = 2**31 - 1$ on machines with 32-bit C longs and $P = 2**61 - 1$ on machines with 64-bit C longs.

Here are the rules in detail:

- If $x = m / n$ is a nonnegative rational number and $n$ is not divisible by $P$, define $\text{hash}(x) = m * \text{invmod}(n, P) \mod P$, where `invmod(n, P)` gives the inverse of $n$ modulo $P$.
- If $x = m / n$ is a nonnegative rational number and $n$ is divisible by $P$ (but $m$ is not) then $n$ has no inverse modulo $P$ and the rule above doesn’t apply; in this case define $\text{hash}(x)$ to be the constant value `sys.hash_info.inf`.
- If $x = m / n$ is a negative rational number define $\text{hash}(x)$ as $-\text{hash}(-x)$. If the resulting hash is $-1$, replace it with $-2$.
- The particular values `sys.hash_info.inf`, $-\text{sys.hash_info.inf}$ and `sys.hash_info.nan` are used as hash values for positive infinity, negative infinity, or nans (respectively). (All hashable nans have the same hash value.)
For a complex number \( z \), the hash values of the real and imaginary parts are combined by computing \( \text{hash}(z.\text{real}) + \text{sys.hash_info.imag} \times \text{hash}(z.\text{imag}) \), reduced modulo \( 2^{\text{sys.hash_info.width}} \) so that it lies in range(-\( 2^{\text{sys.hash_info.width} - 1} \), \( 2^{\text{sys.hash_info.width} - 1} \)). Again, if the result is \(-1\), it's replaced with \(-2\).

To clarify the above rules, here's some example Python code, equivalent to the built-in hash, for computing the hash of a rational number, float, or complex:

```python
import sys, math

def hash_fraction(m, n):
    """Compute the hash of a rational number \( m / n \).

    Assumes \( m \) and \( n \) are integers, with \( n \) positive.
    Equivalent to hash(fractions.Fraction(m, n)).
    """

    P = sys.hash_info.modulus  # Remove common factors of \( P \). (Unnecessary if \( m \) and \( n \) already coprime.)
    while m % P == n % P == 0:
        m, n = m // P, n // P
    if n % P == 0:
        hash_ = sys.hash_info.inf
    else:
        # Fermat's Little Theorem: \( \text{pow}(n, P-1, P) \) is 1, so \( \text{pow}(n, P-2, P) \) gives the inverse of \( n \) modulo \( P \).
        hash_ = (abs(m) % P) * pow(n, P - 2, P) % P
    if m < 0:
        hash_ = -hash_
    if hash_ == -1:
        hash_ = -2
    return hash_

def hash_float(x):
    """Compute the hash of a float \( x \)."""

    if math.isnan(x):
        return sys.hash_info.nan
    elif math.isinf(x):
        return sys.hash_info.inf if x > 0 else -sys.hash_info.inf
    else:
        return hash_fraction(*x.as_integer_ratio())

def hash_complex(z):
    """Compute the hash of a complex number \( z \)."""

    hash_ = hash_float(z.real) + sys.hash_info.imag * hash_float(z.imag)
    # do a signed reduction modulo \( 2^{\text{sys.hash_info.width}} \)
    M = 2**(sys.hash_info.width - 1)
    hash_ = (hash_ & (M - 1)) - (hash & M)
    if hash_ == -1:
        hash_ = -2
    return hash_
```

4.4. Numeric Types — int, float, complex
4.5 Iterator Types

Python supports a concept of iteration over containers. This is implemented using two distinct methods; these are used to allow user-defined classes to support iteration. Sequences, described below in more detail, always support the iteration methods.

One method needs to be defined for container objects to provide iteration support:

```python
class container:
    __iter__(self):
        return self
```

This method returns an iterator object. The object is required to support the iterator protocol described below. If a container supports different types of iteration, additional methods can be provided to specifically request iterators for those iteration types. (An example of an object supporting multiple forms of iteration would be a tree structure which supports both breadth-first and depth-first traversal.)

The iterator objects themselves are required to support the following two methods, which together form the iterator protocol:

```python
class iterator:
    __iter__(self):
        return self

    __next__(self):
        raise StopIteration
```

This method returns the next item from the container. If there are no further items, raise the `StopIteration` exception.

Python defines several iterator objects to support iteration over general and specific sequence types, dictionaries, and other more specialized forms. The specific types are not important beyond their implementation of the iterator protocol.

Once an iterator's `__next__()` method raises `StopIteration`, it must continue to do so on subsequent calls. Implementations that do not obey this property are deemed broken.

4.5.1 Generator Types

Python's generators provide a convenient way to implement the iterator protocol. If a container object's `__iter__()` method is implemented as a generator, it will automatically return an iterator object (technically, a generator object) supplying the `__iter__()` and `__next__()` methods. More information about generators can be found in the documentation for the `yield` expression.

4.6 Sequence Types — list, tuple, range

There are three basic sequence types: lists, tuples, and range objects. Additional sequence types tailored for processing of binary data and text strings are described in dedicated sections.

4.6.1 Common Sequence Operations

The operations in the following table are supported by most sequence types, both mutable and immutable. The `collections.abc.Sequence` ABC is provided to make it easier to correctly implement these operations on custom sequence types.

This table lists the sequence operations sorted in ascending priority (operations in the same box have the same priority). In the table, `s` and `t` are sequences of the same type, `n`, `i`, `j` and `k` are integers and `x` is an arbitrary object that meets any type and value restrictions imposed by `s`.

The `in` and `not in` operations have the same priorities as the comparison operations. The `+` (concatenation) and `*` (repetition) operations have the same priority as the corresponding numeric operations.
Sequences of the same type also support comparisons. In particular, tuples and lists are compared lexicographically by comparing corresponding elements. This means that to compare equal, every element must compare equal and the two sequences must be of the same type and have the same length. (For full details see comparisons in the language reference.)

Notes:

1. While the in and not in operations are used only for simple containment testing in the general case, some specialised sequences (such as str, bytes and bytearray) also use them for subsequence testing:

```python
>>> "gg" in "eggs"
True
```

2. Values of n less than 0 are treated as 0 (which yields an empty sequence of the same type as s). Note also that the copies are shallow; nested structures are not copied. This often haunts new Python programmers; consider:

```python
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

What has happened is that [[]] is a one-element list containing an empty list, so all three elements of [[]] * 3 are (pointers to) this single empty list. Modifying any of the elements of lists modifies this single list. You can create a list of different lists this way:

```python
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

3. If i or j is negative, the index is relative to the end of the string: len(s) + i or len(s) + j is substituted. But note that -0 is still 0.

4. The slice of s from i to j is defined as the sequence of items with index k such that i <= k < j. If i or j is greater than len(s), use len(s). If i is omitted or None, use 0. If j is omitted or None, use len(s). If i is greater than or equal to j, the slice is empty.

5. The slice of s from i to j with step k is defined as the sequence of items with index x = i + n*k such that 0 <= n < (j-i)/k. In other words, the indices are i, i+k, i+2*k, i+3*k and so on, stopping when j is reached (but never including j). If i or j is greater than len(s), use len(s). If i or j are omitted or None, they become "end" values (which end depends on the sign of k). Note, k cannot be zero. If k is None, it is treated like 1.

6. Concatenating immutable sequences always results in a new object. This means that building up a sequence by repeated concatenation will have a quadratic runtime cost in the total sequence length. To get a linear
runtime cost, you must switch to one of the alternatives below:

- if concatenating str objects, you can build a list and use str.join() at the end or else write to a io.StringIO instance and retrieve its value when complete
- if concatenating bytes objects, you can similarly use bytes.join() or io.BytesIO, or you can do in-place concatenation with a bytearray object. bytearray objects are mutable and have an efficient overallocation mechanism
- if concatenating tuple objects, extend a list instead
- for other types, investigate the relevant class documentation

7. Some sequence types (such as range) only support item sequences that follow specific patterns, and hence don’t support sequence concatenation or repetition.

8. index raises ValueError when x is not found in s. When supported, the additional arguments to the index method allow efficient searching of subsections of the sequence. Passing the extra arguments is roughly equivalent to using s[i:j].index(x), only without copying any data and with the returned index being relative to the start of the sequence rather than the start of the slice.

4.6.2 Immutable Sequence Types

The only operation that immutable sequence types generally implement that is not also implemented by mutable sequence types is support for the hash() built-in.

This support allows immutable sequences, such as tuple instances, to be used as dict keys and stored in set and frozenset instances.

Attempting to hash an immutable sequence that contains unhashable values will result in TypeError.

4.6.3 Mutable Sequence Types

The operations in the following table are defined on mutable sequence types. The collections.abc.MutableSequence ABC is provided to make it easier to correctly implement these operations on custom sequence types.

In the table s is an instance of a mutable sequence type, t is any iterable object and x is an arbitrary object that meets any type and value restrictions imposed by s (for example, bytearray only accepts integers that meet the value restriction 0 <= x <= 255).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>s[i] = x</td>
<td>item i of s is replaced by x</td>
<td></td>
</tr>
<tr>
<td>s[i:j] = t</td>
<td>slice of s from i to j is replaced by the contents of the iterable t</td>
<td></td>
</tr>
<tr>
<td>del s[i:j]</td>
<td>same as s[i:j] = []</td>
<td></td>
</tr>
<tr>
<td>s[i:j:k] = t</td>
<td>the elements of s[i:j:k] are replaced by those of t</td>
<td></td>
</tr>
<tr>
<td>del s[i:j:k]</td>
<td>removes the elements of s[i:j:k] from the list</td>
<td>(1)</td>
</tr>
<tr>
<td>s.append(x)</td>
<td>appends x to the end of the sequence (same as s[len(s):len(s)] = [x])</td>
<td></td>
</tr>
<tr>
<td>s.clear()</td>
<td>removes all items from s (same as del s[:])</td>
<td>(5)</td>
</tr>
<tr>
<td>s.copy()</td>
<td>creates a shallow copy of s (same as s[:])</td>
<td>(5)</td>
</tr>
<tr>
<td>s.extend(t)</td>
<td>extends s with the contents of t (same as s[len(s):len(s)] = t)</td>
<td></td>
</tr>
<tr>
<td>s.insert(i, x)</td>
<td>inserts x into s at the index given by i (same as s[i:i] = [x])</td>
<td>(2)</td>
</tr>
<tr>
<td>s.pop(i)</td>
<td>retrieves the item at i and also removes it from s</td>
<td></td>
</tr>
<tr>
<td>s.remove(x)</td>
<td>remove the first item from s where s[i] == x</td>
<td>(3)</td>
</tr>
<tr>
<td>s.reverse()</td>
<td>reverses the items of s in place</td>
<td>(4)</td>
</tr>
</tbody>
</table>

Notes:

1. t must have the same length as the slice it is replacing.
2. The optional argument i defaults to -1, so that by default the last item is removed and returned.
3. remove raises ValueError when x is not found in s.
4. The `reverse()` method modifies the sequence in place for economy of space when reversing a large sequence. To remind users that it operates by side effect, it does not return the reversed sequence.

5. `clear()` and `copy()` are included for consistency with the interfaces of mutable containers that don’t support slicing operations (such as `dict` and `set`) New in version 3.3: `clear()` and `copy()` methods.

4.6.4 Lists

Lists are mutable sequences, typically used to store collections of homogeneous items (where the precise degree of similarity will vary by application).

```python
class list ([iterable])
```

Lists may be constructed in several ways:

- Using a pair of square brackets to denote the empty list: `[ ]`
- Using square brackets, separating items with commas: `[a], [a, b, c]`
- Using a list comprehension: `[x for x in iterable]`
- Using the type constructor: `list() or list(iterable)`

The constructor builds a list whose items are the same and in the same order as `iterable`’s items. `iterable` may be either a sequence, a container that supports iteration, or an iterator object. If `iterable` is already a list, a copy is made and returned, similar to `iterable[:]`. For example, `list(‘abc’)` returns `[‘a’, ‘b’, ‘c’]` and `list((1, 2, 3))` returns `[1, 2, 3]`. If no argument is given, the constructor creates a new empty list, `[ ]`.

Many other operations also produce lists, including the `sorted()` built-in.

Lists implement all of the `common` and `mutable` sequence operations. Lists also provide the following additional method:

```python
sort (*, key=None, reverse=None)
```

This method sorts the list in place, using only `<` comparisons between items. Exceptions are not suppressed - if any comparison operations fail, the entire sort operation will fail (and the list will likely be left in a partially modified state).

`key` specifies a function of one argument that is used to extract a comparison key from each list element (for example, `key=str.lower`). The key corresponding to each item in the list is calculated once and then used for the entire sorting process. The default value of `None` means that list items are sorted directly without calculating a separate key value.

The `functools.cmp_to_key()` utility is available to convert a 2.x style `cmp` function to a `key` function.

`reverse` is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

This method modifies the sequence in place for economy of space when sorting a large sequence. To remind users that it operates by side effect, it does not return the sorted sequence (use `sorted()` to explicitly request a new sorted list instance).

The `sort()` method is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

**CPython implementation detail:** While a list is being sorted, the effect of attempting to mutate, or even inspect, the list is undefined. The C implementation of Python makes the list appear empty for the duration, and raises `ValueError` if it can detect that the list has been mutated during a sort.
4.6.5 Tuples

Tuples are immutable sequences, typically used to store collections of heterogeneous data (such as the 2-tuples produced by the `enumerate()` built-in). Tuples are also used for cases where an immutable sequence of homogeneous data is needed (such as allowing storage in a `set` or `dict` instance).

```python
class tuple(iterable)
```

Tuples may be constructed in a number of ways:

- Using a pair of parentheses to denote the empty tuple: `()`
- Using a trailing comma for a singleton tuple: `a,` or `(a,)`
- Separating items with commas: `a, b, c` or `(a, b, c)`
- Using the `tuple()` built-in: `tuple()` or `tuple(iterable)`

The constructor builds a tuple whose items are the same and in the same order as `iterable`'s items. `iterable` may be either a sequence, a container that supports iteration, or an iterator object. If `iterable` is already a tuple, it is returned unchanged. For example, `tuple('abc')` returns `('a', 'b', 'c')` and `tuple([1, 2, 3])` returns `(1, 2, 3)`. If no argument is given, the constructor creates a new empty tuple, `()`. Note that it is actually the comma which makes a tuple, not the parentheses. The parentheses are optional, except in the empty tuple case, or when they are needed to avoid syntactic ambiguity. For example, `f(a, b, c)` is a function call with three arguments, while `f((a, b, c))` is a function call with a 3-tuple as the sole argument.

Tuples implement all of the `common` sequence operations.

For heterogeneous collections of data where access by name is clearer than access by index, `collections.namedtuple()` may be a more appropriate choice than a simple tuple object.

4.6.6 Ranges

The `range` type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in `for` loops.

```python
class range(stop)
class range(start, stop[, step])
```

The arguments to the range constructor must be integers (either built-in `int` or any object that implements the `__index__` special method). If the `step` argument is omitted, it defaults to 1. If the `start` argument is omitted, it defaults to 0. If `step` is zero, `ValueError` is raised.

For a positive `step`, the contents of a range are determined by the formula `r[i] = start + step*i` where `i >= 0` and `r[i] < stop`.

For a negative `step`, the contents of the range are still determined by the formula `r[i] = start + step*i`, but the constraints are `i > 0` and `r[i] > stop`.

A range object will be empty if `r[0]` does not meet the value constraint. Ranges do support negative indices, but these are interpreted as indexing from the end of the sequence determined by the positive indices.

Ranges containing absolute values larger than `sys.maxsize` are permitted but some features (such as `len()`) may raise `OverflowError`.

Range examples:

```python
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
```
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]

Ranges implement all of the common sequence operations except concatenation and repetition (due to the fact that range objects can only represent sequences that follow a strict pattern and repetition and concatenation will usually violate that pattern).

The advantage of the range type over a regular list or tuple is that a range object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the start, stop and step values, calculating individual items and subranges as needed).

Range objects implement the collections.abc.Sequence ABC, and provide features such as containment tests, element index lookup, slicing and support for negative indices (see Sequence Types — list, tuple, range):

```python
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[5:]
range(0, 10, 2)
>>> r[-1]
18
```

Testing range objects for equality with == and != compares them as sequences. That is, two range objects are considered equal if they represent the same sequence of values. (Note that two range objects that compare equal might have different start, stop and step attributes, for example range(0) == range(2, 1, 3) or range(0, 3, 2) == range(0, 4, 2).) Changed in version 3.2: Implement the Sequence ABC. Support slicing and negative indices. Test int objects for membership in constant time instead of iterating through all items. Changed in version 3.3: Define ‘==’ and ‘!’ to compare range objects based on the sequence of values they define (instead of comparing based on object identity). New in version 3.3: The start, stop and step attributes.

### 4.7 Text Sequence Type — str

Textual data in Python is handled with str objects, or strings. Strings are immutable sequences of Unicode code points. String literals are written in a variety of ways:

- Single quotes: 'allows embedded "double" quotes'
- Double quotes: "allows embedded ‘single’ quotes".
- Triple quoted: '''Three single quotes'', ""Three double quotes"''

Triple quoted strings may span multiple lines - all associated whitespace will be included in the string literal.

String literals that are part of a single expression and have only whitespace between them will be implicitly converted to a single string literal. That is, ("spam " "eggs") == "spam eggs".

4.7. Text Sequence Type — str 37
See *strings* for more about the various forms of string literal, including supported escape sequences, and the `r` ("raw") prefix that disables most escape sequence processing.

Strings may also be created from other objects using the `str` constructor.

Since there is no separate "character" type, indexing a string produces strings of length 1. That is, for a non-empty string `s`, `s[0] == s[0:1]`.

There is also no mutable string type, but `str.join()` or `io.StringIO` can be used to efficiently construct strings from multiple fragments. Changed in version 3.3: For backwards compatibility with the Python 2 series, the `u` prefix is once again permitted on string literals. It has no effect on the meaning of string literals and cannot be combined with the `r` prefix.

```
class str(object=’’)
class str(object=b’’, encoding=’utf-8’, errors=’strict’)
```

Return a string version of `object`. If `object` is not provided, returns the empty string. Otherwise, the behavior of `str()` depends on whether `encoding` or `errors` is given, as follows.

If neither `encoding` nor `errors` is given, `str(object)` returns `object.__str__()`, which is the "informal" or nicely printable string representation of `object`. For string objects, this is the string itself. If `object` does not have a `__str__()` method, then `str()` falls back to returning `repr(object)`.

If at least one of `encoding` or `errors` is given, `object` should be a bytes-like object (e.g. `bytes` or `bytearray`). In this case, if `object` is a bytes (or bytearray) object, then `str(bytes, encoding, errors)` is equivalent to `bytes.decode(encoding, errors)`. Otherwise, the bytes object underlying the buffer object is obtained before calling `bytes.decode()`. See *Binary Sequence Types — bytes, bytearray, memoryview and buffer objects* for information on buffer objects.

Passing a bytes object to `str()` without the `encoding` or `errors` arguments falls under the first case of returning the informal string representation (see also the `-b` command-line option to Python). For example:

```
>>> str(b’Zoot!’
"b’Zoot!’"
```

For more information on the `str` class and its methods, see *Text Sequence Type — str* and the *String Methods* section below. To output formatted strings, see the *String Formatting* section. In addition, see the *Text Processing Services* section.

### 4.7.1 String Methods

Strings implement all of the common sequence operations, along with the additional methods described below.

Strings also support two styles of string formatting, one providing a large degree of flexibility and customization (see `str.format()`, *Format String Syntax* and *String Formatting*) and the other based on C `printf` style formatting that handles a narrower range of types and is slightly harder to use correctly, but is often faster for the cases it can handle (*printf-style String Formatting*).

The *Text Processing Services* section of the standard library covers a number of other modules that provide various text related utilities (including regular expression support in the `re` module).

```
str.capitalize()
str.casefold()
str.center(width[, fillchar])
```

Return a copy of the string with its first character capitalized and the rest lowercased.

Return a casefolded copy of the string. Casefolded strings may be used for caseless matching.

Casefolding is similar to loweringcase but more aggressive because it is intended to remove all case distinctions in a string. For example, the German lowercase letter ‘ß’ is equivalent to "ss". Since it is already lowercase, `lower()` would do nothing to ‘ß’; `casefold()` converts it to "ss".

The casefolding algorithm is described in section 3.13 of the Unicode Standard. New in version 3.3.

Return centered in a string of length `width`. Padding is done using the specified `fillchar` (default is a space).
**str.count**(sub[, start[, end]])

Return the number of non-overlapping occurrences of substring sub in the range [start, end]. Optional arguments start and end are interpreted as in slice notation.

**str.encode**(encoding="utf-8", errors="strict")

Return an encoded version of the string as a bytes object. Default encoding is ‘utf-8’. errors may be given to set a different error handling scheme. The default for errors is ‘strict’, meaning that encoding errors raise a UnicodeError. Other possible values are ‘ignore’, ‘replace’, ‘xmlcharrefreplace’, ‘backslashreplace’ and any other name registered via codecs.register_error(), see section Codec Base Classes. For a list of possible encodings, see section Standard Encodings. Changed in version 3.1: Support for keyword arguments added.

**str.endswith**(suffix[, start[, end]])

Return True if the string ends with the specified suffix, otherwise return False. suffix can also be a tuple of suffixes to look for. With optional start, test beginning at that position. With optional end, stop comparing at that position.

**str.expandtabs**(tabsize)

Return a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size. Tab positions occur every tabsize characters (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the string, the current column is set to zero and the string is examined character by character. If the character is a tab (\t), one or more space characters are inserted in the result until the current column is equal to the next tab position. (The tab character itself is not copied.) If the character is a newline (\n) or return (\r), it is copied and the current column is reset to zero. Any other character is copied unchanged and the current column is incremented by one regardless of how the character is represented when printed.

```python
>>> '01\t012\t0123\t01234'.expandtabs()
'01 012 0123 01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01 012 0123 01234'
```

**str.find**(sub[, start[, end]])

Return the lowest index in the string where substring sub is found, such that sub is contained in the slice s[start:end]. Optional arguments start and end are interpreted as in slice notation. Return -1 if sub is not found.

**Note:** The find() method should be used only if you need to know the position of sub. To check if sub is a substring or not, use the in operator:

```python
>>> 'Py' in 'Python'
True
```

**str.format**(args, **kwargs)

Perform a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces { }. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.

```python
>>> "The sum of 1 + 2 is {0}\".format(1+2)
'The sum of 1 + 2 is 3\'
```

See Format String Syntax for a description of the various formatting options that can be specified in format strings.

**str.format_map**(mapping)

Similar to str.format(**mapping), except that mapping is used directly and not copied to a dict. This is useful if for example mapping is a dict subclass.
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
...     >>> '{name} was born in {country}'.format_map(Default(name='Guido'))
'Guido was born in country'

New in version 3.2.

str.index(sub[, start[, end]]])
Like find(), but raise ValueError when the substring is not found.

str.isalnum()
Return true if all characters in the string are alphanumeric and there is at least one character, false otherwise. A character c is alphanumeric if one of the following returns True: c.isalpha(), c.isdecimal(), c.isdigit(), or c.isnumeric().

str.isalpha()
Return true if all characters in the string are alphabetic and there is at least one character, false otherwise. Alphabetic characters are those characters defined in the Unicode character database as “Letter”, i.e., those with general category property being one of “Lm”, “Lt”, “Lu”, “Ll”, or “Lo”. Note that this is different from the “Alphabetic” property defined in the Unicode Standard.

str.isdecimal()
Return true if all characters in the string are decimal characters and there is at least one character, false otherwise. Decimal characters are those from general category “Nd”. This category includes digit characters, and all characters that can be used to form decimal-radix numbers, e.g. U+0660, ARABIC-INDIC DIGIT ZERO.

str.isdigit()
Return true if all characters in the string are digits and there is at least one character, false otherwise. Digits include decimal characters and digits that need special handling, such as the compatibility superscript digits. Formally, a digit is a character that has the property value Numeric_Type=Digit or Numeric_Type=Decimal.

str.isidentifier()
Return true if the string is a valid identifier according to the language definition, section identifiers. Use keyword.iskeyword() to test for reserved identifiers such as def and class.

str.islower()
Return true if all cased characters 3 in the string are lowercase and there is at least one cased character, false otherwise.

str.isnumeric()
Return true if all characters in the string are numeric characters, and there is at least one character, false otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH. Formally, numeric characters are those with the property value Numeric_Type=Digit, Numeric_Type=Decimal or Numeric_Type=Numeric.

str.isprintable()
Return true if all characters in the string are printable or the string is empty, false otherwise. Nonprintable characters are those characters defined in the Unicode character database as “Other” or “Separator”, excepting the ASCII space (0x20) which is considered printable. (Note that printable characters in this context are those which should not be escaped when repr() is invoked on a string. It has no bearing on the handling of strings written to sys.stdout or sys.stderr.)

str.isspace()
Return true if there are only whitespace characters in the string and there is at least one character, false otherwise. Whitespace characters are those characters defined in the Unicode character database as “Other” or “Separator” and those with bidirectional property being one of “WS”, “B”, or “S”.  

3 Cased characters are those with general category property being one of “Lu” (Letter, uppercase), “Ll” (Letter, lowercase), or “Lt” (Letter, titlecase).
str. `istitle()`
Return true if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return false otherwise.

str. `isupper()`
Return true if all cased characters in the string are uppercase and there is at least one cased character, false otherwise.

str. `join(iterable)`
Return a string which is the concatenation of the strings in the iterable iterable. A TypeError will be raised if there are any non-string values in iterable, including bytes objects. The separator between elements is the string providing this method.

str. `ljust(width[, fillchar])`
Return the string left justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than or equal to len(s).

str. `lower()`
Return a copy of the string with all the cased characters converted to lowercase.
The lowercasing algorithm used is described in section 3.13 of the Unicode Standard.

str. `lstrip([chars])`
Return a copy of the string with leading characters removed. The chars argument is a string specifying the set of characters to be removed. If omitted or None, the chars argument defaults to removing whitespace. The chars argument is not a prefix; rather, all combinations of its values are stripped:

```python
>>> ' spacious '.lstrip()
'spacious '
>>> 'www.example.com'.lstrip('cmowz.')
'exmaple.com'
```

static str. `maketrans(x[, y[, z]])`
This static method returns a translation table usable for str.translate().
If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters (strings of length 1) to Unicode ordinals, strings (of arbitrary lengths) or None. Character keys will then be converted to ordinals.
If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

str. `partition(sep)`
Split the string at the first occurrence of sep, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

str. `replace(old, new[, count])`
Return a copy of the string with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

str. `rfind(sub[, start[, end]])`
Return the highest index in the string where substring sub is found, such that sub is contained within s[start:end]. Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

str. `rindex(sub[, start[, end]])`
Like rfind() but raises ValueError when the substring sub is not found.

str. `rjust(width[, fillchar])`
Return the string right justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than or equal to len(s).
str.\texttt{rpartition}(sep)
Split the string at the last occurrence of \textit{sep}, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

str.\texttt{rsplit}(sep=None, maxsplit=-1)
Return a list of the words in the string, using \textit{sep} as the delimiter string. If \textit{maxsplit} is given, at most \textit{maxsplit} splits are done, the rightmost ones. If \textit{sep} is not specified or None, any whitespace string is a separator. Except for splitting from the right, \texttt{rsplit()} behaves like \texttt{split()} which is described in detail below.

str.\texttt{rstrip}([\textit{chars}])
Return a copy of the string with trailing characters removed. The \textit{chars} argument is a string specifying the set of characters to be removed. If omitted or None, the \textit{chars} argument defaults to removing whitespace. The \textit{chars} argument is not a suffix; rather, all combinations of its values are stripped:

\begin{verbatim}
>>> ' spacious '.rstrip()
  'spacious'
>>> 'mississippi'.rstrip('ipz')
  'mississ'
\end{verbatim}

str.\texttt{split}(sep=None, maxsplit=-1)
Return a list of the words in the string, using \textit{sep} as the delimiter string. If \textit{maxsplit} is given, at most \textit{maxsplit} splits are done (thus, the list will have at most \textit{maxsplit}+1 elements). If \textit{maxsplit} is not specified or -1, then there is no limit on the number of splits (all possible splits are made).

If \textit{sep} is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, \texttt{1,2}.split(\texttt{,}) returns \texttt{[‘1’, ‘’, ‘2’]}). The \textit{sep} argument may consist of multiple characters (for example, \texttt{1<2<3}.split(\texttt{<>)}) returns \texttt{[‘1’, ‘2’, ‘3’]}. Splitting an empty string with a specified separator returns \texttt{[”]}. If \textit{sep} is not specified or is None, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a None separator returns \texttt{[]}.

For example, \texttt{ 1 2 3 }.split() returns \texttt{[‘1’, ‘2’, ‘3’]}, and \texttt{ 1 2 3 }.split(None, 1) returns \texttt{[‘1’, ‘2 3 ’]}.

str.\texttt{splitlines}([\textit{keepends}])
Return a list of the lines in the string, breaking at line boundaries. This method uses the universal newlines approach to splitting lines. Line breaks are not included in the resulting list unless \textit{keepends} is given and true.

For example, \texttt{‘ab c\n\nde fg\rkl\r\n’}.splitlines() returns \texttt{[‘ab c’, ‘’, ‘de fg’, ‘kl’]}, while the same call with \texttt{splitlines(True)} returns \texttt{[‘ab c\n’, ‘\n’, ‘de fg\r’, ‘\rk\n’]}.

Unlike \texttt{split()} when a delimiter string \textit{sep} is given, this method returns an empty list for the empty string, and a terminal line break does not result in an extra line.

str.\texttt{startswith}(prefix[, start, end])
Return True if string starts with the \textit{prefix}, otherwise return False. \textit{prefix} can also be a tuple of prefixes to look for. With optional \textit{start}, test string beginning at that position. With optional \textit{end}, stop comparing string at that position.

str.\texttt{strip}([\textit{chars}])
Return a copy of the string with the leading and trailing characters removed. The \textit{chars} argument is a string specifying the set of characters to be removed. If omitted or None, the \textit{chars} argument defaults to removing whitespace. The \textit{chars} argument is not a prefix or suffix; rather, all combinations of its values are stripped:

\begin{verbatim}
>>> ' spacious '.strip()
  'spacious'
\end{verbatim}
>>> 'www.example.com'.strip('cmowz.'),
'example'

str.swapcase()
Return a copy of the string with uppercase characters converted to lowercase and vice versa. Note that it is
not necessarily true that s.swapcase().swapcase() == s.

str.title()
Return a titlecased version of the string where words start with an uppercase character and the remaining
characters are lowercase.

The algorithm uses a simple language-independent definition of a word as groups of consecutive letters.
The definition works in many contexts but it means that apostrophes in contractions and possessives form
word boundaries, which may not be the desired result:

>>> "they’re bill’s friends from the UK".title()
"They’Re Bill’S Friends From The Uk"

A workaround for apostrophes can be constructed using regular expressions:

>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+'(\[A-Za-z]+)\?",
...                   lambda mo: mo.group(0)[0].upper() +
...                             mo.group(0)[1:].lower(),
...                     s)
... >>> titlecase("they’re bill’s friends.")
"They’re Bill’S Friends."

str.translate(map)
Return a copy of the s where all characters have been mapped through the map which must be a dictionary of
Unicode ordinals (integers) to Unicode ordinals, strings or None. Unmapped characters are left untouched.
Characters mapped to None are deleted.

You can use str.maketrans() to create a translation map from character-to-character mappings in
different formats.

Note: An even more flexible approach is to create a custom character mapping codec using the codecs
module (see encodings.cp1251 for an example).

str.upper()
Return a copy of the string with all the cased characters 4 converted to uppercase. Note that
str.upper().isupper() might be False if s contains uncased characters or if the Unicode cat-
egory of the resulting character(s) is not “Lu” (Letter, uppercase), but e.g. “Lt” (Letter, titlecase).

The uppercasing algorithm used is described in section 3.13 of the Unicode Standard.

str.zfill(width)
Return the numeric string left filled with zeros in a string of length width. A sign prefix is handled correctly.
The original string is returned if width is less than or equal to len(s).

4.7.2 printf-style String Formatting

Note: The formatting operations described here exhibit a variety of quirks that lead to a number of common errors
(such as failing to display tuples and dictionaries correctly). Using the newer str.format() interface helps
avoid these errors, and also provides a generally more powerful, flexible and extensible approach to formatting
text.
String objects have one unique built-in operation: the % operator (modulo). This is also known as the string formatting or interpolation operator. Given format % values (where format is a string), % conversion specifications in format are replaced with zero or more elements of values. The effect is similar to using the sprintf() in the C language.

If format requires a single argument, values may be a single non-tuple object. Otherwise, values must be a tuple with exactly the number of items specified by the format string, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The ‘%’ character, which marks the start of the specifier.
2. Mapping key (optional), consisting of a parenthesised sequence of characters (for example, (somename)).
3. Conversion flags (optional), which affect the result of some conversion types.
4. Minimum field width (optional). If specified as an ‘*’ (asterisk), the actual width is read from the next element of the tuple in values, and the object to convert comes after the minimum field width and optional precision.
5. Precision (optional), given as a ‘.’ (dot) followed by the precision. If specified as ‘*’ (an asterisk), the actual precision is read from the next element of the tuple in values, and the value to convert comes after the precision.
7. Conversion type.

When the right argument is a dictionary (or other mapping type), then the formats in the string must include a parenthesised mapping key into that dictionary inserted immediately after the ‘%’ character. The mapping key selects the value to be formatted from the mapping. For example:

```python
>>> print('%(language)s has %(number)03d quote types.' %
...     {'language': "Python", "number": 2})
Python has 002 quote types.
```

In this case no * specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘#’</td>
<td>The value conversion will use the “alternate form” (where defined below).</td>
</tr>
<tr>
<td>‘0’</td>
<td>The conversion will be zero padded for numeric values.</td>
</tr>
<tr>
<td>‘-’</td>
<td>The converted value is left adjusted (overrides the ‘0’ conversion if both are given).</td>
</tr>
<tr>
<td>‘ ’</td>
<td>(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.</td>
</tr>
<tr>
<td>‘+’</td>
<td>A sign character (‘+’ or ‘-’) will precede the conversion (overrides a “space” flag).</td>
</tr>
<tr>
<td>h, l, or L</td>
<td>A length modifier may be present, but is ignored as it is not necessary for Python – so e.g. %ld is identical to %d.</td>
</tr>
</tbody>
</table>

A length modifier (h, l, or L) may be present, but is ignored as it is not necessary for Python – so e.g. %ld is identical to %d.

The conversion types are:

---

4 To format only a tuple you should therefore provide a singleton tuple whose only element is the tuple to be formatted.
<table>
<thead>
<tr>
<th>Conversion</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>'d'</td>
<td>Signed integer decimal.</td>
<td></td>
</tr>
<tr>
<td>'i'</td>
<td>Signed integer decimal.</td>
<td></td>
</tr>
<tr>
<td>'o'</td>
<td>Signed octal value.</td>
<td></td>
</tr>
<tr>
<td>'u'</td>
<td>Obsolete type – it is identical to 'd'.</td>
<td>(1)</td>
</tr>
<tr>
<td>'x'</td>
<td>Signed hexadecimal (lowercase).</td>
<td>(2)</td>
</tr>
<tr>
<td>'X'</td>
<td>Signed hexadecimal (uppercase).</td>
<td>(2)</td>
</tr>
<tr>
<td>'e'</td>
<td>Floating point exponential format (lowercase).</td>
<td>(3)</td>
</tr>
<tr>
<td>'E'</td>
<td>Floating point exponential format (uppercase).</td>
<td>(3)</td>
</tr>
<tr>
<td>'f'</td>
<td>Floating point decimal format.</td>
<td>(3)</td>
</tr>
<tr>
<td>'F'</td>
<td>Floating point decimal format.</td>
<td>(3)</td>
</tr>
<tr>
<td>'g'</td>
<td>Floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.</td>
<td>(4)</td>
</tr>
<tr>
<td>'G'</td>
<td>Floating point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.</td>
<td>(4)</td>
</tr>
<tr>
<td>'c'</td>
<td>Single character (accepts integer or single character string).</td>
<td></td>
</tr>
<tr>
<td>'r'</td>
<td>String (converts any Python object using repr()).</td>
<td></td>
</tr>
<tr>
<td>'s'</td>
<td>String (converts any Python object using str()).</td>
<td></td>
</tr>
<tr>
<td>'a'</td>
<td>String (converts any Python object using ascii()).</td>
<td></td>
</tr>
<tr>
<td>'%'</td>
<td>No argument is converted, results in a '%' character in the result.</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. The alternate form causes a leading zero ('0') to be inserted between left-hand padding and the formatting of the number if the leading character of the result is not already a zero.

2. The alternate form causes a leading '0x' or '0X' (depending on whether the 'x' or 'X' format was used) to be inserted between left-hand padding and the formatting of the number if the leading character of the result is not already a zero.

3. The alternate form causes the result to always contain a decimal point, even if no digits follow it.

   The precision determines the number of digits after the decimal point and defaults to 6.

4. The alternate form causes the result to always contain a decimal point, and trailing zeroes are not removed as they would otherwise be.

   The precision determines the number of significant digits before and after the decimal point and defaults to 6.

5. If precision is N, the output is truncated to N characters.

7. See PEP 237.

Since Python strings have an explicit length, %s conversions do not assume that '\0' is the end of the string. Changed in version 3.1: %f conversions for numbers whose absolute value is over 1e50 are no longer replaced by %g conversions.

### 4.8 Binary Sequence Types — bytes, bytearray, memoryview

The core built-in types for manipulating binary data are bytes and bytearray. They are supported by memoryview which uses the buffer protocol to access the memory of other binary objects without needing to make a copy.

The array module supports efficient storage of basic data types like 32-bit integers and IEEE754 double-precision floating values.
4.8.1 Bytes

Bytes objects are immutable sequences of single bytes. Since many major binary protocols are based on the ASCII text encoding, bytes objects offer several methods that are only valid when working with ASCII compatible data and are closely related to string objects in a variety of other ways.

Firstly, the syntax for bytes literals is largely the same as that for string literals, except that a b prefix is added:

- Single quotes: `b'still allows embedded "double" quotes'`
- Double quotes: `b"still allows embedded 'single' quotes"`
- Triple quoted: `b'"3 single quotes"', b"""3 double quotes""

Only ASCII characters are permitted in bytes literals (regardless of the declared source code encoding). Any binary values over 127 must be entered into bytes literals using the appropriate escape sequence.

As with string literals, bytes literals may also use a r prefix to disable processing of escape sequences. See strings for more about the various forms of bytes literal, including supported escape sequences.

While bytes literals and representations are based on ASCII text, bytes objects actually behave like immutable sequences of integers, with each value in the sequence restricted such that $0 \leq x < 256$ (attempts to violate this restriction will trigger `ValueError`. This is done deliberately to emphasise that while many binary formats include ASCII based elements and can be usefully manipulated with some text-oriented algorithms, this is not generally the case for arbitrary binary data (blindly applying text processing algorithms to binary data formats that are not ASCII compatible will usually lead to data corruption).

In addition to the literal forms, bytes objects can be created in a number of other ways:

- A zero-filled bytes object of a specified length: `bytes(10)`
- From an iterable of integers: `bytes(range(20))`
- Copying existing binary data via the buffer protocol: `bytes(obj)`

Also see the `bytes` built-in.

Since bytes objects are sequences of integers, for a bytes object $b$, $b[0]$ will be an integer, while $b[0:1]$ will be a bytes object of length 1. (This contrasts with text strings, where both indexing and slicing will produce a string of length 1)

The representation of bytes objects uses the literal format (`b'...'`) since it is often more useful than e.g. `bytes([46, 46, 46])`. You can always convert a bytes object into a list of integers using `list(b)`.

**Note:** For Python 2.x users: In the Python 2.x series, a variety of implicit conversions between 8-bit strings (the closest thing 2.x offers to a built-in binary data type) and Unicode strings were permitted. This was a backwards compatibility workaround to account for the fact that Python originally only supported 8-bit text, and Unicode text was a later addition. In Python 3.x, those implicit conversions are gone - conversions between 8-bit binary data and Unicode text must be explicit, and bytes and string objects will always compare unequal.

4.8.2 Bytearray Objects

`bytearray` objects are a mutable counterpart to `bytes` objects. There is no dedicated literal syntax for bytearray objects, instead they are always created by calling the constructor:

- Creating an empty instance: `bytearray()`
- Creating a zero-filled instance with a given length: `bytearray(10)`
- From an iterable of integers: `bytearray(range(20))`
- Copying existing binary data via the buffer protocol: `bytearray(b'Hi!')`

As bytearray objects are mutable, they support the mutable sequence operations in addition to the common bytes and bytearray operations described in *Bytes and Bytearray Operations.*
Also see the `bytearray` built-in.

### 4.8.3 Bytes and Bytearray Operations

Both bytes and bytearray objects support the *common* sequence operations. They interoperate not just with operands of the same type, but with any object that supports the *buffer protocol*. Due to this flexibility, they can be freely mixed in operations without causing errors. However, the return type of the result may depend on the order of operands.

Due to the common use of ASCII text as the basis for binary protocols, bytes and bytearray objects provide almost all methods found on text strings, with the exceptions of:

- `str.encode()` (which converts text strings to bytes objects)
- `str.format()` and `str.format_map()` (which are used to format text for display to users)
- `str.isidentifier()`, `str.isnumeric()`, `str.isdecimal()`, `str.isprintable()` (which are used to check various properties of text strings which are not typically applicable to binary protocols).

All other string methods are supported, although sometimes with slight differences in functionality and semantics (as described below).

**Note:** The methods on bytes and bytearray objects don’t accept strings as their arguments, just as the methods on strings don’t accept bytes as their arguments. For example, you have to write:

```python
a = "abc"
b = a.replace("a", "f")
```

and:

```python
a = b"abc"
b = a.replace(b"a", b"f")
```

Whenever a bytes or bytearray method needs to interpret the bytes as characters (e.g. the `is...()` methods, `split()`, `strip()`), the ASCII character set is assumed (text strings use Unicode semantics).

**Note:** Using these ASCII based methods to manipulate binary data that is not stored in an ASCII based format may lead to data corruption.

The search operations (`in`, `count()`, `find()`, `index()`, `rfind()` and `rindex()`) all accept both integers in the range 0 to 255 (inclusive) as well as bytes and byte array sequences. Changed in version 3.3: All of the search methods also accept an integer in the range 0 to 255 (inclusive) as their first argument. Each bytes and bytearray instance provides a `decode()` convenience method that is the inverse of `str.encode()`:

```python
bytes.decode(encoding="utf-8", errors="strict")
bytearray.decode(encoding="utf-8", errors="strict")
```

Return a string decoded from the given bytes. Default encoding is `utf-8`. `errors` may be given to set a different error handling scheme. The default for `errors` is `strict`, meaning that encoding errors raise a `UnicodeError`. Other possible values are `ignore`, `replace` and any other name registered via `codecs.register_error()`, see section Codec Base Classes. For a list of possible encodings, see section Standard Encodings. Changed in version 3.1: Added support for keyword arguments.

Since 2 hexadecimal digits correspond precisely to a single byte, hexadecimal numbers are a commonly used format for describing binary data. Accordingly, the bytes and bytearray types have an additional class method to read data in that format:

```python
classmethod bytes.fromhex(string)
classmethod bytearray.fromhex(string)
```

This `bytes` class method returns a bytes or bytearray object, decoding the given string object. The string must contain two hexadecimal digits per byte, spaces are ignored.
The maketrans and translate methods differ in semantics from the versions available on strings:

\[
\begin{align*}
\text{bytes.translate} & \left( \text{table[, delete]} \right) \\
\text{bytearray.translate} & \left( \text{table[, delete]} \right)
\end{align*}
\]

Return a copy of the bytes or bytearray object where all bytes occurring in the optional argument \text{delete} are removed, and the remaining bytes have been mapped through the given translation table, which must be a bytes object of length 256.

You can use the \text{bytes.maketrans()} method to create a translation table.

Set the \text{table} argument to \text{None} for translations that only delete characters:

\[
\begin{align*}
\texttt{b’read this short text’}.\text{translate}(\text{None, b’aeiou’}) \\
\texttt{b’rd ths shrt txt’}
\end{align*}
\]

\[
\begin{align*}
\text{static bytes.maketrans} & \left( \text{from, to} \right) \\
\text{static bytearray.maketrans} & \left( \text{from, to} \right)
\end{align*}
\]

This static method returns a translation table usable for \text{bytes.translate()} that will map each character in \text{from} into the character at the same position in \text{to}; \text{from} and \text{to} must be bytes objects and have the same length. New in version 3.1.

## 4.8.4 Memory Views

\text{memoryview} objects allow Python code to access the internal data of an object that supports the buffer protocol without copying.

\text{class memoryview(obj)}

Create a \text{memoryview} that references \text{obj}. \text{obj} must support the buffer protocol. Built-in objects that support the buffer protocol include \text{bytes} and \text{bytearray}.

A \text{memoryview} has the notion of an \text{element}, which is the atomic memory unit handled by the originating object \text{obj}. For many simple types such as \text{bytes} and \text{bytearray}, an element is a single byte, but other types such as \text{array.array} may have bigger elements.

\text{len(view)} is equal to the length of \text{tolist}. If \text{view.ndim} = 0, the length is 1. If \text{view.ndim} = 1, the length is equal to the number of elements in the view. For higher dimensions, the length is equal to the length of the nested list representation of the view. The \text{itemsize} attribute will give you the number of bytes of a single element.

A \text{memoryview} supports slicing to expose its data. If \text{format} is one of the native format specifiers from the \text{struct} module, indexing will return a single element with the correct type. Full slicing will result in a subview:

\[
\begin{align*}
\texttt{v = memoryview(b’abcefg’)} \\
\texttt{v[1]} \quad \text{98} \\
\texttt{v[-1]} \quad \text{103} \\
\texttt{v[1:4]} \quad \text{<memory at 0x7f3ddc9f4350>} \\
\texttt{bytes(v[1:4])} \\
\texttt{b’bce’}
\end{align*}
\]

Other native formats:

\[
\begin{align*}
\texttt{import array} \\
\texttt{a = array.array(’l’, [-11111111, 22222222, -33333333, 44444444])} \\
\texttt{a[0]}
\end{align*}
\]
The Python Library Reference, Release 3.3.3

-11111111
>>> a[-1]
44444444
>>> a[2:3].tolist()
[-33333333]
>>> a[::2].tolist()
[-11111111, -33333333]
>>> a[::-1].tolist()
[44444444, -33333333, 22222222, -11111111]
New in version 3.3. If the underlying object is writable, the memoryview supports slice assignment. Resizing is not allowed:
>>> data = bytearray(b’abcefg’)
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b’z’)
>>> data
bytearray(b’zbcefg’)
>>> v[1:4] = b’123’
>>> data
bytearray(b’z123fg’)
>>> v[2:3] = b’spam’
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rvalue have different structures
>>> v[2:6] = b’spam’
>>> data
bytearray(b’z1spam’)
One-dimensional memoryviews of hashable (read-only) types with formats ‘B’, ‘b’ or ‘c’ are also hashable.
The hash is defined as hash(m) == hash(m.tobytes()):
>>> v = memoryview(b’abcefg’)
>>> hash(v) == hash(b’abcefg’)
True
>>> hash(v[2:4]) == hash(b’ce’)
True
>>> hash(v[::-2]) == hash(b’abcefg’[::-2])
True
Changed in version 3.3: One-dimensional memoryviews with formats ‘B’, ‘b’ or ‘c’ are now hashable.
memoryview has several methods:
__eq__(exporter)
A memoryview and a PEP 3118 exporter are equal if their shapes are equivalent and if all corresponding values are equal when the operands’ respective format codes are interpreted using struct
syntax.
For the subset of struct format strings currently supported by tolist(), v and w are equal if
v.tolist() == w.tolist():
>>>
>>>
>>>
>>>
>>>
>>>

import array
a = array.array(’I’, [1, 2, 3, 4, 5])
b = array.array(’d’, [1.0, 2.0, 3.0, 4.0, 5.0])
c = array.array(’b’, [5, 3, 1])
x = memoryview(a)
y = memoryview(b)

4.8. Binary Sequence Types — bytes, bytearray, memoryview

49


>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[::-2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True

If either format string is not supported by the struct module, then the objects will always compare as unequal (even if the format strings and buffer contents are identical):

```python
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [('x', c_long), ('y', c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

Note that, as with floating point numbers, v is w does not imply v == w for memoryview objects. Changed in version 3.3: Previous versions compared the raw memory disregarding the item format and the logical array structure.

tobytes()
Return the data in the buffer as a bytestring. This is equivalent to calling the bytes constructor on the memoryview.

```python
>>> m = memoryview(b"abc")
>>> m.tobytes()
b'abc'
>>> bytes(m)
b'abc'
```

For non-contiguous arrays the result is equal to the flattened list representation with all elements converted to bytes. tobytes() supports all format strings, including those that are not in struct module syntax.

tolist()
Return the data in the buffer as a list of elements.

```python
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]
```

Changed in version 3.3: tolist() now supports all single character native formats in struct module syntax as well as multi-dimensional representations.

release()
Release the underlying buffer exposed by the memoryview object. Many objects take special actions
when a view is held on them (for example, a `bytearray` would temporarily forbid resizing); therefore, calling `release()` is handy to remove these restrictions (and free any dangling resources) as soon as possible.

After this method has been called, any further operation on the view raises a `ValueError` (except `release()` itself which can be called multiple times):

```python
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

The context management protocol can be used for a similar effect, using the `with` statement:

```python
>>> with memoryview(b'abc') as m:
...   m[0]
... 97
```

```python
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

New in version 3.2.

`cast(format[, shape])`

Cast a memoryview to a new format or shape. `shape` defaults to `[byte_length//new_itemsize]`, which means that the result view will be one-dimensional. The return value is a new memoryview, but the buffer itself is not copied. Supported casts are 1D -> C-contiguous and C-contiguous -> 1D.

Both formats are restricted to single element native formats in `struct` syntax. One of the formats must be a byte format (`'B'`, `'b'` or `'c'`). The byte length of the result must be the same as the original length.

Cast 1D/long to 1D/unsigned bytes:

```python
>>> import array
>>> a = array.array('l', [1,2,3])
>>> x = memoryview(a)
>>> x.format
'l'
>>> x.itemsize
8
>>> len(x)
3
>>> x.nbytes
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24
```
Cast 1D/unsigned bytes to 1D/char:

```python
>>> b = bytearray(b'zyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview: invalid value for format "B"
>>> y = x.cast('c')
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')
```

Cast 1D/bytes to 3D/int to 1D/signed char:

```python
>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
>>> y.format
'i'
>>> y.itemsize
4
>>> len(y)
2
>>> y.nbytes
48
>>> z = y.cast('b')
>>> z.format
'b'
>>> z.itemsize
1
>>> len(z)
48
>>> z.nbytes
48
```

Cast 1D/unsigned char to 2D/unsigned long:

```python
>>> buf = struct.pack("L"*6, *list(range(6)))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)
2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]
```

New in version 3.3.

There are also several readonly attributes available:

- **obj**
  - The underlying object of the memoryview:
>>> b = bytearray(b'xyz')
>>> m = memoryview(b)
>>> m.obj is b
True

New in version 3.3.

**nbytes**

nbytes == product(shape) * itemsize == len(m.tobytes()). This is the amount of space in bytes that the array would use in a contiguous representation. It is not necessarily equal to len(m):

>>> import array
>>> a = array.array('i', [1, 2, 3, 4, 5])
>>> m = memoryview(a)
>>> len(m)
5
>>> m.nbytes
20
>>> y = m[:, :2]
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12

Multi-dimensional arrays:

>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in range(12)])
>>> x = memoryview(buf)
>>> y = x.cast('d', shape=[3, 4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5, 15.0, 16.5]]
>>> len(y)
3
>>> y.nbytes
96

New in version 3.3.

**readonly**

A bool indicating whether the memory is read only.

**format**

A string containing the format (in struct module style) for each element in the view. A memoryview can be created from exporters with arbitrary format strings, but some methods (e.g. tolist()) are restricted to native single element formats. Changed in version 3.3: format ‘B’ is now handled according to the struct module syntax. This means that memoryview(b’abc’) [0] == b’abc’ [0] == 97.

**itemsize**

The size in bytes of each element of the memoryview:

>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001, 32002]))
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True

ndim
An integer indicating how many dimensions of a multi-dimensional array the memory represents.

shape
A tuple of integers the length of ndim giving the shape of the memory as an N-dimensional array.
Changed in version 3.3: An empty tuple instead of None when ndim = 0.

strides
A tuple of integers the length of ndim giving the size in bytes to access each element for each dimension of the array.
Changed in version 3.3: An empty tuple instead of None when ndim = 0.

suboffsets
Used internally for PIL-style arrays. The value is informational only.

c_contiguous
A bool indicating whether the memory is C-contiguous. New in version 3.3.

f_contiguous
A bool indicating whether the memory is Fortran contiguous. New in version 3.3.

contiguous
A bool indicating whether the memory is contiguous. New in version 3.3.

4.9 Set Types — set, frozenset

A set object is an unordered collection of distinct hashable objects. Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference. (For other containers see the built-in dict, list, and tuple classes, and the collections module.)

Like other collections, sets support x in set, len(set), and for x in set. Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

There are currently two built-in set types, set and frozenset. The set type is mutable — the contents can be changed using methods like add() and remove(). Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set. The frozenset type is immutable and hashable — its contents cannot be altered after it is created; it can therefore be used as a dictionary key or as an element of another set.

Non-empty sets (not frozensets) can be created by placing a comma-separated list of elements within braces, for example: {'jack', 'sjoerd'}, in addition to the set constructor.

The constructors for both classes work the same:

class set([Iterable])
class frozenset([Iterable])

Return a new set or frozenset object whose elements are taken from iterable. The elements of a set must be hashable. To represent sets of sets, the inner sets must be frozenset objects. If iterable is not specified, a new empty set is returned.

Instances of set and frozenset provide the following operations:

len(s)
Return the cardinality of set s.

x in s
Test x for membership in s.
x not in s
Test x for non-membership in s.

isdisjoint(other)
Return True if the set has no elements in common with other. Sets are disjoint if and only if their intersection is the empty set.

issubset(other)
set <= other
Test whether every element in the set is in other.

set < other
Test whether the set is a proper subset of other, that is, set <= other and set != other.

issuperset(other)
set >= other
Test whether every element in other is in the set.

set > other
Test whether the set is a proper superset of other, that is, set >= other and set != other.

union(other, ...)
set | other | ...
Return a new set with elements from the set and all others.

intersection(other, ...)
set & other & ...
Return a new set with elements common to the set and all others.

difference(other, ...)
set - other - ...
Return a new set with elements in the set that are not in the others.

symmetric_difference(other)
set ^ other
Return a new set with elements in either the set or other but not both.

copy()
Return a new set with a shallow copy of s.

Note, the non-operator versions of union(), intersection(), difference(), and symmetric_difference(), issubset(), and issuperset() methods will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be sets. This precludes error-prone constructions like set(‘abc’) & ‘cbs’ in favor of the more readable set(‘abc’).intersection(‘cbs’).

Both set and frozenset support set to set comparisons. Two sets are equal if and only if every element of each set is contained in the other (each is a subset of the other). A set is less than another set if and only if the first set is a proper subset of the second set (is a subset, but is not equal). A set is greater than another set if and only if the first set is a proper superset of the second set (is a superset, but is not equal).

Instances of set are compared to instances of frozenset based on their members. For example, set(‘abc’) == frozenset(‘abc’) returns True and so does set(‘abc’) in set([frozenset(‘abc’)]).

The subset and equality comparisons do not generalize to a total ordering function. For example, any two nonempty disjoint sets are not equal and are not subsets of each other, so all of the following return False: a<b, a==b, or a>b.

Since sets only define partial ordering (subset relationships), the output of the list.sort() method is undefined for lists of sets.

Set elements, like dictionary keys, must be hashable.

Binary operations that mix set instances with frozenset return the type of the first operand. For example: frozenset(‘ab’) | set(‘bc’) returns an instance of frozenset.
The following table lists operations available for set that do not apply to immutable instances of frozenset:

- `update(other, ...)`: Update the set, adding elements from all others.
- `set |= other | ...`: Update the set, keeping only elements found in it and all others.
- `intersection_update(other, ...)`: Update the set, keeping only elements found in either set, but not in both.
- `set &= other & ...`: Update the set, removing elements found in others.
- `difference_update(other, ...)`: Update the set, removing elements found in others.
- `symmetric_difference_update(other)`: Update the set, keeping only elements found in either set, but not in both.
- `add(elem)`: Add element `elem` to the set.
- `remove(elem)`: Remove element `elem` from the set. Raises `KeyError` if `elem` is not contained in the set.
- `discard(elem)`: Remove element `elem` from the set if it is present.
- `pop()` : Remove and return an arbitrary element from the set. Raises `KeyError` if the set is empty.
- `clear()` : Remove all elements from the set.

Note, the non-operator versions of the `update()`, `intersection_update()`, `difference_update()`, and `symmetric_difference_update()` methods will accept any iterable as an argument.

Note, the `elem` argument to the `__contains__()`, `remove()`, and `discard()` methods may be a set. To support searching for an equivalent frozenset, the `elem` set is temporarily mutated during the search and then restored. During the search, the `elem` set should not be read or mutated since it does not have a meaningful value.

### 4.10 Mapping Types — `dict`

A `mapping` object maps `hashable` values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the `dictionary`. (For other containers see the built-in `list`, `set`, and `tuple` classes, and the `collections` module.)

A dictionary’s keys are *almost* arbitrary values. Values that are not `hashable`, that is, values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may not be used as keys. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (such as 1 and 1.0) then they can be used interchangeably to index the same dictionary entry. (Note however, that since computers store floating-point numbers as approximations it is usually unwise to use them as dictionary keys.)

Dictionaries can be created by placing a comma-separated list of key: value pairs within braces, for example: `{‘jack’: 4098, ‘sjoerd’: 4127} or `{4098: ‘jack’, 4127: ‘sjoerd’}`, or by the `dict` constructor.

- `class dict(**kwarg)`
- `class dict(mapping, **kwarg)`
class `dict`(iterable, **kwarg)

Return a new dictionary initialized from an optional positional argument and a possibly empty set of keyword arguments.

If no positional argument is given, an empty dictionary is created. If a positional argument is given and it is a mapping object, a dictionary is created with the same key-value pairs as the mapping object. Otherwise, the positional argument must be an `iterator` object. Each item in the iterable must itself be an iterator with exactly two objects. The first object of each item becomes a key in the new dictionary, and the second object the corresponding value. If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary.

If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument. If a key being added is already present, the value from the keyword argument replaces the value from the positional argument.

To illustrate, the following examples all return a dictionary equal to `{"one": 1, "two": 2, "three": 3}`:

```python
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(["one", "two", "three"], [1, 2, 3]))
>>> d = dict(("two", 2), ("one", 1), ("three", 3))
>>> e = dict(3, one: 1, two: 2))
>>> a == b == c == d == e
True
```

Providing keyword arguments as in the first example only works for keys that are valid Python identifiers. Otherwise, any valid keys can be used.

These are the operations that dictionaries support (and therefore, custom mapping types should support too):

- **len(d)**
  Return the number of items in the dictionary d.

- **d[key]**
  Return the item of d with key key. Raises a `KeyError` if key is not in the map.
  
  If a subclass of dict defines a method `__missing__`, if the key key is not present, the d[key] operation calls that method with the key key as argument. The d[key] operation then returns or raises whatever is returned or raised by the `__missing__(key)` call if the key is not present. No other operations or methods invoke `__missing__`. If `__missing__` is not defined, `KeyError` is raised. `__missing__` must be a method; it cannot be an instance variable:

```python
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
>>> c = Counter()
>>> c["red"]
0
>>> c["red"] += 1
>>> c["red"]
1
```

See `collections.Counter` for a complete implementation including other methods helpful for accumulating and managing tallies.

- **d[key] = value**
  Set d[key] to value.

- **del d[key]**
  Remove d[key] from d. Raises a `KeyError` if key is not in the map.
key in d
Return True if d has a key key, else False.

key not in d
Equivalent to not key in d.

iter(d)
Return an iterator over the keys of the dictionary. This is a shortcut for iter(d.keys()).

clear()
Remove all items from the dictionary.

copy()
Return a shallow copy of the dictionary.

classmethod fromkeys(seq[, value])
Create a new dictionary with keys from seq and values set to value.

fromkeys() is a class method that returns a new dictionary. value defaults to None.

get(key[, default])
Return the value for key if key is in the dictionary, else default. If default is not given, it defaults to None, so that this method never raises a KeyError.

items()
Return a new view of the dictionary’s items ((key, value) pairs). See the documentation of view objects.

keys()
Return a new view of the dictionary’s keys. See the documentation of view objects.

pop(key[, default])
If key is in the dictionary, remove it and return its value, else return default. If default is not given and key is not in the dictionary, a KeyError is raised.

popitem()
Remove and return an arbitrary (key, value) pair from the dictionary.

popitem() is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling popitem() raises a KeyError.

setdefault(key[, default])
If key is in the dictionary, return its value. If not, insert key with a value of default and return default. default defaults to None.

update([other])
Update the dictionary with the key/value pairs from other, overwriting existing keys. Return None.

update() accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: d.update(red=1, blue=2).

values()
Return a new view of the dictionary’s values. See the documentation of view objects.

See Also:
types.MappingProxyType can be used to create a read-only view of a dict.

4.10.1 Dictionary view objects

The objects returned by dict.keys(), dict.values() and dict.items() are view objects. They provide a dynamic view on the dictionary’s entries, which means that when the dictionary changes, the view reflects these changes.

Dictionary views can be iterated over to yield their respective data, and support membership tests:
len(dictview)
Return the number of entries in the dictionary.

iter(dictview)
Return an iterator over the keys, values or items (represented as tuples of `(key, value)`) in the dictionary.

Keys and values are iterated over in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary's history of insertions and deletions. If keys, values and items views are iterated over with no intervening modifications to the dictionary, the order of items will directly correspond. This allows the creation of `(value, key)` pairs using `zip()`: pairs = `zip(d.values(), d.keys())`. Another way to create the same list is pairs = `{(v, k) for (k, v) in d.items()}`.

Iterating views while adding or deleting entries in the dictionary may raise a `RuntimeError` or fail to iterate over all entries.

x in dictview
Return `True` if `x` is in the underlying dictionary’s keys, values or items (in the latter case, `x` should be a `(key, value)` tuple).

Keys views are set-like since their entries are unique and hashable. If all values are hashable, so that `(key, value)` pairs are unique and hashable, then the items view is also set-like. (Values views are not treated as set-like since the entries are generally not unique.) For set-like views, all of the operations defined for the abstract base class `collections.abc.Set` are available (for example, `==`, `<`, or `^`).

An example of dictionary view usage:

```python
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504

>>> # keys and values are iterated over in the same order
>>> list(keys)
['eggs', 'bacon', 'sausage', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['spam', 'bacon']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'}
{'juice', 'sausage', 'bacon', 'spam'}
```
### 4.11 Context Manager Types

Python's `with` statement supports the concept of a runtime context defined by a context manager. This is implemented using a pair of methods that allow user-defined classes to define a runtime context that is entered before the statement body is executed and exited when the statement ends:

**contextmanager.** `__enter__()`  
Enter the runtime context and return either this object or another object related to the runtime context. The value returned by this method is bound to the identifier in the `as` clause of `with` statements using this context manager.

An example of a context manager that returns itself is a file object. File objects return themselves from `__enter__()` to allow `open()` to be used as the context expression in a `with` statement.

An example of a context manager that returns a related object is the one returned by `decimal.localcontext()`. These managers set the active decimal context to a copy of the original decimal context and then return the copy. This allows changes to be made to the current decimal context in the body of the `with` statement without affecting code outside the `with` statement.

**contextmanager.** `__exit__`(exc_type, exc_val, exc_tb)  
Exit the runtime context and return a Boolean flag indicating if any exception that occurred should be suppressed. If an exception occurred while executing the body of the `with` statement, the arguments contain the exception type, value and traceback information. Otherwise, all three arguments are `None`.

Returning a true value from this method will cause the `with` statement to suppress the exception and continue execution with the statement immediately following the `with` statement. Otherwise the exception continues propagating after this method has finished executing. Exceptions that occur during execution of this method will replace any exception that occurred in the body of the `with` statement.

The exception passed in should never be reraised explicitly - instead, this method should return a false value to indicate that the method completed successfully and does not want to suppress the raised exception. This allows context management code (such as `contextlib.nested`) to easily detect whether or not an `__exit__()` method has actually failed.

Python defines several context managers to support easy thread synchronisation, prompt closure of files or other objects, and simpler manipulation of the active decimal arithmetic context. The specific types are not treated specially beyond their implementation of the context management protocol. See the `contextlib` module for some examples.

Python's `generators` and the `contextlib.contextmanager` decorator provide a convenient way to implement these protocols. If a generator function is decorated with the `contextlib.contextmanager` decorator, it will return a context manager implementing the necessary `__enter__()` and `__exit__()` methods, rather than the iterator produced by an undecorated generator function.

Note that there is no specific slot for any of these methods in the type structure for Python objects in the Python/C API. Extension types wanting to define these methods must provide them as a normal Python accessible method. Compared to the overhead of setting up the runtime context, the overhead of a single class dictionary lookup is negligible.

### 4.12 Other Built-in Types

The interpreter supports several other kinds of objects. Most of these support only one or two operations.

#### 4.12.1 Modules

The only special operation on a module is attribute access: `m.name`, where `m` is a module and `name` accesses a name defined in `m`'s symbol table. Module attributes can be assigned to. (Note that the `import` statement is not, strictly speaking, an operation on a module object; `import foo` does not require a module object named `foo` to exist, rather it requires an (external) definition for a module named `foo` somewhere.)
A special attribute of every module is \_dict\_. This is the dictionary containing the module’s symbol table. Modifying this dictionary will actually change the module’s symbol table, but direct assignment to the \_dict\_ attribute is not possible (you can write \texttt{m._dict_['a'] = 1}, which defines \texttt{m.a} to be 1, but you can’t write \texttt{m._dict_ = {}}). Modifying \_dict\_ directly is not recommended.

Modules built into the interpreter are written like this: \texttt{<module 'sys' (built-in)>}. If loaded from a file, they are written as \texttt{<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>}.

### 4.12.2 Classes and Class Instances

See \texttt{objects} and \texttt{class} for these.

### 4.12.3 Functions

Function objects are created by function definitions. The only operation on a function object is to call it: \texttt{func} (argument-list).

There are really two flavors of function objects: built-in functions and user-defined functions. Both support the same operation (to call the function), but the implementation is different, hence the different object types. See \texttt{function} for more information.

### 4.12.4 Methods

Methods are functions that are called using the attribute notation. There are two flavors: built-in methods (such as \texttt{append()} on lists) and class instance methods. Built-in methods are described with the types that support them.

If you access a method (a function defined in a class namespace) through an instance, you get a special object: a bound method (also called instance method) object. When called, it will add the self argument to the argument list. Bound methods have two special read-only attributes: \texttt{m._self__} is the object on which the method operates, and \texttt{m._func__} is the function implementing the method. Calling \texttt{m(arg-1, arg-2, ..., arg-n)} is completely equivalent to calling \texttt{m._func__(m._self__, arg-1, arg-2, ..., arg-n)}.

Like function objects, bound method objects support getting arbitrary attributes. However, since method attributes are actually stored on the underlying function object (\texttt{meth._func__}), setting method attributes on bound methods is disallowed. Attempting to set an attribute on a method results in an \texttt{AttributeError} being raised. In order to set a method attribute, you need to explicitly set it on the underlying function object:

```python
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method'  # can’t set on the method
AttributeError: ‘method’ object has no attribute ‘whoami’
```

In order to set a method attribute, you need to explicitly set it on the underlying function object:

```python
>>> c.method.__func__.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

See \texttt{types} for more information.

### 4.12.5 Code Objects

Code objects are used by the implementation to represent “pseudo-compiled” executable Python code such as a function body. They differ from function objects because they don’t contain a reference to their global execution
environment. Code objects are returned by the built-in `compile()` function and can be extracted from function objects through their `__code__` attribute. See also the `code` module.

A code object can be executed or evaluated by passing it (instead of a source string) to the `exec()` or `eval()` built-in functions.

See `types` for more information.

### 4.12.6 Type Objects

Type objects represent the various object types. An object’s type is accessed by the built-in function `type()`. There are no special operations on types. The standard module `types` defines names for all standard built-in types.

Types are written like this: `<class 'int'>`.

### 4.12.7 The Null Object

This object is returned by functions that don’t explicitly return a value. It supports no special operations. There is exactly one null object, named `None` (a built-in name). `type(None)` produces the same singleton.

It is written as `None`.

### 4.12.8 The Ellipsis Object

This object is commonly used by slicing (see `slicings`). It supports no special operations. There is exactly one ellipsis object, named `Ellipsis` (a built-in name). `type(Ellipsis)` produces the `Ellipsis` singleton.

It is written as `Ellipsis` or `...`.

### 4.12.9 The NotImplemented Object

This object is returned from comparisons and binary operations when they are asked to operate on types they don’t support. See `comparisons` for more information. There is exactly one `NotImplemented` object.

`type(NotImplemented)` produces the singleton instance.

It is written as `NotImplemented`.

### 4.12.10 Boolean Values

Boolean values are the two constant objects `False` and `True`. They are used to represent truth values (although other values can also be considered false or true). In numeric contexts (for example when used as the argument to an arithmetic operator), they behave like the integers 0 and 1, respectively. The built-in function `bool()` can be used to convert any value to a Boolean, if the value can be interpreted as a truth value (see section `Truth Value Testing` above).

They are written as `False` and `True`, respectively.

### 4.12.11 Internal Objects

See `types` for this information. It describes stack frame objects, traceback objects, and slice objects.
4.13 Special Attributes

The implementation adds a few special read-only attributes to several object types, where they are relevant. Some of these are not reported by the `dir()` built-in function.

```python
object.__dict__
   A dictionary or other mapping object used to store an object’s (writable) attributes.

instance.__class__
   The class to which a class instance belongs.

class.__bases__
   The tuple of base classes of a class object.

class.__name__
   The name of the class or type.

class.__qualname__
   The qualified name of the class or type. New in version 3.3.

class.__mro__
   This attribute is a tuple of classes that are considered when looking for base classes during method resolution.

class.mro()
   This method can be overridden by a metaclass to customize the method resolution order for its instances. It is called at class instantiation, and its result is stored in __mro__.

class.__subclasses__()
   Each class keeps a list of weak references to its immediate subclasses. This method returns a list of all those references still alive. Example:

   >>> int.__subclasses__()
   [<class 'bool'>]
```
In Python, all exceptions must be instances of a class that derives from `BaseException`. In a try statement with an except clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which it is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an “associated value” indicating the detailed cause of the error. This may be a string or a tuple of several items of information (e.g., an error code and a string explaining the code). The associated value is usually passed as arguments to the exception class’s constructor.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition “just like” the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The built-in exception classes can be sub-classed to define new exceptions; programmers are encouraged to at least derive new exceptions from the `Exception` class and not `BaseException`. More information on defining exceptions is available in the Python Tutorial under `tut-userexceptions`.

When raising (or re-raising) an exception in an except clause `__context__` is automatically set to the last exception caught; if the new exception is not handled the traceback that is eventually displayed will include the originating exception(s) and the final exception.

When raising a new exception (rather than using a bare `raise` to re-raise the exception currently being handled), the implicit exception context can be supplemented with an explicit cause by using `from with raise`:

```python
raise new_exc from original_exc
```

The expression following `from` must be an exception or `None`. It will be set as `__cause__` on the raised exception. Setting `__cause__` also implicitly sets the `__suppress_context__` attribute to `True`, so that using `raise new_exc from None` effectively replaces the old exception with the new one for display purposes (e.g., converting `KeyError` to `AttributeError`, while leaving the old exception available in `__context__` for introspection when debugging).

The default traceback display code shows these chained exceptions in addition to the traceback for the exception itself. An explicitly chained exception in `__cause__` is always shown when present. An implicitly chained exception in `__context__` is shown only if `__cause__` is `None` and `__suppress_context__` is false.

In either case, the exception itself is always shown after any chained exceptions so that the final line of the traceback always shows the last exception that was raised.

### 5.1 Base classes

The following exceptions are used mostly as base classes for other exceptions.

#### exception BaseException

The base class for all built-in exceptions. It is not meant to be directly inherited by user-defined classes (for that, use `Exception`). If `str()` is called on an instance of this class, the representation of the argument(s) to the instance are returned, or the empty string when there were no arguments.
The tuple of arguments given to the exception constructor. Some built-in exceptions (like \texttt{IOError}) expect a certain number of arguments and assign a special meaning to the elements of this tuple, while others are usually called only with a single string giving an error message.

\textbf{with\_traceback} \((tb)\)

This method sets \(tb\) as the new traceback for the exception and returns the exception object. It is usually used in exception handling code like this:

\begin{verbatim}
try:  
...  
except SomeException:  
  tb = sys.exc_info()[2]  
  raise OtherException(...).with_traceback(tb)
\end{verbatim}

\textbf{exception \texttt{Exception}}

All built-in, non-system-exiting exceptions are derived from this class. All user-defined exceptions should also be derived from this class.

\textbf{exception \texttt{ArithmeticError}}

The base class for those built-in exceptions that are raised for various arithmetic errors: \texttt{OverflowError}, \texttt{ZeroDivisionError}, \texttt{FloatingPointError}.

\textbf{exception \texttt{BufferError}}

Raised when a buffer related operation cannot be performed.

\textbf{exception \texttt{LookupError}}

The base class for the exceptions that are raised when a key or index used on a mapping or sequence is invalid: \texttt{IndexError}, \texttt{KeyError}. This can be raised directly by \texttt{codecs.lookup()}.  

5.2 Concrete exceptions

The following exceptions are the exceptions that are usually raised.

\textbf{exception \texttt{AssertionError}}

Raised when an \texttt{assert} statement fails.

\textbf{exception \texttt{AttributeError}}

Raised when an attribute reference (see \texttt{attribute-references}) or assignment fails. (When an object does not support attribute references or attribute assignments at all, \texttt{TypeError} is raised.)

\textbf{exception \texttt{EOFError}}

Raised when the \texttt{input()} function hits an end-of-file condition (EOF) without reading any data. (N.B.: the \texttt{io.IOBase.read()} and \texttt{io.IOBase.readline()} methods return an empty string when they hit EOF.)

\textbf{exception \texttt{FloatingPointError}}

Raised when a floating point operation fails. This exception is always defined, but can only be raised when Python is configured with the \texttt{--with-fpectl} option, or the \texttt{WANT_SIGFPE_HANDLER} symbol is defined in the \texttt{pyconfig.h} file.

\textbf{exception \texttt{GeneratorExit}}

Raise when a \texttt{generator}'s \texttt{close()} method is called. It directly inherits from \texttt{BaseException} instead of \texttt{Exception} since it is technically not an error.

\textbf{exception \texttt{ImportError}}

Raised when an \texttt{import} statement fails to find the module definition or when a \texttt{from ... import} fails to find a name that is to be imported.

The \texttt{name} and \texttt{path} attributes can be set using keyword-only arguments to the constructor. When set they represent the name of the module that was attempted to be imported and the path to any file which triggered the exception, respectively. Changed in version 3.3: Added the \texttt{name} and \texttt{path} attributes.

Chapter 5. Built-in Exceptions
exception **IndexError**

Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not an integer, **TypeError** is raised.)

exception **KeyError**

Raised when a mapping (dictionary) key is not found in the set of existing keys.

exception **KeyboardInterrupt**

Raised when the user hits the interrupt key (normally Control-C or Delete). During execution, a check for interrupts is made regularly. The exception inherits from **BaseException** so as to not be accidentally caught by code that catches **Exception** and thus prevent the interpreter from exiting.

exception **MemoryError**

Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated value is a string indicating what kind of (internal) operation ran out of memory. Note that because of the underlying memory management architecture (C’s **malloc()** function), the interpreter may not always be able to completely recover from this situation; it nevertheless raises an exception so that a stack traceback can be printed, in case a run-away program was the cause.

exception **NameError**

Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.

exception **NotImplementedError**

This exception is derived from **RuntimeError**. In user defined base classes, abstract methods should raise this exception when they require derived classes to override the method.

exception **OSError**

This exception is raised when a system function returns a system-related error, including I/O failures such as “file not found” or “disk full” (not for illegal argument types or other incidental errors). Often a subclass of **OSError** will actually be raised as described in OS exceptions below. The **errno** attribute is a numeric error code from the C variable **errno**.

Under Windows, the **winerror** attribute gives you the native Windows error code. The **errno** attribute is then an approximate translation, in POSIX terms, of that native error code.

Under all platforms, the **strerror** attribute is the corresponding error message as provided by the operating system (as formatted by the C functions **perror()** under POSIX, and **FormatMessage()** Windows).

For exceptions that involve a file system path (such as **open()** or **os.unlink()**), the exception instance will contain an additional attribute, **filename**, which is the file name passed to the function. Changed in version 3.3: **EnvironmentError**, **IOError**, **WindowsError**, **VMSError**, **socket.error**, **select.error** and **mmap.error** have been merged into **OSError**.

exception **OverflowError**

Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for integers (which would rather raise **MemoryError** than give up). Because of the lack of standardization of floating point exception handling in C, most floating point operations also aren’t checked.

exception **ReferenceError**

This exception is raised when a weak reference proxy, created by the **weakref.proxy()** function, is used to access an attribute of the referent after it has been garbage collected. For more information on weak references, see the **weakref** module.

exception **RuntimeError**

Raised when an error is detected that doesn’t fall in any of the other categories. The associated value is a string indicating what precisely went wrong. (This exception is mostly a relic from a previous version of the interpreter; it is not used very much any more.)

exception **StopIteration**

Raised by built-in function **next()** and an iterator’s **__next__()** method to signal that there are no further items produced by the iterator.
The exception object has a single attribute `value`, which is given as an argument when constructing the exception, and defaults to `None`.

When a generator function returns, a new `StopIteration` instance is raised, and the value returned by the function is used as the `value` parameter to the constructor of the exception. Changed in version 3.3: Added `value` attribute and the ability for generator functions to use it to return a value.

### Exception `SyntaxError`

Raised when the parser encounters a syntax error. This may occur in an `import` statement, in a call to the built-in functions `exec()` or `eval()`, or when reading the initial script or standard input (also interactively).

Instances of this class have attributes `filename`, `lineno`, `offset` and `text` for easier access to the details. `str()` of the exception instance returns only the message.

### Exception `IndentationError`

Base class for syntax errors related to incorrect indentation. This is a subclass of `SyntaxError`.

### Exception `TabError`

Raised when indentation contains an inconsistent use of tabs and spaces. This is a subclass of `IndentationError`.

### Exception `SystemError`

Raised when the interpreter finds an internal error, but the situation does not look so serious to cause it to abandon all hope. The associated value is a string indicating what went wrong (in low-level terms).

You should report this to the author or maintainer of your Python interpreter. Be sure to report the version of the Python interpreter (`sys.version`; it is also printed at the start of an interactive Python session), the exact error message (the exception’s associated value) and if possible the source of the program that triggered the error.

### Exception `SystemExit`

This exception is raised by the `sys.exit()` function. When it is not handled, the Python interpreter exits; no stack traceback is printed. If the associated value is an integer, it specifies the system exit status (passed to C’s `exit()` function); if it is `None`, the exit status is zero; if it has another type (such as a string), the object’s value is printed and the exit status is one.

Instances have an attribute `code` which is set to the proposed exit status or error message (defaulting to `None`). Also, this exception derives directly from `BaseException` and not `Exception`, since it is not technically an error.

A call to `sys.exit()` is translated into an exception so that clean-up handlers (finally clauses of try statements) can be executed, and so that a debugger can execute a script without running the risk of losing control. The `os._exit()` function can be used if it is absolutely positively necessary to exit immediately (for example, in the child process after a call to `os.fork()`).

The exception inherits from `BaseException` instead of `Exception` so that it is not accidentally caught by code that catches `Exception`. This allows the exception to properly propagate up and cause the interpreter to exit.

### Exception `TypeError`

Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

### Exception `UnboundLocalError`

Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. This is a subclass of `NameError`.

### Exception `UnicodeError`

Raised when a Unicode-related encoding or decoding error occurs. It is a subclass of `ValueError`.

`UnicodeError` has attributes that describe the encoding or decoding error. For example, `err.object[err.start:err.end]` gives the particular invalid input that the codec failed on.

### Encoding

The name of the encoding that raised the error.
reason
   A string describing the specific codec error.

object
   The object the codec was attempting to encode or decode.

start
   The first index of invalid data in object.

end
   The index after the last invalid data in object.

exception UnicodeEncodeError
   Raised when a Unicode-related error occurs during encoding. It is a subclass of UnicodeError.

exception UnicodeDecodeError
   Raised when a Unicode-related error occurs during decoding. It is a subclass of UnicodeError.

exception UnicodeTranslateError
   Raised when a Unicode-related error occurs during translating. It is a subclass of UnicodeError.

exception ValueError
   Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as IndexError.

exception ZeroDivisionError
   Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

The following exceptions are kept for compatibility with previous versions; starting from Python 3.3, they are aliases of OSError.

exception EnvironmentError

exception IOError

exception VMSError
   Only available on VMS.

exception WindowsError
   Only available on Windows.

5.2.1 OS exceptions

The following exceptions are subclasses of OSError, they get raised depending on the system error code.

exception BlockingIOError
   Raised when an operation would block on an object (e.g. socket) set for non-blocking operation. Corresponds to errno EAGAIN, EALREADY, EWOULDDBLOCK and EINPROGRESS.

   In addition to those of OSError, BlockingIOError can have one more attribute:

characters_written
   An integer containing the number of characters written to the stream before it blocked. This attribute is available when using the buffered I/O classes from the io module.

exception ChildProcessError
   Raised when an operation on a child process failed. Corresponds to errno ECHILD.

exception ConnectionError
   A base class for connection-related issues.

   Subclasses are BrokenPipeError, ConnectionAbortedError, ConnectionRefusedError and ConnectionResetError.

exception BrokenPipeError
   A subclass of ConnectionError, raised when trying to write on a pipe while the other end has been
exception ConnectionAbortedError
A subclass of ConnectionError, raised when a connection attempt is aborted by the peer. Corresponds to \texttt{errno} ECONNABORTED.

exception ConnectionRefusedError
A subclass of ConnectionError, raised when a connection attempt is refused by the peer. Corresponds to \texttt{errno} ECONNREFUSED.

exception ConnectionResetError
A subclass of ConnectionError, raised when a connection is reset by the peer. Corresponds to \texttt{errno} ECONNRESET.

exception FileExistsError
Raised when trying to create a file or directory which already exists. Corresponds to \texttt{errno} EEXIST.

exception FileNotFoundError
Raised when a file or directory is requested but doesn’t exist. Corresponds to \texttt{errno} ENOENT.

exception InterruptedError
Raised when a system call is interrupted by an incoming signal. Corresponds to \texttt{errno} EINTR.

exception IsADirectoryError
Raised when a file operation (such as \texttt{os.remove()}) is requested on a directory. Corresponds to \texttt{errno} EISDIR.

exception NotADirectoryError
Raised when a directory operation (such as \texttt{os.listdir()}) is requested on something which is not a directory. Corresponds to \texttt{errno} ENOTDIR.

exception PermissionError
Raised when trying to run an operation without the adequate access rights - for example filesystem permissions. Corresponds to \texttt{errno} EACCES and EPERM.

exception ProcessLookupError
Raised when a given process doesn’t exist. Corresponds to \texttt{errno} ESRCH.

exception TimeoutError
Raised when a system function timed out at the system level. Corresponds to \texttt{errno} ETIMEDOUT.

New in version 3.3: All the above OSError subclasses were added.

See Also:
PEP 3151 - Reworking the OS and IO exception hierarchy

5.3 Warnings

The following exceptions are used as warning categories; see the \texttt{warnings} module for more information.

exception Warning
Base class for warning categories.

exception UserWarning
Base class for warnings generated by user code.

exception DeprecationWarning
Base class for warnings about deprecated features.

exception PendingDeprecationWarning
Base class for warnings about features which will be deprecated in the future.

exception SyntaxWarning
Base class for warnings about dubious syntax
exception **RuntimeWarning**  
Base class for warnings about dubious runtime behavior.

exception **FutureWarning**  
Base class for warnings about constructs that will change semantically in the future.

exception **ImportWarning**  
Base class for warnings about probable mistakes in module imports.

exception **UnicodeWarning**  
Base class for warnings related to Unicode.

exception **BytesWarning**  
Base class for warnings related to `bytes` and `bytearray`.

exception **ResourceWarning**  
Base class for warnings related to resource usage. New in version 3.2.

### 5.4 Exception hierarchy

The class hierarchy for built-in exceptions is:

```
BaseException  
  +-- SystemExit  
  +-- KeyboardInterrupt  
  +-- GeneratorExit  
  +-- Exception  
      +-- StopIteration  
      +-- ArithmeticError  
          +-- FloatingPointError  
          +-- OverflowError  
          +-- ZeroDivisionError  
      +-- AssertionError  
      +-- AttributeError  
      +-- BufferError  
      +-- EOFError  
      +-- ImportError  
      +-- LookupError  
          +-- IndexError  
          +-- KeyError  
      +-- MemoryError  
      +-- NameError  
          +-- UnboundLocalError  
      +-- OSError  
          +-- BlockingIOError  
          +-- ChildProcessError  
          +-- ConnectionError  
              +-- BrokenPipeError  
              +-- ConnectionAbortedError  
              +-- ConnectionRefusedError  
              +-- ConnectionResetError  
          +-- FileExistsError  
          +-- FileNotFoundError  
          +-- InterruptedError  
          +-- IsADirectoryError  
          +-- NotADirectoryError  
          +-- PermissionError  
          +-- ProcessLookupError  
              +-- TimeoutError
```
The Python Library Reference, Release 3.3.3

Chapter 5. Built-in Exceptions

++-- ReferenceError
++-- RuntimeError
|   ++-- NotImplementedError
++-- SyntaxError
|   ++-- IndentationError
|   |   ++-- TabError
++-- SystemError
++-- TypeError
++-- ValueError
|   ++-- UnicodeError
|   |   ++-- UnicodeDecodeError
|   |   ++-- UnicodeEncodeError
|   |   ++-- UnicodeTranslateError
++-- Warning
    ++-- DeprecationWarning
    ++-- PendingDeprecationWarning
    ++-- RuntimeWarning
    ++-- SyntaxWarning
    ++-- UserWarning
    ++-- FutureWarning
    ++-- ImportWarning
    ++-- UnicodeWarning
    ++-- BytesWarning
    ++-- ResourceWarning
The modules described in this chapter provide a wide range of string manipulation operations and other text processing services.

The `codecs` module described under *Binary Data Services* is also highly relevant to text processing. In addition, see the documentation for Python’s built-in string type in *Text Sequence Type — str*.

### 6.1 string — Common string operations

**Source code:** Lib/string.py

**See Also:**

*Text Sequence Type — str*

*String Methods*

#### 6.1.1 String constants

The constants defined in this module are:

- **string.ascii_letters**
  - The concatenation of the `ascii_lowercase` and `ascii_uppercase` constants described below. This value is not locale-dependent.

- **string.ascii_lowercase**
  - The lowercase letters 'abcdefghijklmnopqrstuvwxyz'. This value is not locale-dependent and will not change.

- **string.ascii_uppercase**
  - The uppercase letters 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. This value is not locale-dependent and will not change.

- **string.digits**
  - The string '0123456789'.

- **string.hexdigits**
  - The string '0123456789abcdefABCDEF'.

- **string.octdigits**
  - The string '01234567'.

- **string.punctuation**
  - String of ASCII characters which are considered punctuation characters in the C locale.
The Python Library Reference, Release 3.3.3

string.printable
String of ASCII characters which are considered printable. This is a combination of digits, ascii_letters, punctuation, and whitespace.

string.whitespace
A string containing all ASCII characters that are considered whitespace. This includes the characters space, tab, linefeed, return, formfeed, and vertical tab.

6.1.2 String Formatting

The built-in string class provides the ability to do complex variable substitutions and value formatting via the format() method described in PEP 3101. The Formatter class in the string module allows you to create and customize your own string formatting behaviors using the same implementation as the built-in format() method.

class string.Formatter
The Formatter class has the following public methods:

format (format_string, *args, **kwargs)
format() is the primary API method. It takes a format string and an arbitrary set of positional and keyword arguments. format() is just a wrapper that calls vformat().

vformat (format_string, args, kwargs)
This function does the actual work of formatting. It is exposed as a separate function for cases where you want to pass in a predefined dictionary of arguments, rather than unpacking and repacking the dictionary as individual arguments using the *args and **kwargs syntax. vformat() does the work of breaking up the format string into character data and replacement fields. It calls the various methods described below.

In addition, the Formatter defines a number of methods that are intended to be replaced by subclasses:

parse (format_string)
Loop over the format_string and return an iterable of tuples (literal_text, field_name, format_spec, conversion). This is used by vformat() to break the string into either literal text, or replacement fields.

The values in the tuple conceptually represent a span of literal text followed by a single replacement field. If there is no literal text (which can happen if two replacement fields occur consecutively), then literal_text will be a zero-length string. If there is no replacement field, then the values of field_name, format_spec and conversion will be None.

get_field (field_name, args, kwargs)
Given field_name as returned by parse() (see above), convert it to an object to be formatted. Returns a tuple (obj, used_key). The default version takes strings of the form defined in PEP 3101, such as “0[name]” or “label.title”. args and kwargs are as passed in to vformat(). The return value used_key has the same meaning as the key parameter to get_value().

get_value (key, args, kwargs)
Retrieve a given field value. The key argument will be either an integer or a string. If it is an integer, it represents the index of the positional argument in args; if it is a string, then it represents a named argument in kwargs.

The args parameter is set to the list of positional arguments to vformat(), and the kwargs parameter is set to the dictionary of keyword arguments.

For compound field names, these functions are only called for the first component of the field name; Subsequent components are handled through normal attribute and indexing operations.

So for example, the field expression ‘0.name’ would cause get_value() to be called with a key argument of 0. The name attribute will be looked up after get_value() returns by calling the built-in getattr() function.
If the index or keyword refers to an item that does not exist, then an `IndexError` or `KeyError` should be raised.

**check_unused_args** *(used_args, args, kwargs)*
Implement checking for unused arguments if desired. The arguments to this function is the set of all argument keys that were actually referred to in the format string (integers for positional arguments, and strings for named arguments), and a reference to the `args` and `kwargs` that was passed to `vformat`. The set of unused args can be calculated from these parameters. `check_unused_args()` is assumed to raise an exception if the check fails.

**format_field** *(value, format_spec)*
`format_field()` simply calls the global `format()` built-in. The method is provided so that subclasses can override it.

**convert_field** *(value, conversion)*
Converts the value (returned by `get_field()`) given a conversion type (as in the tuple returned by the `parse()` method). The default version understands ‘s’ (str), ‘r’ (repr) and ‘a’ (ascii) conversion types.

### 6.1.3 Format String Syntax

The `str.format()` method and the `Formatter` class share the same syntax for format strings (although in the case of `Formatter`, subclasses can define their own format string syntax).

Format strings contain “replacement fields” surrounded by curly braces `{}`. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: `{ { and } }`.

The grammar for a replacement field is as follows:

```asciigray
replacement_field ::= "{" [field_name] ["!" conversion] [":" format_spec"]}
field_name ::= arg_name ("." attribute_name | "[" element_index "]")+
arg_name ::= [identifier | integer]
attribute_name ::= identifier
element_index ::= integer | index_string
index_string ::= <any source character except "]"> +
conversion ::= "r" | "s" | "a"
format_spec ::= <described in the next section>
```

In less formal terms, the replacement field can start with a *field_name* that specifies the object whose value is to be formatted and inserted into the output instead of the replacement field. The *field_name* is optionally followed by a *conversion* field, which is preceded by an exclamation point ‘!’ and a *format_spec*, which is preceded by a colon ‘:’. These specify a non-default format for the replacement value.

See also the *Format Specification Mini-Language* section.

The *field_name* itself begins with an *arg_name* that is either a number or a keyword. If it’s a number, it reverts to a positional argument, and if it’s a keyword, it refers to a named keyword argument. If the numerical arg_names in a format string are 0, 1, 2, ... in sequence, they can all be omitted (not just some) and the numbers 0, 1, 2, ... will be automatically inserted in that order. Because *arg_name* is not quote-delimited, it is not possible to specify arbitrary dictionary keys (e.g., the strings ‘10’ or ‘:1’) within a format string. The *arg_name* can be followed by any number of index or attribute expressions. An expression of the form ‘.name’ selects the named attribute using `getattr()`, while an expression of the form ‘[index]’ does an index lookup using `__getitem__()`. Changed in version 3.1: The positional argument specifiers can be omitted, so `{0} {1}` is equivalent to `{0} {1}`.

Some simple format string examples:

```python
"First, thou shalt count to (0)"
# References first positional argument
"Bring me a {}"
# Implicitly references the first positional argument
"From {} to {}"
# Same as "From (0) to (1)"
"My quest is {name}"
# References keyword argument ‘name’
```
"Weight in tons {0.weight}"  # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}"  # First element of keyword argument 'players'.

The conversion field causes a type coercion before formatting. Normally, the job of formatting a value is done by the __format__() method of the value itself. However, in some cases it is desirable to force a type to be formatted as a string, overriding its own definition of formatting. By converting the value to a string before calling __format__(), the normal formatting logic is bypassed.

Three conversion flags are currently supported: ’!s’ which calls str() on the value, ’!r’ which calls repr() and ’!a’ which calls ascii().

Some examples:

"Harold’s a clever {0!s}"    # Calls str() on the argument first
"Bring out the holy {name!r}"  # Calls repr() on the argument first
"More {!a}"                    # Calls ascii() on the argument first

The format_spec field contains a specification of how the value should be presented, including such details as field width, alignment, padding, decimal precision and so on. Each value type can define its own “formatting mini-language” or interpretation of the format_spec.

Most built-in types support a common formatting mini-language, which is described in the next section.

A format_spec field can also include nested replacement fields within it. These nested replacement fields can contain only a field name; conversion flags and format specifications are not allowed. The replacement fields within the format_spec are substituted before the format_spec string is interpreted. This allows the formatting of a value to be dynamically specified.

See the Format examples section for some examples.

Format Specification Mini-Language

“Format specifications” are used within replacement fields contained within a format string to define how individual values are presented (see Format String Syntax). They can also be passed directly to the built-in format() function. Each formattable type may define how the format specification is to be interpreted.

Most built-in types implement the following options for format specifications, although some of the formatting options are only supported by the numeric types.

A general convention is that an empty format string ("") produces the same result as if you had called str() on the value. A non-empty format string typically modifies the result.

The general form of a standard format specifier is:

format_spec ::= [[fill][align][sign][#][0][width][,][.precision]][type]
fill ::= <any character>
align ::= "<" | ">" | "=" | "^"
sign ::= "+" | "-" | " "
width ::= integer
precision ::= integer
type ::= "b" | "c" | "d" | "e" | "f" | "e" | "E" | "F" | "g" | "G" | "n" | "o" |...
### Option Meaning

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'&lt;'</td>
<td>Forcing the field to be left-aligned within the available space (this is the default for most objects).</td>
</tr>
<tr>
<td>'&gt;'</td>
<td>Forcing the field to be right-aligned within the available space (this is the default for numbers).</td>
</tr>
<tr>
<td>'='</td>
<td>Forcing the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form ‘+000000120’. This alignment option is only valid for numeric types.</td>
</tr>
<tr>
<td>'^'</td>
<td>Forcing the field to be centered within the available space.</td>
</tr>
</tbody>
</table>

Note that unless a minimum field width is defined, the field width will always be the same size as the data to fill it, so that the alignment option has no meaning in this case.

The *sign* option is only valid for number types, and can be one of the following:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'+'</td>
<td>Indicates that a sign should be used for both positive as well as negative numbers.</td>
</tr>
<tr>
<td>'-'</td>
<td>Indicates that a sign should be used only for negative numbers (this is the default behavior).</td>
</tr>
<tr>
<td>space</td>
<td>Indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.</td>
</tr>
</tbody>
</table>

The `'#'` option causes the “alternate form” to be used for the conversion. The alternate form is defined differently for different types. This option is only valid for integer, float, complex and Decimal types. For integers, when binary, octal, or hexadecimal output is used, this option adds the prefix respective `0b`, `0o`, or `0x` to the output value. For floats, complex and Decimal the alternate form causes the result of the conversion to always contain a decimal-point character, even if no digits follow it. Normally, a decimal-point character appears in the result of these conversions only if a digit follows it. In addition, for `g` and `G` conversions, trailing zeros are not removed from the result.

The `,` option signals the use of a comma for a thousands separator. For a locale aware separator, use the `n` integer presentation type instead. Changed in version 3.1: Added the `,` option (see also PEP 378). `width` is a decimal integer defining the minimum field width. If not specified, then the field width will be determined by the content.

Preceding the `width` field by a zero (`0`) character enables sign-aware zero-padding for numeric types. This is equivalent to a fill character of `'0'` with an alignment type of `'=.'`.

The `precision` is a decimal number indicating how many digits should be displayed after the decimal point for a floating point value formatted with `'f'` and `'F'`, or before and after the decimal point for a floating point value formatted with `'g'` or `'G'`. For non-number types the field indicates the maximum field size - in other words, how many characters will be used from the field content. The `precision` is not allowed for integer values.

Finally, the `type` determines how the data should be presented.

The available string presentation types are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s</code></td>
<td>String format. This is the default type for strings and may be omitted. The same as <code>s</code>.</td>
</tr>
<tr>
<td>None</td>
<td>Same as <code>s</code>.</td>
</tr>
</tbody>
</table>

The available integer presentation types are:
In addition to the above presentation types, integers can be formatted with the floating point presentation types listed below (except ‘n’ and None). When doing so, `float()` is used to convert the integer to a floating point number before formatting.

The available presentation types for floating point and decimal values are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘e’</td>
<td>Exponent notation. Prints the number in scientific notation using the letter ‘e’ to indicate the exponent. The default precision is 6.</td>
</tr>
<tr>
<td>‘E’</td>
<td>Exponent notation. Same as ‘e’ except it uses an upper case ‘E’ as the separator character.</td>
</tr>
<tr>
<td>‘f’</td>
<td>Fixed point. Displays the number as a fixed-point number. The default precision is 6.</td>
</tr>
<tr>
<td>‘F’</td>
<td>Fixed point. Same as ‘f’, but converts nan to NAN and inf to INF.</td>
</tr>
<tr>
<td>‘g’</td>
<td>General format. For a given precision p &gt;= 1, this rounds the number to p significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude. The precise rules are as follows: suppose that the result formatted with presentation type ‘e’ and precision p-1 would have exponent exp. Then if -4 &lt;= exp &lt; p, the number is formatted with presentation type ‘f’ and precision p-1-exp. Otherwise, the number is formatted with presentation type ‘e’ and precision p-1. In both cases insignificant trailing zeros are removed from the significand, and the decimal point is also removed if there are no remaining digits following it. Positive and negative infinity, positive and negative zero, and nans, are formatted as inf, -inf, 0, -0 and nan respectively, regardless of the precision. A precision of 0 is treated as equivalent to a precision of 1. The default precision is 6.</td>
</tr>
<tr>
<td>‘G’</td>
<td>General format. Same as ‘g’ except switches to ‘E’ if the number gets too large. The representations of infinity and NaN are uppercased, too.</td>
</tr>
<tr>
<td>‘n’</td>
<td>Number. This is the same as ‘g’, except that it uses the current locale setting to insert the appropriate number separator characters.</td>
</tr>
<tr>
<td>‘%’</td>
<td>Percentage. Multiplies the number by 100 and displays in fixed (‘f’) format, followed by a percent sign.</td>
</tr>
<tr>
<td>‘%’</td>
<td>Similar to ‘g’, except with at least one digit past the decimal point and a default precision of 12. This is intended to match <code>str()</code>, except you can add the other format modifiers.</td>
</tr>
</tbody>
</table>

Format examples

This section contains examples of the new format syntax and comparison with the old %-formatting.

In most of the cases the syntax is similar to the old %-formatting, with the addition of the () and with : used instead of %. For example, `%03.2f` can be translated to `{:03.2f}`.

The new format syntax also supports new and different options, shown in the follow examples.

Accessing arguments by position:

```python
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 3.1+ only
```

Chapter 6. Text Processing Services
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')  # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')  # arguments’ indices can be repeated
'abracadabra'

Accessing arguments by name:

```python
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

Accessing arguments’ attributes:

```python
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} ' ...
' and the imaginary part {0.imag}').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part -5.0.'
```

```python
>>> class Point:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

Accessing arguments’ items:

```python
>>> coord = (3, 5)
>>> 'X: {0[0]}; Y: {0[1]}'.format(coord)
'X: 3; Y: 5'
```

Replacing `%%s` and `%%r`:

```python
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
'repr() shows quotes: 'test1'; str() doesn’t: test2"
```

Aligning the text and specifying a width:

```python
>>> '{:<30}'.format('left aligned')
'left aligned'
>>> '{:>30}'.format('right aligned')
' right aligned'
>>> '{:^30}'.format('centered')
' centered
'**********centered**********'
```
>>> # format also supports binary numbers
>>> "int: {0:d}; hex: {0:x}; oct: {0:o}; bin: {0:b}".format(42)
'int: 42; hex: 2a; oct: 52; bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d}; hex: {0:#x}; oct: {0:#o}; bin: {0:#b}".format(42)
'int: 42; hex: 0x2a; oct: 0o52; bin: 0b101010'

Using the comma as a thousands separator:
>>> '{:,}'.format(1234567890)
'1,234,567,890'

Expressing a percentage:
>>> points = 19
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 86.36%'

Using type-specific formatting:
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'

Nesting arguments and more complex examples:
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...      '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...      'left<<<<<<<<<<<'
...      '^^^^^^center^^^^^^'
...      '>>>>>>>>>>>right'
...>
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
>>> int(_, 16)
3232235521
>>> width = 5
>>> for num in range(5,12):
...      for base in 'dXob':
...          print('{0:{width}{base}}'.format(num, base=base, width=width), end=' ')
...          print()
...          5 5 5 101
6 6 6 110
7 7 7 111
8 8 10 1000
9 9 11 1001
10 A 12 1010
11 B 13 1011

6.1.4 Template strings

Templates provide simpler string substitutions as described in PEP 292. Instead of the normal %-based substitutions, Templates support $-based substitutions, using the following rules:

- $$ is an escape; it is replaced with a single $.
$identifier names a substitution placeholder matching a mapping key of "identifier". By default, "identifier" must spell a Python identifier. The first non-identifier character after the $ character terminates this placeholder specification.

${identifier} is equivalent to $identifier. It is required when valid identifier characters follow the placeholder but are not part of the placeholder, such as "${noun}ification".

Any other appearance of $ in the string will result in a ValueError being raised.

The string module provides a Template class that implements these rules. The methods of Template are:

class string.Template(template)
The constructor takes a single argument which is the template string.

substitute(mapping, **kwds)
Performs the template substitution, returning a new string. mapping is any dictionary-like object with keys that match the placeholders in the template. Alternatively, you can provide keyword arguments, where the keywords are the placeholders. When both mapping and kwds are given and there are duplicates, the placeholders from kwds take precedence.

safe_substitute(mapping, **kwds)
Like substitute(), except that if placeholders are missing from mapping and kwds, instead of raising a KeyError exception, the original placeholder will appear in the resulting string intact. Also, unlike with substitute(), any other appearances of the $ will simply return $ instead of raising ValueError.

While other exceptions may still occur, this method is called “safe” because substitutions always tries to return a usable string instead of raising an exception. In another sense, safe_substitute() may be anything other than safe, since it will silently ignore malformed templates containing dangling delimiters, unmatched braces, or placeholders that are not valid Python identifiers.

Template instances also provide one public data attribute:

template
This is the object passed to the constructor’s template argument. In general, you shouldn’t change it, but read-only access is not enforced.

Here is an example of how to use a Template:

```python
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
  ... ValueError: Invalid placeholder in string: line 1, col 11
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
  ...
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

Advanced usage: you can derive subclasses of Template to customize the placeholder syntax, delimiter character, or the entire regular expression used to parse template strings. To do this, you can override these class attributes:

- delimiter – This is the literal string describing a placeholder introducing delimiter. The default value is $. Note that this should not be a regular expression, as the implementation will call re.escape() on this string as needed.

- idpattern – This is the regular expression describing the pattern for non-braced placeholders (the braces will be added automatically as appropriate). The default value is the regular expression [\_a-z][\_a-z0-9]*.
• **flags** – The regular expression flags that will be applied when compiling the regular expression used for recognizing substitutions. The default value is `re.IGNORECASE`. Note that `re.VERBOSE` will always be added to the flags, so custom *idpatterns* must follow conventions for verbose regular expressions. New in version 3.2.

Alternatively, you can provide the entire regular expression pattern by overriding the class attribute *pattern*. If you do this, the value must be a regular expression object with four named capturing groups. The capturing groups correspond to the rules given above, along with the invalid placeholder rule:

• **escaped** – This group matches the escape sequence, e.g. `$$`, in the default pattern.

• **named** – This group matches the unbraced placeholder name; it should not include the delimiter in capturing group.

• **braced** – This group matches the brace enclosed placeholder name; it should not include either the delimiter or braces in the capturing group.

• **invalid** – This group matches any other delimiter pattern (usually a single delimiter), and it should appear last in the regular expression.

### 6.1.5 Helper functions

`string.capwords(s, sep=None)`

Split the argument into words using `str.split()`, capitalize each word using `str.capitalize()`, and join the capitalized words using `str.join()`. If the optional second argument *sep* is absent or `None`, runs of whitespace characters are replaced by a single space and leading and trailing whitespace are removed, otherwise *sep* is used to split and join the words.

### 6.2 `re` — Regular expression operations

This module provides regular expression matching operations similar to those found in Perl.

Both patterns and strings to be searched can be Unicode strings as well as 8-bit strings. However, Unicode strings and 8-bit strings cannot be mixed: that is, you cannot match an Unicode string with a byte pattern or vice-versa; similarly, when asking for a substitution, the replacement string must be of the same type as both the pattern and the search string.

Regular expressions use the backslash character (`'\'`) to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python’s usage of the same character for the same purpose in string literals; for example, to match a literal backslash, one might have to write `'\\'` as the pattern string, because the regular expression must be `\`, and each backslash must be expressed as `\` inside a regular Python string literal.

The solution is to use Python’s raw string notation for regular expression patterns; backslashes are not handled in any special way in a string literal prefixed with `r`. So `r"\n"` is a two-character string containing `'\n'`, and `r'n'`, while `"\n"` is a one-character string containing a newline. Usually patterns will be expressed in Python code using this raw string notation.

It is important to note that most regular expression operations are available as module-level functions and methods on compiled regular expressions. The functions are shortcuts that don’t require you to compile a regex object first, but miss some fine-tuning parameters.

**See Also:**

**Mastering Regular Expressions** Book on regular expressions by Jeffrey Friedl, published by O’Reilly. The second edition of the book no longer covers Python at all, but the first edition covered writing good regular expression patterns in great detail.
6.2.1 Regular Expression Syntax

A regular expression (or RE) specifies a set of strings that matches it; the functions in this module let you check if a particular string matches a given regular expression (or if a given regular expression matches a particular string, which comes down to the same thing).

Regular expressions can be concatenated to form new regular expressions; if A and B are both regular expressions, then AB is also a regular expression. In general, if a string p matches A and another string q matches B, the string pq will match AB. This holds unless A or B contain low precedence operations; boundary conditions between A and B; or have numbered group references. Thus, complex expressions can easily be constructed from simpler primitive expressions like the ones described here. For details of the theory and implementation of regular expressions, consult the Friedl book referenced above, or almost any textbook about compiler construction.

A brief explanation of the format of regular expressions follows. For further information and a gentler presentation, consult the regex-howto.

Regular expressions can contain both special and ordinary characters. Most ordinary characters, like ‘A’, ‘a’, or ‘0’, are the simplest regular expressions; they simply match themselves. You can concatenate ordinary characters, so last matches the string ‘last’. (In the rest of this section, we’ll write RE’s in this special style, usually without quotes, and strings to be matched ‘in single quotes’.)

Some characters, like ‘|’, or ‘(’, are special. Special characters either stand for classes of ordinary characters, or affect how the regular expressions around them are interpreted. Regular expression pattern strings may not contain null bytes, but can specify the null byte using a \number notation such as ‘\x00’.

The special characters are:

'.' (Dot.) In the default mode, this matches any character except a newline. If the DOTALL flag has been specified, this matches any character including a newline.

'^' (Caret.) Matches the start of the string, and in MULTILINE mode also matches immediately after each newline.

'\$' Matches the end of the string or just before the newline at the end of the string, and in MULTILINE mode also matches before a newline. foo matches both ‘foo’ and ‘fooobar’, while the regular expression foo$ matches only ‘foo’. More interestingly, searching for foo.$ in ‘foool\nfoo2\n’ matches ‘foo2’ normally, but ‘fool’ in MULTILINE mode; searching for a single $ in ‘foo\n’ will find two (empty) matches: one just before the newline, and one at the end of the string.

'*' Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. ab+ will match ‘a’, ‘ab’, or ‘a’ followed by any number of ‘b’ s.

'+' Causes the resulting RE to match 1 or more repetitions of the preceding RE. ab+ will match ‘a’ followed by any non-zero number of ‘b’ s; it will not match just ‘a’.

'? ' Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. ab? will match either ‘a’ or ‘ab’.

'?', '+?', '??' The ‘*’, ‘+’, and ‘?’ qualifiers are all greedy; they match as much text as possible. Sometimes this behaviour isn’t desired; if the RE <.+> is matched against ‘<H1>title</H1>‘, it will match the entire string, and not just ‘<H1>‘. Adding ‘? ’ after the qualifier makes it perform the match in non-greedy or minimal fashion; as few characters as possible will be matched. Using .? in the previous expression will match only ‘<H1>‘.

(m) Specifies that exactly m copies of the previous RE should be matched; fewer matches cause the entire RE not to match. For example, a{6} will match exactly six ‘a’ characters, but not five.

(m, n) Causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, a{3,5} will match from 3 to 5 ‘a’ characters. Omitting m specifies a lower bound of zero, and omitting n specifies an infinite upper bound. As an example, a{4,}b will match aaab or a thousand ‘a’ characters followed by a b, but not aaab. The comma may not be omitted or the modifier would be confused with the previously described form.

(m, n) ? Causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as few repetitions as possible. This is the non-greedy version of the previous qualifier. For example, on the 6-character string ‘aaaaaa’, a{3,5} will match 5 ‘a’ characters, while a(3, 5)? will only match 3 characters.
Either escapes special characters (permitting you to match characters like `*`, `?`, and so forth), or signals a special sequence; special sequences are discussed below.

If you’re not using a raw string to express the pattern, remember that Python also uses the backslash as an escape sequence in string literals; if the escape sequence isn’t recognized by Python’s parser, the backslash and subsequent character are included in the resulting string. However, if Python would recognize the resulting sequence, the backslash should be repeated twice. This is complicated and hard to understand, so it’s highly recommended that you use raw strings for all but the simplest expressions.

Used to indicate a set of characters. In a set:

- Characters can be listed individually, e.g. `[amk]` will match `a`, `m`, or `k`.
- Ranges of characters can be indicated by giving two characters and separating them by a `-`, for example `[a-z]` will match any lowercase ASCII letter, `[0-5][0-9]` will match all the two-digits numbers from 00 to 59, and `[0-9A-Fa-f]` will match any hexadecimal digit. If `-` is escaped (e.g. `[a\-z]`) or if it’s placed as the first or last character (e.g. `[a-]`), it will match a literal `-`.
- Special characters lose their special meaning inside sets. For example, `{(++)}` will match any of the literal characters `(`, `+`, `*`, or `)`.
- Character classes such as `\w` or `\s` (defined below) are also accepted inside a set, although the characters they match depends on whether `ASCII` or `locale` mode is in force.
- Characters that are not within a range can be matched by complementing the set. If the first character of the set is `^`, all the characters that are not in the set will be matched. For example, `[^5]` will match any character except `5`, and `[^^]` will match any character except `^`. ^ has no special meaning if it’s not the first character in the set.
- To match a literal `|` inside a set, precede it with a backslash, or place it at the beginning of the set. For example, both `() [] {}` and `[] [] []` will both match a parenthesis.

A B, where A and B can be arbitrary REs, creates a regular expression that will match either A or B. An arbitrary number of REs can be separated by the `|` in this way. This can be used inside groups (see below) as well. As the target string is scanned, REs separated by `|` are tried from left to right. When one pattern completely matches, that branch is accepted. This means that once a matches, B will not be tested further, even if it would produce a longer overall match. In other words, the `|` operator is never greedy. To match a literal `|`, use `\|`, or enclose it inside a character class, as in `[|]`.

(. . .) Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed, and can be matched later in the string with the `number` special sequence, described below. To match the literals `( ` or `)``, use `\( ` or `\)`, or enclose them inside a character class: `[{] [}]`.

(? . . .) This is an extension notation (a `?` following a `(` is not meaningful otherwise). The first character after the `(?` determines what the meaning and further syntax of the construct is. Extensions usually do not create a new group; `(?P<name>...)` is the only exception to this rule. Following are the currently supported extensions.

(?ailmsux) (One or more letters from the set `a`, `i`, `L`, `m`, `s`, `u`, `x`.) The group matches the empty string; the letters set the corresponding flags: `re.A` (ASCII-only matching), `re.I` (ignore case), `re.L` (locale dependent), `re.M` (multi-line), `re.S` (dot matches all), and `re.X` (verbose), for the entire regular expression. (The flags are described in `Module Contents`.) This is useful if you wish to include the flags as part of the regular expression, instead of passing a `flag` argument to the `re.compile()` function.

Note that the `(?)` flag changes how the expression is parsed. It should be used first in the expression string, or after one or more whitespace characters. If there are non-whitespace characters before the flag, the results are undefined.

(? . . .) A non-capturing version of regular parentheses. Matches whatever regular expression is inside the parentheses, but the substring matched by the group cannot be retrieved after performing a match or referenced later in the pattern. 

(?P<name>...) Similar to regular parentheses, but the substring matched by the group is accessible via the symbolic group name `name`. Group names must be valid Python identifiers, and each group name must be
defined only once within a regular expression. A symbolic group is also a numbered group, just as if the group were not named.

Named groups can be referenced in three contexts. If the pattern is (?P<quote>['"']).*?(?P=quote) (i.e. matching a string quoted with either single or double quotes):

<table>
<thead>
<tr>
<th>Context of reference to group “quote”</th>
<th>Ways to reference it</th>
</tr>
</thead>
<tbody>
<tr>
<td>in the same pattern itself</td>
<td>• (?P=quote) (as shown)</td>
</tr>
<tr>
<td></td>
<td>• \1</td>
</tr>
<tr>
<td>when processing match object m</td>
<td>• m.group(‘quote’)</td>
</tr>
<tr>
<td></td>
<td>• m.end(‘quote’) (etc.)</td>
</tr>
<tr>
<td>in a string passed to the repl argument of re.sub()</td>
<td>• \g&lt;quote&gt;</td>
</tr>
<tr>
<td></td>
<td>• \g&lt;1&gt;</td>
</tr>
<tr>
<td></td>
<td>• \1</td>
</tr>
</tbody>
</table>

(?P=name) A backreference to a named group; it matches whatever text was matched by the earlier group named name.

(?#... ) A comment; the contents of the parentheses are simply ignored.

(?=...) Matches if ... matches next, but doesn’t consume any of the string. This is called a lookahead assertion. For example, Isaac (?=Asimov) will match ‘Isaac ’ only if it’s followed by ‘Asimov’.

(?!...) Matches if ... doesn’t match next. This is a negative lookahead assertion. For example, Isaac (?!Asimov) will match ‘Isaac ’ only if it’s not followed by ‘Asimov’.

(?<=...) Matches if the current position in the string is preceded by a match for ... that ends at the current position. This is called a positive lookbehind assertion. (?<=abc)def will find a match in abcdef, since the lookbehind will back up 3 characters and check if the contained pattern matches. The contained pattern must only match strings of some fixed length, meaning that abc or a|b are allowed, but a* and a{3,4} are not. Note that patterns which start with positive lookbehind assertions will not match at the beginning of the string being searched; you will most likely want to use the search() function rather than the match() function:

```python
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

This example looks for a word following a hyphen:

```python
>>> m = re.search('(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

(?!<...) Matches if the current position in the string is not preceded by a match for .... This is called a negative lookbehind assertion. Similar to positive lookbehind assertions, the contained pattern must only match strings of some fixed length. Patterns which start with negative lookbehind assertions may match at the beginning of the string being searched.

(?(id/name)yes-pattern|no-pattern) Will try to match with yes-pattern if the group with given id or name exists, and with no-pattern if it doesn’t. no-pattern is optional and can be omitted. For example, (<)?(\w+@\w+(?:\..\w+)+)(?(1)>|$) is a poor email matching pattern, which will match with ’<user@host.com>’ as well as ’user@host.com’, but not with ’<user@host.com’ nor ’user@host.com>’.

The special sequences consist of ‘ \ ’ and a character from the list below. If the ordinary character is not on the list, then the resulting RE will match the second character. For example, \$ matches the character ’$’.  

6.2. re — Regular expression operations 85
\number Matches the contents of the group of the same number. Groups are numbered starting from 1. For example, (.+) \l matches ‘the the’ or ‘55 55’, but not ‘thethe’ (note the space after the group). This special sequence can only be used to match one of the first 99 groups. If the first digit of \number is 0, or \number is 3 octal digits long, it will not be interpreted as a group match, but as the character with octal value \number. Inside the ‘[‘ and ‘]’ of a character class, all numeric escapes are treated as characters.

\A Matches only at the start of the string.

\b Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of Unicode alphanumeric or underscore characters, so the end of a word is indicated by whitespace or a non-alphanumeric, non-underscore Unicode character. Note that formally, \b is defined as the boundary between a \w and a \W character (or vice versa), or between \W and the beginning/end of the string. This means that r’\bfoo\b’ matches ‘foo’, ‘foo.’, ‘(foo)’, ‘bar foo baz’ but not ‘foobaz’ or ‘foo3’.

By default Unicode alphanumerics are the ones used, but this can be changed by using the ASCII flag. Inside a character range, \b represents the backspace character, for compatibility with Python’s string literals.

\B Matches the empty string, but only when it is not at the beginning or end of a word. This means that r’\py\B’ matches ‘python’, ‘py3’, ‘py2’, but not ‘py’, ‘py.’, or ‘py!’ \B is just the opposite of \b, so word characters are Unicode alphanumerics or the underscore, although this can be changed by using the ASCII flag.

\d For Unicode (str) patterns: Matches any Unicode decimal digit (that is, any character in Unicode character category [Nd]). This includes [0-9], and also many other digit characters. If the ASCII flag is used only [0-9] is matched (but the flag affects the entire regular expression, so in such cases using an explicit [0-9] may be a better choice).

For 8-bit (bytes) patterns: Matches any decimal digit; this is equivalent to [0-9].

\D Matches any character which is not a Unicode decimal digit. This is the opposite of \d. If the ASCII flag is used this becomes the equivalent of [^0-9] (but the flag affects the entire regular expression, so in such cases using an explicit [^0-9] may be a better choice).

\s For Unicode (str) patterns: Matches Unicode whitespace characters (which includes [ \t\n\r\f\v], and also many other characters, for example the non-breaking spaces mandated by typography rules in many languages). If the ASCII flag is used, only [ \t\n\r\f\v] is matched (but the flag affects the entire regular expression, so in such cases using an explicit [ \t\n\r\f\v] may be a better choice).

For 8-bit (bytes) patterns: Matches characters considered whitespace in the ASCII character set; this is equivalent to [ \t\n\r\f\v].

\S Matches any character which is not a Unicode whitespace character. This is the opposite of \s. If the ASCII flag is used this becomes the equivalent of [^ \t\n\r\f\v] (but the flag affects the entire regular expression, so in such cases using an explicit [^ \t\n\r\f\v] may be a better choice).

\w For Unicode (str) patterns: Matches Unicode word characters; this includes most characters that can be part of a word in any language, as well as numbers and the underscore. If the ASCII flag is used, only [a-zA-Z0-9_] is matched (but the flag affects the entire regular expression, so in such cases using an explicit [a-zA-Z0-9_] may be a better choice).

For 8-bit (bytes) patterns: Matches characters considered alphanumeric in the ASCII character set; this is equivalent to [a-zA-Z0-9_].

\W Matches any character which is not a Unicode word character. This is the opposite of \w. If the ASCII flag is used this becomes the equivalent of [^a-zA-Z0-9_] (but the flag affects the entire regular expression, so in such cases using an explicit [^a-zA-Z0-9_] may be a better choice).
\Z Matches only at the end of the string.

Most of the standard escapes supported by Python string literals are also accepted by the regular expression parser:

\a \b \f \n \r \t \u \U
\v \x \\n
(Note that \b is used to represent word boundaries, and means “backspace” only inside character classes.)

‘\u’ and ‘\U’ escape sequences are only recognized in Unicode patterns. In bytes patterns they are not treated specially.

Octal escapes are included in a limited form. If the first digit is a 0, or if there are three octal digits, it is considered an octal escape. Otherwise, it is a group reference. As for string literals, octal escapes are always at most three digits in length. Changed in version 3.3: The ‘\u’ and ‘\U’ escape sequences have been added.

6.2.2 Module Contents

The module defines several functions, constants, and an exception. Some of the functions are simplified versions of the full featured methods for compiled regular expressions. Most non-trivial applications always use the compiled form.

re.compile(pattern, flags=0)
Compile a regular expression pattern into a regular expression object, which can be used for matching using its match() and search() methods, described below.

The expression’s behaviour can be modified by specifying a flags value. Values can be any of the following variables, combined using bitwise OR (the | operator).

The sequence

prog = re.compile(pattern)
result = prog.match(string)

is equivalent to

result = re.match(pattern, string)

but using re.compile() and saving the resulting regular expression object for reuse is more efficient when the expression will be used several times in a single program.

Note: The compiled versions of the most recent patterns passed to re.match(), re.search() or re.compile() are cached, so programs that use only a few regular expressions at a time needn’t worry about compiling regular expressions.

re.A
re.ASCII
Make \w, \W, \b, \B, \d, \D, \s and \S perform ASCII-only matching instead of full Unicode matching. This is only meaningful for Unicode patterns, and is ignored for byte patterns.

Note that for backward compatibility, the re.U flag still exists (as well as its synonym re.UNICODE and its embedded counterpart (?u)), but these are redundant in Python 3 since matches are Unicode by default for strings (and Unicode matching isn’t allowed for bytes).

re.DEBUG
Display debug information about compiled expression.

re.I
re.IGNORECASE
Perform case-insensitive matching; expressions like [A-Z] will match lowercase letters, too. This is not affected by the current locale and works for Unicode characters as expected.
The Python Library Reference, Release 3.3.3

\texttt{re.L}

Make \texttt{\w, \W, \b, \B, \s} and \texttt{\S} dependent on the current locale. The use of this flag is discouraged as the locale mechanism is very unreliable, and it only handles one “culture” at a time anyway; you should use Unicode matching instead, which is the default in Python 3 for Unicode (str) patterns.

\texttt{re.M}

\texttt{re.MULTILINE}

When specified, the pattern character ‘\^’ matches at the beginning of the string and at the beginning of each line (immediately following each newline); and the pattern character ‘\$’ matches at the end of the string and at the end of each line (immediately preceding each newline). By default, ‘\^’ matches only at the beginning of the string, and ‘\$’ only at the end of the string and immediately before the newline (if any) at the end of the string.

\texttt{re.S}

\texttt{re.DOTALL}

Make the ‘.’ special character match any character at all, including a newline; without this flag, ‘.’ will match anything \textit{except} a newline.

\texttt{re.X}

\texttt{re.VERBOSE}

This flag allows you to write regular expressions that look nicer. Whitespace within the pattern is ignored, except when in a character class or preceded by an unescaped backslash, and, when a line contains a ‘\#’ neither in a character class or preceded by an unescaped backslash, all characters from the leftmost such ‘\#’ through the end of the line are ignored.

That means that the two following regular expression objects that match a decimal number are functionally equal:

\begin{verbatim}
a = re.compile(r"\d + # the integral part \. # the decimal point \d * # some fractional digits"", re.X)
b = re.compile(r"\d+\.\d*")
\end{verbatim}

\texttt{re.search (pattern, string, flags=0)}

Scan through \texttt{string} looking for a location where the regular expression \texttt{pattern} produces a match, and return a corresponding \texttt{match object}. Return None if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

\texttt{re.match (pattern, string, flags=0)}

If zero or more characters at the beginning of \texttt{string} match the regular expression \texttt{pattern}, return a corresponding \texttt{match object}. Return None if the string does not match the pattern; note that this is different from a zero-length match.

Note that even in \texttt{MULTILINE} mode, \texttt{re.match()} will only match at the beginning of the string and not at the beginning of each line.

If you want to locate a match anywhere in \texttt{string}, use \texttt{search()} instead (see also \texttt{search()} vs. \texttt{match}()).

\texttt{re.split (pattern, string, maxsplit=0, flags=0)}

Split \texttt{string} by the occurrences of \texttt{pattern}. If capturing parentheses are used in \texttt{pattern}, then the text of all groups in the pattern are also returned as part of the resulting list. If \texttt{maxsplit} is nonzero, at most \texttt{maxsplit} splits occur, and the remainder of the string is returned as the final element of the list.

\begin{verbatim}
>>> re.split(’\W+’, ’Words, words, words.’)
[’Words’, ’words’, ’words’, ’’]
>>> re.split(’(\W+)’, ’Words, words, words.’)
[’Words’, ’’, ’words’, ’’, ’’, ’’, ’’]
>>> re.split(’\W+’, ’Words, words, words.’, 1)
[’Words’, ’words, words.’]
>>> re.split(’[a-f]+’, ’0a3B9’, flags=re.IGNORECASE)
[’0’, ’3’, ’9’]
\end{verbatim}
If there are capturing groups in the separator and it matches at the start of the string, the result will start with an empty string. The same holds for the end of the string:

```python
>>> re.split('(?:\W+)', '...words, words...')
["", "...", "words", ",", "words", ",...", "]
```

That way, separator components are always found at the same relative indices within the result list.

Note that `split` will never split a string on an empty pattern match. For example:

```python
>>> re.split('x*', 'foo')
['foo'
```
syntax. \g<number> uses the corresponding group number; \g<2> is therefore equivalent to \2, but isn’t ambiguous in a replacement such as \g<2>0. \2 would be interpreted as a reference to group 20, not a reference to group 2 followed by the literal character ’0’. The backreference \g<0> substitutes in the entire substring matched by the RE. Changed in version 3.1: Added the optional flags argument.

re.subn (pattern, repl, string, count=0, flags=0)
Perform the same operation as sub(), but return a tuple (new_string, number_of_substitutions). Changed in version 3.1: Added the optional flags argument.

re.escape (string)
Escape all the characters in pattern except ASCII letters, numbers and ‘_’ . This is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it. Changed in version 3.3: The ‘_’ character is no longer escaped.

re.purge ()
Clear the regular expression cache.

exception re.error
Exception raised when a string passed to one of the functions here is not a valid regular expression (for example, it might contain unmatched parentheses) or when some other error occurs during compilation or matching. It is never an error if a string contains no match for a pattern.

6.2.3 Regular Expression Objects

Compiled regular expression objects support the following methods and attributes:

regex.search (string[, pos[, endpos]]]
Scan through string looking for a location where this regular expression produces a match, and return a corresponding match object. Return None if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

The optional second parameter pos gives an index in the string where the search is to start; it defaults to 0. This is not completely equivalent to slicing the string: the ‘^’ pattern character matches at the real beginning of the string and at positions just after a newline, but not necessarily at the index where the search is to start.

The optional parameter endpos limits how far the string will be searched; it will be as if the string is endpos characters long, so only the characters from pos to endpos - 1 will be searched for a match. If endpos is less than pos, no match will be found; otherwise, if rx is a compiled regular expression object, rx.search(string, 0, 50) is equivalent to rx.search(string[:50], 0).

>>> pattern = re.compile("d")
>>> pattern.search("dog")
<_sre.SRE_Match object at ...>
>>> pattern.search("dog", 1)
# No match; search doesn’t include the "d"

regex.match (string[, pos[, endpos]]]
If zero or more characters at the beginning of string match this regular expression, return a corresponding match object. Return None if the string does not match the pattern; note that this is different from a zero-length match.

The optional pos and endpos parameters have the same meaning as for the search() method.

>>> pattern = re.compile("o")
>>> pattern.match("dog")
# No match as "o" is not at the start of "dog".
>>> pattern.match("dog", 1)
# Match as "o" is the 2nd character of "dog".
<_sre.SRE_Match object at ...>

If you want to locate a match anywhere in string, use search() instead (see also search() vs. match()).

regex.split (string, maxsplit=0)
Identical to the split() function, using the compiled pattern.
The Python Library Reference, Release 3.3.3

regex.findall(string[, pos[, endpos]])
Similar to the findall() function, using the compiled pattern, but also accepts optional pos and endpos parameters that limit the search region like for match().

regex.finditer(string[, pos[, endpos]])
Similar to the finditer() function, using the compiled pattern, but also accepts optional pos and endpos parameters that limit the search region like for match().

regex.sub(repl, string, count=0)
Identical to the sub() function, using the compiled pattern.

regex.subn(repl, string, count=0)
Identical to the subn() function, using the compiled pattern.

regex.flags
The regex matching flags. This is a combination of the flags given to compile(), any (?...) inline flags in the pattern, and implicit flags such as UNICODE if the pattern is a Unicode string.

regex.groups
The number of capturing groups in the pattern.

regex.groupindex
A dictionary mapping any symbolic group names defined by (?P<id>) to group numbers. The dictionary is empty if no symbolic groups were used in the pattern.

regex.pattern
The pattern string from which the RE object was compiled.

6.2.4 Match Objects

Match objects always have a boolean value of True. Since match() and search() return None when there is no match, you can test whether there was a match with a simple if statement:

```python
match = re.search(pattern, string)
if match:
    process(match)
```

Match objects support the following methods and attributes:

```python
match.expand(template)
```
Return the string obtained by doing backslash substitution on the template string template, as done by the sub() method. Escapes such as \n are converted to the appropriate characters, and numeric backreferences (\1, \2) and named backreferences (\g<1>, \g<name>) are replaced by the contents of the corresponding group.

```python
match.group([group1, ...])
```
Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, group1 defaults to zero (the whole match is returned). If a groupN argument is zero, the corresponding return value is the entire matching string; if it is in the inclusive range [1..99], it is the string matching the corresponding parenthesized group. If a group number is negative or larger than the number of groups defined in the pattern, an IndexError exception is raised. If a group is contained in a part of the pattern that did not match, the corresponding result is None. If a group is contained in a part of the pattern that matched multiple times, the last match is returned.

```python
>>> m = re.match(r"(\w+) \(\w+\)", "Isaac Newton, physicist")
>>> m.group(0)  # The entire match
'Isaac Newton'
>>> m.group(1)  # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)  # The second parenthesized subgroup.
'Newton'
```
>>> m.group(1, 2)  # Multiple arguments give us a tuple.
('Isaac', 'Newton')

If the regular expression uses the (?P<name>...) syntax, the groupN arguments may also be strings identifying groups by their group name. If a string argument is not used as a group name in the pattern, an IndexError exception is raised.

A moderately complicated example:

>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'

Named groups can also be referred to by their index:

>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'

If a group matches multiple times, only the last match is accessible:

>>> m = re.match(r"(\w+)\.(\w+)", "a1b2c3")  # Matches 3 times.
>>> m.group(1)  # Returns only the last match.
'c3'

match.groups (default=None)

Return a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern. The default argument is used for groups that did not participate in the match; it defaults to None.

For example:

>>> m = re.match(r"(\d+).\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')

If we make the decimal place and everything after it optional, not all groups might participate in the match. These groups will default to None unless the default argument is given:

>>> m = re.match(r"(\d+)\.(?\d+)?", "24")
>>> m.groups()  # Second group defaults to None.
('24', None)
>>> m.groups('0')  # Now, the second group defaults to '0'.
('24', '0')

match.groupdict (default=None)

Return a dictionary containing all the named subgroups of the match, keyed by the subgroup name. The default argument is used for groups that did not participate in the match; it defaults to None. For example:

>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}

match.start ([group ])

Chapter 6. Text Processing Services
match.end([group])

Return the indices of the start and end of the substring matched by group; group defaults to zero (meaning the whole matched substring). Return -1 if group exists but did not contribute to the match. For a match object m, and a group g that did contribute to the match, the substring matched by group g (equivalent to m.group(g)) is

m.string[m.start(g):m.end(g)]

Note that m.start(group) will equal m.end(group) if group matched a null string. For example, after m = re.search('b(c?)', 'cba'), m.start(0) is 1, m.end(0) is 2, m.start(1) and m.end(1) are both 2, and m.start(2) raises an IndexError exception.

An example that will remove remove_this from email addresses:

```python
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

match.span([group])

For a match m, return the 2-tuple (m.start(group), m.end(group)). Note that if group did not contribute to the match, this is (-1, -1). group defaults to zero, the entire match.

match.pos

The value of pos which was passed to the search() or match() method of a regex object. This is the index into the string at which the RE engine started looking for a match.

match.endpos

The value of endpos which was passed to the search() or match() method of a regex object. This is the index into the string beyond which the RE engine will not go.

match.lastindex

The integer index of the last matched capturing group, or None if no group was matched at all. For example, the expressions (a)b, ((a)(b)), and ((ab)) will have lastindex == 1 if applied to the string ‘ab’, while the expression (a)(b) will have lastindex == 2, if applied to the same string.

match.lastgroup

The name of the last matched capturing group, or None if the group didn’t have a name, or if no group was matched at all.

match.re

The regular expression object whose match() or search() method produced this match instance.

match.string

The string passed to match() or search().

### 6.2.5 Regular Expression Examples

#### Checking for a Pair

In this example, we’ll use the following helper function to display match objects a little more gracefully:

```python
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

Suppose you are writing a poker program where a player’s hand is represented as a 5-character string with each character representing a card, “a” for ace, “k” for king, “q” for queen, ”j” for jack, ”t” for 10, and “2” through “9” representing the card with that value.

To see if a given string is a valid hand, one could do the following:
>>> valid = re.compile(r'^[a2-9tjqk]{5}$')
>>> displaymatch(valid.match("akt5q")) # Valid.
"<Match: 'akt5q', groups=()>
>>> displaymatch(valid.match("akt5e")) # Invalid.
>>> displaymatch(valid.match("akt")) # Invalid.
"<Match: 'akt', groups=()>
>>> displaymatch(valid.match("727ak")) # Valid.
"<Match: '727ak', groups=()>

That last hand, "727ak", contained a pair, or two of the same valued cards. To match this with a regular expression, one could use backreferences as such:

```python
>>> pair = re.compile(r".*(.).*\1")
>>> displaymatch(pair.match("717ak")) # Pair of 7s.
"<Match: '717', groups=('7',)>
>>> displaymatch(pair.match("718ak")) # No pairs.
>>> displaymatch(pair.match("354aa")) # Pair of aces.
"<Match: '354aa', groups=('a',)>
```

To find out what card the pair consists of, one could use the `group()` method of the match object in the following manner:

```python
>>> pair.match("717ak").group(1)
'7'
```

# Error because re.match() returns None, which doesn’t have a group() method:

```python
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r".*(.).*\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'
```

```python
>>> pair.match("354aa").group(1)
'a'
```

### Simulating scanf()

Python does not currently have an equivalent to `scanf()`. Regular expressions are generally more powerful, though also more verbose, than `scanf()` format strings. The table below offers some more-or-less equivalent mappings between `scanf()` format tokens and regular expressions.

<table>
<thead>
<tr>
<th><code>scanf()</code> Token</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td>.</td>
</tr>
<tr>
<td>%s</td>
<td>.{5}</td>
</tr>
<tr>
<td>%d</td>
<td>[-+]?\d+</td>
</tr>
<tr>
<td>%e, %E, %f, %g</td>
<td>[-+]?(\d+(.\d*)?</td>
</tr>
<tr>
<td>%i</td>
<td>[-+]?0[0-7]+</td>
</tr>
<tr>
<td>%o</td>
<td>\S+</td>
</tr>
<tr>
<td>%u</td>
<td>\d+</td>
</tr>
<tr>
<td>%x, %X</td>
<td>[-+]?0[0-7]+</td>
</tr>
</tbody>
</table>

To extract the filename and numbers from a string like

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

you would use a `scanf()` format like

```python
%s - %d errors, %d warnings
```

The equivalent regular expression would be

```
(\S+) - (\d+) errors, (\d+) warnings
```
search() vs. match()

Python offers two different primitive operations based on regular expressions: \texttt{re.match()} checks for a match only at the beginning of the string, while \texttt{re.search()} checks for a match anywhere in the string (this is what Perl does by default).

For example:

```python
>>> re.match("c", "abcdef")    # No match
<_sre.SRE_Match object at ...>
>>> re.search("c", "abcdef")  # Match
<_sre.SRE_Match object at ...>
```

Regular expressions beginning with ‘\^’ can be used with \texttt{search()} to restrict the match at the beginning of the string:

```python
>>> re.match("c", "abcdef")    # No match
<_sre.SRE_Match object at ...>
>>> re.search("\^c", "abcdef") # No match
<_sre.SRE_Match object at ...>
>>> re.search("\^a", "abcdef") # Match
<_sre.SRE_Match object at ...>
```

Note however that in MULTILINE mode \texttt{match()} only matches at the beginning of the string, whereas using \texttt{search()} with a regular expression beginning with ‘\^’ will match at the beginning of each line.

```python
>>> re.match('X', 'A

B

X', re.MULTILINE)  # No match
<_sre.SRE_Match object at ...>
>>> re.search('^X', 'A

B

X', re.MULTILINE)  # Match
<_sre.SRE_Match object at ...>
```

Making a Phonebook

\texttt{split()} splits a string into a list delimited by the passed pattern. The method is invaluable for converting textual data into data structures that can be easily read and modified by Python as demonstrated in the following example that creates a phonebook.

First, here is the input. Normally it may come from a file, here we are using triple-quoted string syntax:

```python
>>> text = "Ross McFluff: 834.345.1254 155 Elm Street

... Ronald Heathmore: 892.345.3428 436 Finley Avenue

... Frank Burger: 925.541.7625 662 South Dogwood Way

... Heather Albrecht: 548.326.4584 919 Park Place"
```

The entries are separated by one or more newlines. Now we convert the string into a list with each nonempty line having its own entry:

```python
>>> entries = re.split("\n+", text)
```

Finally, split each entry into a list with first name, last name, telephone number, and address. We use the \texttt{maxsplit} parameter of \texttt{split()} because the address has spaces, our splitting pattern, in it:

```python
>>> [re.split("?: ", entry, 3) for entry in entries]
["Ross McFluff: 834.345.1254 155 Elm Street",
 'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
 'Frank Burger: 925.541.7625 662 South Dogwood Way',
 'Heather Albrecht: 548.326.4584 919 Park Place']
```

The \texttt{?:} pattern matches the colon after the last name, so that it does not occur in the result list. With a \texttt{maxsplit} of 4, we could separate the house number from the street name:

```python
[re.split("?: ", entry, 3) for entry in entries]
["Ross", 'McFluff', '834.345.1254', '155 Elm Street'],
["Ronald", 'Heathmore', '892.345.3428', '436 Finley Avenue'],
["Frank", 'Burger', '925.541.7625', '662 South Dogwood Way'],
["Heather", 'Albrecht', '548.326.4584', '919 Park Place']
```
Text Munging

sub() replaces every occurrence of a pattern with a string or the result of a function. This example demonstrates using sub() with a function to “munge” text, or randomize the order of all the characters in each word of a sentence except for the first and last characters:

```python
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + ''.join(inner_word) + m.group(3)

>>> text = "Professor Abdolmalek, please report your absences promptly."
>>> re.sub(r"(\w)(\w+)(\w)\s+\w+\s+\w+", repl, text)
'Poefsroser Aealmobdk, palasee reoprt yuor asnebces plmrpyo."
>>> re.sub(r"(\w)(\w+)(\w)\s+\w+\s+\w+", repl, text)
'Pofsroser Aodlambelk, plasee reoprt your asnebces potlmrpyo."
```

Finding all Adverbs

findall() matches all occurrences of a pattern, not just the first one as search() does. For example, if one was a writer and wanted to find all of the adverbs in some text, he or she might use findall() in the following manner:

```python
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly", text)
['carefully', 'quickly']
```

Finding all Adverbs and their Positions

If one wants more information about all matches of a pattern than the matched text, finditer() is useful as it provides match objects instead of strings. Continuing with the previous example, if one was a writer who wanted to find all of the adverbs and their positions in some text, he or she would use finditer() in the following manner:

```python
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly", text):
...     print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
07-16: carefully
40-47: quickly
```

Raw String Notation

Raw string notation (r:"text") keeps regular expressions sane. Without it, every backslash (\) in a regular expression would have to be prefixed with another one to escape it. For example, the two following lines of code are functionally identical:

```python
>>> re.match(r"\W(.)\1\W", " ff ")
<_sre.SRE_Match object at ...>
>>> re.match("\W(.)\1\W", " ff ")
<_sre.SRE_Match object at ...>
```
When one wants to match a literal backslash, it must be escaped in the regular expression. With raw string notation, this means `r"\\"`. Without raw string notation, one must use "\\\\", making the following lines of code functionally identical:

```python
g>>> re.match(r"\\", r"\\")
<_sre.SRE_Match object at ...>
g>>> re.match("\\\\", r"\\")
<_sre.SRE_Match object at ...>
```

**Writing a Tokenizer**

A tokenizer or scanner analyzes a string to categorize groups of characters. This is a useful first step in writing a compiler or interpreter.

The text categories are specified with regular expressions. The technique is to combine those into a single master regular expression and to loop over successive matches:

```python
import collections
import re

Token = collections.namedtuple('Token', ['typ', 'value', 'line', 'column'])
def tokenize(s):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}
token_specification = [
    ('NUMBER', r'[dD][.dD]*'),  # Integer or decimal number
    ('ASSIGN', r'(:=)'),      # Assignment operator
    ('END', r';'),           # Statement terminator
    ('ID', r'\[A-Za-z\]+'), # Identifiers
    ('OP', r'\[+ */-\]'),  # Arithmetic operators
    ('NEWLINE', r'\n'),      # Line endings
    ('SKIP', r'\[ \t\]'),   # Skip over spaces and tabs
]
tok_regex = |'.join('(?P<%s>%s)' % pair for pair in token_specification)
get_token = re.compile(tok_regex).match
line = 1
pos = line_start = 0
mo = get_token(s)
while mo is not None:
    typ = mo.lastgroup
    if typ == 'NEWLINE':
        line_start = pos
        line += 1
    elif typ == 'ID' and typ in keywords:
        typ = val
    yield Token(typ, val, line, mo.start()-line_start)
    pos = mo.end()
    mo = get_token(s)
if pos != len(s):
    raise RuntimeError('Unexpected character %r on line %d' % (s[pos], line))
```

```
6.2. re — Regular expression operations 97
```

```
statements = '''
IF quantity THEN
    total := total + price * quantity;
    tax := price * 0.05;
ENDIF;
'''
```
for token in tokenize(statements):
    print(token)

The tokenizer produces the following output:

Token(typ='IF', value='IF', line=2, column=5)
Token(typ='ID', value='quantity', line=2, column=8)
Token(typ='THEN', value='THEN', line=2, column=17)
Token(typ='ID', value='total', line=3, column=9)
Token(typ='ASSIGN', value=':=', line=3, column=15)
Token(typ='ID', value='total', line=3, column=18)
Token(typ='OP', value='+', line=3, column=24)
Token(typ='ID', value='price', line=3, column=26)
Token(typ='OP', value='*', line=3, column=32)
Token(typ='ID', value='quantity', line=3, column=34)
Token(typ='ID', value='price', line=3, column=34)
Token(typ='OP', value='*', line=4, column=22)
Token(typ='NUMBER', value='0.05', line=4, column=24)
Token(typ='END', value=';', line=4, column=28)
Token(typ='ENDIF', value='ENDIF', line=5, column=5)
Token(typ='END', value=';', line=5, column=10)

6.3 difflib — Helpers for computing deltas

This module provides classes and functions for comparing sequences. It can be used for example, for comparing files, and can produce difference information in various formats, including HTML and context and unified diffs. For comparing directories and files, see also, the filecmp module.

class difflib.SequenceMatcher
    This is a flexible class for comparing pairs of sequences of any type, so long as the sequence elements are hashable. The basic algorithm predates, and is a little fancier than, an algorithm published in the late 1980’s by Ratcliff and Obershelp under the hyperbolic name “gestalt pattern matching.” The idea is to find the longest contiguous matching subsequence that contains no “junk” elements (the Ratcliff and Obershelp algorithm doesn’t address junk). The same idea is then applied recursively to the pieces of the sequences to the left and to the right of the matching subsequence. This does not yield minimal edit sequences, but does tend to yield matches that “look right” to people.

    Timing: The basic Ratcliff-Obershelp algorithm is cubic time in the worst case and quadratic time in the expected case. SequenceMatcher is quadratic time for the worst case and has expected-case behavior dependent on a complicated way on how many elements the sequences have in common; best case time is linear.

    Automatic junk heuristic: SequenceMatcher supports a heuristic that automatically treats certain sequence items as junk. The heuristic counts how many times each individual item appears in the sequence. If an item’s duplicates (after the first one) account for more than 1% of the sequence and the sequence is at least 200 items long, this item is marked as “popular” and is treated as junk for the purpose of sequence matching. This heuristic can be turned off by setting the autojunk argument to False when creating the SequenceMatcher. New in version 3.2: The autojunk parameter.

class difflib.Differ
    This is a class for comparing sequences of lines of text, and producing human-readable differences or deltas. Differ uses SequenceMatcher both to compare sequences of lines, and to compare sequences of characters within similar (near-matching) lines.

    Each line of a Differ delta begins with a two-letter code:
### The Python Library Reference, Release 3.3.3

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-'</td>
<td>line unique to sequence 1</td>
</tr>
<tr>
<td>'+'</td>
<td>line unique to sequence 2</td>
</tr>
<tr>
<td>','</td>
<td>line common to both sequences</td>
</tr>
<tr>
<td>'?'</td>
<td>line not present in either input sequence</td>
</tr>
</tbody>
</table>

Lines beginning with '?' attempt to guide the eye to intraline differences, and were not present in either input sequence. These lines can be confusing if the sequences contain tab characters.

**class `difflib.HtmlDiff`**

This class can be used to create an HTML table (or a complete HTML file containing the table) showing a side by side, line by line comparison of text with inter-line and intra-line change highlights. The table can be generated in either full or contextual difference mode.

The constructor for this class is:

```python
__init__(
tabsize=8, wrapcolumn=None, linejunk=None, charjunk=IS_CHARACTER_JUNK)
```

Initializes instance of `HtmlDiff`.

- `tabsize` is an optional keyword argument to specify tab stop spacing and defaults to 8.
- `wrapcolumn` is an optional keyword to specify column number where lines are broken and wrapped, defaults to `None` where lines are not wrapped.
- `linejunk` and `charjunk` are optional keyword arguments passed into `ndiff()` (used by `HtmlDiff` to generate the side by side HTML differences). See `ndiff()` documentation for argument default values and descriptions.

The following methods are public:

- **`make_file(fromlines, tolines, fromdesc='', todesc='', context=False, numlines=5)`**
  Compares `fromlines` and `tolines` (lists of strings) and returns a string which is a complete HTML file containing a table showing line by line differences with inter-line and intra-line changes highlighted.
  - `fromdesc` and `todesc` are optional keyword arguments to specify from/to file column header strings (both default to an empty string).
  - `context` and `numlines` are both optional keyword arguments. Set `context` to `True` when contextual differences are to be shown, else the default is `False` to show the full files. `numlines` defaults to 5. When `context` is `True` `numlines` controls the number of context lines which surround the difference highlights. When `context` is `False` `numlines` controls the number of lines which are shown before a difference highlight when using the “next” hyperlinks (setting to zero would cause the “next” hyperlinks to place the next difference highlight at the top of the browser without any leading context).

- **`make_table(fromlines, tolines, fromdesc='', todesc='', context=False, numlines=5)`**
  Compares `fromlines` and `tolines` (lists of strings) and returns a string which is a complete HTML table showing line by line differences with inter-line and intra-line changes highlighted.
  - `fromdesc` and `todesc` are optional keyword arguments to specify from/to file column header strings (both default to an empty string).
  - `context` and `numlines` are both optional keyword arguments. Set `context` to `True` when contextual differences are to be shown, else the default is `False` to show the full files. `numlines` defaults to 5. When `context` is `True` `numlines` controls the number of context lines which surround the difference highlights. When `context` is `False` `numlines` controls the number of lines which are shown before a difference highlight when using the “next” hyperlinks (setting to zero would cause the “next” hyperlinks to place the next difference highlight at the top of the browser without any leading context).

**Tools/scripts/diff.py** is a command-line front-end to this class and contains a good example of its use.

**`difflib.context_diff(a, b, fromfile='', tofile='', fromfiledate='', tofiledate='', n=3, lineterm='')`**

Compare `a` and `b` (lists of strings); return a delta (a generator generating the delta lines) in context diff format.

Context diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in a before/after style. The number of context lines is set by `n` which defaults to three.

By default, the diff control lines (those with `***` or `---`) are created with a trailing newline. This is helpful so that inputs created from `io.IOBase.readlines()` result in diffs that are suitable for use with `io.IOBase.writelines()` since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the `lineterm` argument to "" so that the output will be uniformly newline free.

### 6.3. `difflib` — Helpers for computing deltas

99
The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for fromfile, tofile, fromfiledate, and tofiledate. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```python
>>> s1 = ['bacon
', 'eggs
', 'ham
', 'guido
']
>>> s2 = ['python
', 'eggy
', 'hamster
', 'guido
']
>>> for line in context_diff(s1, s2, fromfile='before.py', tofile='after.py'):
...    sys.stdout.write(line)
```

The best (no more than n) matches among the possibilities are returned in a list, sorted by similarity score, most similar first.

```python
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('apple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for fromfile, tofile, fromfiledate, and tofiledate. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```python
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('apple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```
Tools/scripts/ndiff.py is a command-line front-end to this function.

```python
>>> diff = ndiff('one
two
three'.splitlines(1),
... 'ore
tree
emu'.splitlines(1))
>>> print(''.join(diff), end='')
- one
? ^
+ ore
? ^
- two
- three
? -
+ tree
+ emu
```

difflib.restore(sequence, which)

Return one of the two sequences that generated a delta.

Given a sequence produced by Differ.compare() or ndiff(), extract lines originating from file 1 or 2 (parameter which), stripping off line prefixes.

Example:

```python
>>> diff = ndiff('one
two
three'.splitlines(1),
... 'ore
tree
emu'.splitlines(1))
>>> diff = list(diff)  # materialize the generated delta into a list
>>> print(''.join(restore(diff, 1)), end='')
one
two
three
>>> print(''.join(restore(diff, 2)), end='')
ore
tree
emu
```

difflib.unified_diff(a, b, fromfile='', tofile='', fromfiledate='', tofiledate='', n=3, lineterm='
')

Compare a and b (lists of strings); return a delta (a generator generating the delta lines) in unified diff format.

Unified diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in a inline style (instead of separate before/after blocks). The number of context lines is set by n which defaults to three.

By default, the diff control lines (those with ---, +++, or @@) are created with a trailing newline. This is helpful so that inputs created from io.IOBase.readlines() result in diffs that are suitable for use with io.IOBase.writelines() since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the lineterm argument to "" so that the output will be uniformly newline free.

The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for fromfile, tofile, fromfiledate, and tofiledate. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```python
>>> s1 = ['bacon
', 'eggs
', 'ham
', 'guido
']
>>> s2 = ['python
', 'eggy
', 'hamster
', 'guido
']
>>> for line in unified_diff(s1, s2, fromfile='before.py', tofile='after.py'):
...     sys.stdout.write(line)
--- before.py
+++ after.py
```
@@ -1,4 +1,4 @@
-bacon
-eggs
-ham
+python
+eggy
+hamster
-guido

See `A command-line interface to difflib` for a more detailed example.

difflib.IS_LINE_JUNK (line)

Return true for ignorable lines. The line `line` is ignorable if `line` is blank or contains a single ‘#’, otherwise it is not ignorable. Used as a default for parameter `linejunk` in `ndiff()` in older versions.

difflib.IS_CHARACTER_JUNK (ch)

Return true for ignorable characters. The character `ch` is ignorable if `ch` is a space or tab, otherwise it is not ignorable. Used as a default for parameter `charjunk` in `ndiff()`.

See Also:

Pattern Matching: The Gestalt Approach  Discussion of a similar algorithm by John W. Ratcliff and D. E. Metzener. This was published in Dr. Dobb’s Journal in July, 1988.

6.3.1 SequenceMatcher Objects

The `SequenceMatcher` class has this constructor:

class difflib.SequenceMatcher (isjunk=None, a='', b='', autojunk=True)

Optional argument `isjunk` must be `None` (the default) or a one-argument function that takes a sequence element and returns true if and only if the element is “junk” and should be ignored. Passing `None` for `isjunk` is equivalent to passing `lambda x: 0`; in other words, no elements are ignored. For example, pass:

```
lambda x: x in " \t"
```

if you’re comparing lines as sequences of characters, and don’t want to synch up on blanks or hard tabs.

The optional arguments `a` and `b` are sequences to be compared; both default to empty strings. The elements of both sequences must be `hashable`.

The optional argument `autojunk` can be used to disable the automatic junk heuristic. New in version 3.2: The `autojunk` parameter. `SequenceMatcher` objects get three data attributes: `bjunk` is the set of elements of `b` for which `isjunk` is `True`; `bpopular` is the set of non-junk elements considered popular by the heuristic (if it is not disabled); `b2j` is a dict mapping the remaining elements of `b` to a list of positions where they occur. All three are reset whenever `b` is reset with `set_seqs()` or `set_seq2()`.

New in version 3.2: The `bjunk` and `bpopular` attributes. `SequenceMatcher` objects have the following methods:

```
set_seqs (a, b)
```

Set the two sequences to be compared.

```
set_seq1 (a)
```

Set the first sequence to be compared. The second sequence to be compared is not changed.

```
set_seq2 (b)
```

Set the second sequence to be compared. The first sequence to be compared is not changed.

```
find_longest_match (alo, ahi, blo, bhi)
```

Find longest matching block in `a[alo:ahi]` and `b[blo:bhi]`.
If `isjunk` was omitted or `None`, `find_longest_match()` returns `(i, j, k)` such that `a[i:i+k]` is equal to `b[j:j+k]`, where `alo <= i <= i+k <= ahi` and `blo <= j <= j+k <= bhi`. For all `(i', j', k')` meeting those conditions, the additional conditions `k >= k'`, `i <= i'`, and if `i == i'`, `j <= j'` are also met. In other words, of all maximal matching blocks, return one that starts earliest in `a`, and of all those maximal matching blocks that start earliest in `a`, return the one that starts earliest in `b`.

```python
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

If `isjunk` was provided, first the longest matching block is determined as above, but with the additional restriction that no junk element appears in the block. Then that block is extended as far as possible by matching (only) junk elements on both sides. So the resulting block never matches on junk except as identical junk happens to be adjacent to an interesting match.

Here’s the same example as before, but considering blanks to be junk. That prevents ‘`abcd’ from matching the ‘`abcd’ at the tail end of the second sequence directly. Instead only the ‘`abcd’ can match, and matches the leftmost ‘`abcd’ in the second sequence:

```python
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

If no blocks match, this returns `(alo, blo, 0)`.

This method returns a named tuple `Match(a, b, size)`.

get_matching_blocks()  
Return list of triples describing matching subsequences. Each triple is of the form `(i, j, n)`, and means that `a[i:i+n] == b[j:j+n]`. The triples are monotonically increasing in `i` and `j`.

The last triple is a dummy, and has the value `(len(a), len(b), 0)`. It is the only triple with `n == 0`. If `(i, j, n)` and `(i', j', n')` are adjacent triples in the list, and the second is not the last triple in the list, then `i+n != i'` or `j+n != j'`; in other words, adjacent triples always describe non-adjacent equal blocks.

```python
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2), Match(a=5, b=4, size=0)]
```

get_opcodes()  
Return list of 5-tuples describing how to turn `a` into `b`. Each tuple is of the form `(tag, i1, i2, j1, j2)`. The first tuple has `i1 == j1 == 0`, and remaining tuples have `i1` equal to the `i2` from the preceding tuple, and, likewise, `j1` equal to the previous `j2`.

The `tag` values are strings, with these meanings:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>replace</code></td>
<td><code>a[i1:i2]</code> should be replaced by <code>b[j1:j2]</code>.</td>
</tr>
<tr>
<td><code>delete</code></td>
<td><code>a[i1:i2]</code> should be deleted. Note that <code>j1 == j2</code> in this case.</td>
</tr>
<tr>
<td><code>insert</code></td>
<td><code>b[j1:j2]</code> should be inserted at <code>a[i1:i1]</code>. Note that <code>i1 == i2</code> in this case.</td>
</tr>
<tr>
<td><code>equal</code></td>
<td><code>a[i1:i2]</code> == <code>b[j1:j2]</code> (the sub-sequences are equal).</td>
</tr>
</tbody>
</table>

For example:

```python
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...     print('{:7} a[{}:{}|---> b[{}:{}])'.format('!', tag, i1, i2, j1, j2))
```

6.3. `difflib` — Helpers for computing deltas

103

**get_grouped_opcodes** *(n=3)*

Return a *generator* of groups with up to *n* lines of context.

Starting with the groups returned by *get_opcodes()*, this method splits out smaller change clusters and eliminates intervening ranges which have no changes.

The groups are returned in the same format as *get_opcodes()*. 

**ratio()**

Return a measure of the sequences’ similarity as a float in the range [0, 1].

Where *T* is the total number of elements in both sequences, and *M* is the number of matches, this is \(2.0 * M / T\). Note that this is 1.0 if the sequences are identical, and 0.0 if they have nothing in common.

This is expensive to compute if *get_matching_blocks()* or *get_opcodes()* hasn’t already been called, in which case you may want to try *quick_ratio()* or *real_quick_ratio()* first to get an upper bound.

**quick_ratio()**

Return an upper bound on *ratio()* relatively quickly.

**real_quick_ratio()**

Return an upper bound on *ratio()* very quickly.

The three methods that return the ratio of matching to total characters can give different results due to differing levels of approximation, although *quick_ratio()* and *real_quick_ratio()* are always at least as large as *ratio()*:

```python
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

### 6.3.2 SequenceMatcher Examples

This example compares two strings, considering blanks to be “junk”:

```python
>>> s = SequenceMatcher(lambda x: x == " ", ...
...   "private Thread currentThread;", ...
...   "private volatile Thread currentThread;")
```

*ratio()* returns a float in [0, 1], measuring the similarity of the sequences. As a rule of thumb, a *ratio()* value over 0.6 means the sequences are close matches:

```python
>>> print(round(s.ratio(), 3))
0.866
```

If you’re only interested in where the sequences match, *get_matching_blocks()* is handy:

```python
>>> for block in s.get_matching_blocks():
...   print("a[\%d] and b[\%d] match for \%d elements" % block)
```

Note that the last tuple returned by *get_matching_blocks()* is always a dummy, (len(a), len(b), 0), and this is the only case in which the last tuple element (number of elements matched) is 0.

If you want to know how to change the first sequence into the second, use *get_opcodes()*:
>>> for opcode in s.get_opcodes():
...     print("%6s a[%d:%d] b[%d:%d]" % opcode)
    equal a[0:8] b[0:8]
    insert a[8:8] b[8:17]
    equal a[8:29] b[17:38]

See Also:

- The `get_close_matches()` function in this module which shows how simple code building on `SequenceMatcher` can be used to do useful work.
- Simple version control recipe for a small application built with `SequenceMatcher`.

### 6.3.3 Differ Objects

Note that `Differ`-generated deltas make no claim to be **minimal** diffs. To the contrary, minimal diffs are often counter-intuitive, because they synch up anywhere possible, sometimes accidental matches 100 pages apart. Restricting synch points to contiguous matches preserves some notion of locality, at the occasional cost of producing a longer diff.

The `Differ` class has this constructor:

```python
class difflib.Differ(linejunk=None, charjunk=None)
```

Optional keyword parameters `linejunk` and `charjunk` are for filter functions (or `None`):

- `linejunk`: A function that accepts a single string argument, and returns true if the string is junk. The default is `None`, meaning that no line is considered junk.
- `charjunk`: A function that accepts a single character argument (a string of length 1), and returns true if the character is junk. The default is `None`, meaning that no character is considered junk.

`Differ` objects are used (deltas generated) via a single method:

```python
compare(a, b)
```

Compare two sequences of lines, and generate the delta (a sequence of lines).

Each sequence must contain individual single-line strings ending with newlines. Such sequences can be obtained from the `readlines()` method of file-like objects. The delta generated also consists of newline-terminated strings, ready to be printed as-is via the `writelines()` method of a file-like object.

### 6.3.4 Differ Example

This example compares two texts. First we set up the texts, sequences of individual single-line strings ending with newlines (such sequences can also be obtained from the `readlines()` method of file-like objects):

```python
>>> text1 = '''  1. Beautiful is better than ugly.
...     2. Explicit is better than implicit.
...     3. Simple is better than complex.
...     4. Complex is better than complicated.
... '''.splitlines(1)
```

Next we instantiate a `Differ` object:

```python
Differ()
```
>>> d = Differ()

Note that when instantiating a `Differ` object we may pass functions to filter out line and character “junk.” See the `Differ()` constructor for details.

Finally, we compare the two:

```python
>>> result = list(d.compare(text1, text2))
```

result is a list of strings, so let’s pretty-print it:

```python
>>> from pprint import pprint
>>> pprint(result)
[' 1. Beautiful is better than ugly.
', '- 2. Explicit is better than implicit.
', '+ 3. Simple is better than complex.
', '?  ++
', '- 4. Complex is better than complicated.
', '?  ^ ---- ^
', '+ 4. Complicated is better than complex.
', '?  +++ ^ ^
', '+ 5. Flat is better than nested.
']
```

As a single multi-line string it looks like this:

```python
>>> import sys
>>> sys.stdout.writelines(result)
1. Beautiful is better than ugly.
- 2. Explicit is better than implicit.
- 3. Simple is better than complex.
+ 3. Simple is better than complex.
?  ++
- 4. Complex is better than complicated.
?  ^ ---- ^
+ 4. Complicated is better than complex.
?  +++ ^ ^
+ 5. Flat is better than nested.
```

### 6.3.5 A command-line interface to difflib

This example shows how to use difflib to create a `diff`-like utility. It is also contained in the Python source distribution, as `Tools/scripts/diff.py`.

```python
""" Command line interface to difflib.py providing diffs in four formats:

* ndiff: lists every line and highlights interline changes.
* context: highlights clusters of changes in a before/after format.
* unified: highlights clusters of changes in an inline format.
* html: generates side by side comparison with change highlights.

"""

```python
import sys, os, time, difflib, optparse

def main():
    # Configure the option parser
    usage = "usage: %prog [options] fromfile tofile"
    parser = optparse.OptionParser(usage)
    parser.add_option("-c", action="store_true", default=False, help='Produce a context format diff (default)')
```
parser.add_option("-u", action="store_true", default=False, help='Produce a unified format diff')

hlp = 'Produce HTML side by side diff (can use -c and -l in conjunction)'
parser.add_option("-m", action="store_true", default=False, help=hlp)
parser.add_option("-n", action="store_true", default=False, help='Produce a ndiff format diff')
parser.add_option("-l", "--lines", type="int", default=3, help='Set number of context lines (default 3)')

(options, args) = parser.parse_args()

if len(args) == 0:
    parser.print_help()
    sys.exit(1)

if len(args) != 2:
    parser.error("need to specify both a fromfile and tofile")

n = options.lines
fromfile, tofile = args # as specified in the usage string

# we're passing these as arguments to the diff function
fromdate = time.ctime(os.stat(fromfile).st_mtime)
todate = time.ctime(os.stat(tofile).st_mtime)
with open(fromfile) as fromf, open(tofile) as tof:
    fromlines, tolines = list(fromf), list(tof)

if options.u:
    diff = difflib.unified_diff(fromlines, tolines, fromfile, tofile, fromdate, todate, n=n)
elif options.n:
    diff = difflib.ndiff(fromlines, tolines)
elif options.m:
    diff = difflib.HtmlDiff().make_file(fromlines, tolines, fromfile, tofile, context=options.c, numlines=n)
else:
    diff = difflib.context_diff(fromlines, tolines, fromfile, tofile, fromdate, todate, n=n)

# we're using writelines because diff is a generator
sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()

6.4 textwrap — Text wrapping and filling

Source code: Lib/textwrap.py

The textwrap module provides two convenience functions, wrap() and fill(), as well as TextWrapper, the class that does all the work, and two utility functions, dedent() and indent(). If you’re just wrapping or filling one or two text strings, the convenience functions should be good enough; otherwise, you should use an instance of TextWrapper for efficiency.

textwrap.wrap(text, width=70, **kwargs)

Wraps the single paragraph in text (a string) so every line is at most width characters long. Returns a list of output lines, without final newlines.
Optional keyword arguments correspond to the instance attributes of `TextWrapper`, documented below. `width` defaults to 70.

See the `TextWrapper.wrap()` method for additional details on how `wrap()` behaves.

`textwrap.fill(text, width=70, **kwargs)`

Wraps the single paragraph in `text`, and returns a single string containing the wrapped paragraph. `fill()` is shorthand for

```
"\n".join(wrap(text, ...))
```

In particular, `fill()` accepts exactly the same keyword arguments as `wrap()`.

Both `wrap()` and `fill()` work by creating a `TextWrapper` instance and calling a single method on it. That instance is not reused, so for applications that wrap/fill many text strings, it will be more efficient for you to create your own `TextWrapper` object.

Text is preferably wrapped on whitespaces and right after the hyphens in hyphenated words; only then will long words be broken if necessary, unless `TextWrapper.break_long_words` is set to false.

Two additional utility function, `dedent()` and `indent()`, are provided to remove indentation from strings that have unwanted whitespace to the left of the text and to add an arbitrary prefix to selected lines in a block of text.

`textwrap.dedent(text)`

Remove any common leading whitespace from every line in `text`.

This can be used to make triple-quoted strings line up with the left edge of the display, while still presenting them in the source code in indented form.

Note that tabs and spaces are both treated as whitespace, but they are not equal: the lines " hello" and "\thello" are considered to have no common leading whitespace.

For example:

```python
def test():
    # end first line with \ to avoid the empty line!
    s = '',''
    hello
    world
    
    print(repr(s))    # prints ' hello
    print(repr(dedent(s))) # prints 'hello
```

`textwrap.indent(text, prefix, predicate=None)`

Add `prefix` to the beginning of selected lines in `text`.

Lines are separated by calling `text.splitlines(True)`.

By default, `prefix` is added to all lines that do not consist solely of whitespace (including any line endings).

For example:

```python
>>> s = 'hello

 \nworld'
>>> indent(s, ' ')
' hello\n
 \n world'
```

The optional `predicate` argument can be used to control which lines are indented. For example, it is easy to add `prefix` to even empty and whitespace-only lines:

```python
>>> print(indent(s, '+ ', lambda line: True))
+ hello
+ + world
```
class `textwrap.TextWrapper`(**kwargs)

The `TextWrapper` constructor accepts a number of optional keyword arguments. Each keyword argument corresponds to an instance attribute, so for example

```python
wrapper = TextWrapper(initial_indent="* ")
```

is the same as

```python
wrapper = TextWrapper()
wrapper.initial_indent = "* 
```

You can re-use the same `TextWrapper` object many times, and you can change any of its options through direct assignment to instance attributes between uses.

The `TextWrapper` instance attributes (and keyword arguments to the constructor) are as follows:

- **width**
  (default: 70) The maximum length of wrapped lines. As long as there are no individual words in the input text longer than `width`, `TextWrapper` guarantees that no output line will be longer than `width` characters.

- **expand_tabs**
  (default: True) If true, then all tab characters in `text` will be expanded to spaces using the `expandtabs()` method of `text`.

- **tabsize**
  (default: 8) If `expand_tabs` is true, then all tab characters in `text` will be expanded to zero or more spaces, depending on the current column and the given tab size. New in version 3.3.

- **replace_whitespace**
  (default: True) If true, after tab expansion but before wrapping, the `wrap()` method will replace each whitespace character with a single space. The whitespace characters replaced are as follows: tab, newline, vertical tab, formfeed, and carriage return (`'	
\v\f\r`').

  **Note:** If `expand_tabs` is false and `replace_whitespace` is true, each tab character will be replaced by a single space, which is not the same as tab expansion.

  **Note:** If `replace_whitespace` is false, newlines may appear in the middle of a line and cause strange output. For this reason, text should be split into paragraphs (using `str.splitlines()` or similar) which are wrapped separately.

- **drop_whitespace**
  (default: True) If true, whitespace at the beginning and ending of every line (after wrapping but before indenting) is dropped. Whitespace at the beginning of the paragraph, however, is not dropped if non-whitespace follows it. If whitespace being dropped takes up an entire line, the whole line is dropped.

- **initial_indent**
  (default: "") String that will be prepended to the first line of wrapped output. Counts towards the length of the first line. The empty string is not indented.

- **subsequent_indent**
  (default: "") String that will be prepended to all lines of wrapped output except the first. Counts towards the length of each line except the first.

- **fix_sentence_endings**
  (default: False) If true, `TextWrapper` attempts to detect sentence endings and ensure that sentences are always separated by exactly two spaces. This is generally desired for text in a monospaced font. However, the sentence detection algorithm is imperfect: it assumes that a sentence ending consists of a lowercase letter followed by one of `.`, `.`, `'`, or `?`, possibly followed by one of `'"`
or ",", followed by a space. One problem with this algorithm is that it is unable to detect the
difference between “Dr.” in

[...] Dr. Frankenstein’s monster [...]  

and “Spot.” in

[...] See Spot. See Spot run [...]  

fix_sentence_endings is false by default.
Since the sentence detection algorithm relies on string.lowercase for the definition of “lower-
case letter,” and a convention of using two spaces after a period to separate sentences on the same line,
it is specific to English-language texts.

break_long_words
(default: True) If true, then words longer than width will be broken in order to ensure that no lines
are longer than width. If it is false, long words will not be broken, and some lines may be longer
than width. (Long words will be put on a line by themselves, in order to minimize the amount by
which width is exceeded.)

break_on_hyphens
(default: True) If true, wrapping will occur preferably on whitespaces and right after hyphens in
compound words, as it is customary in English. If false, only whitespaces will be considered as
potentially good places for line breaks, but you need to set break_long_words to false if you
want truly inseparable words. Default behaviour in previous versions was to always allow breaking
hyphenated words.

TextWrapper also provides two public methods, analogous to the module-level convenience functions:

wrap(text)
Wraps the single paragraph in text (a string) so every line is at most width characters long. All
wrapping options are taken from instance attributes of the TextWrapper instance. Returns a list of
output lines, without final newlines. If the wrapped output has no content, the returned list is empty.

fill(text)
Wraps the single paragraph in text, and returns a single string containing the wrapped paragraph.

6.5 unicodedata — Unicode Database

This module provides access to the Unicode Character Database (UCD) which defines character properties for all
Unicode characters. The data contained in this database is compiled from the UCD version 6.1.0.

The module uses the same names and symbols as defined by Unicode Standard Annex #44, “Unicode Character
Database”. It defines the following functions:

unicodedata.lookup(name)
Look up character by name. If a character with the given name is found, return the corresponding character.
If not found, KeyError is raised. Changed in version 3.3: Support for name aliases \(^1\) and named sequences \(^2\) has been added.

unicodedata.name(chr[, default])
Returns the name assigned to the character chr as a string. If no name is defined, default is returned, or, if
not given, ValueError is raised.

unicodedata.decimal(chr[, default])
Returns the decimal value assigned to the character chr as integer. If no such value is defined, default is
returned, or, if not given, ValueError is raised.

\(^1\) http://www.unicode.org/Public/6.1.0/ucd/NameAliases.txt
\(^2\) http://www.unicode.org/Public/6.1.0/ucd/NamedSequences.txt
The Python Library Reference, Release 3.3.3

unicodedata.
digit(chr[, default])
Returns the digit value assigned to the character chr as integer. If no such value is defined, default is returned, or, if not given, ValueError is raised.

unicodedata.
numeric(chr[, default])
Returns the numeric value assigned to the character chr as float. If no such value is defined, default is returned, or, if not given, ValueError is raised.

unicodedata.
category(chr)
Returns the general category assigned to the character chr as string.

unicodedata.
bidirectional(chr)
Returns the bidirectional class assigned to the character chr as string. If no such value is defined, an empty string is returned.

unicodedata.
combining(chr)
Returns the canonical combining class assigned to the character chr as integer. Returns 0 if no combining class is defined.

unicodedata.
east_asian_width(chr)
Returns the east asian width assigned to the character chr as string.

unicodedata.
mirrored(chr)
Returns the mirrored property assigned to the character chr as integer. Returns 1 if the character has been identified as a “mirrored” character in bidirectional text, 0 otherwise.

unicodedata.
decomposition(chr)
Returns the character decomposition mapping assigned to the character chr as string. An empty string is returned in case no such mapping is defined.

unicodedata.
normalize(form, unistr)
Return the normal form form for the Unicode string unistr. Valid values for form are ‘NFC’, ‘NFKC’, ‘NFD’, and ‘NFKD’.

The Unicode standard defines various normalization forms of a Unicode string, based on the definition of canonical equivalence and compatibility equivalence. In Unicode, several characters can be expressed in various way. For example, the character U+00C7 (LATIN CAPITAL LETTER C WITH CEDILLA) can also be expressed as the sequence U+0043 (LATIN CAPITAL LETTER C) U+0327 (COMBINING CEDILLA).

For each character, there are two normal forms: normal form C and normal form D. Normal form D (NFD) is also known as canonical decomposition, and translates each character into its decomposed form. Normal form C (NFC) first applies a canonical decomposition, then composes pre-combined characters again.

In addition to these two forms, there are two additional normal forms based on compatibility equivalence. In Unicode, certain characters are supported which normally would be unified with other characters. For example, U+2160 (ROMAN NUMERAL ONE) is really the same thing as U+0049 (LATIN CAPITAL LETTER I). However, it is supported in Unicode for compatibility with existing character sets (e.g. gb2312).

The normal form KD (NFKD) will apply the compatibility decomposition, i.e. replace all compatibility characters with their equivalents. The normal form KC (NFKC) first applies the compatibility decomposition, followed by the canonical composition.

Even if two unicode strings are normalized and look the same to a human reader, if one has combining characters and the other doesn’t, they may not compare equal.

In addition, the module exposes the following constant:

unicodedata.
unidata_version
The version of the Unicode database used in this module.

unicodedata.
ucd_3_2_0
This is an object that has the same methods as the entire module, but uses the Unicode database version 3.2 instead, for applications that require this specific version of the Unicode database (such as IDNA).

Examples:
The Python Library Reference, Release 3.3.3

>>> import unicodedata
>>> unicodedata.lookup(’LEFT CURLY BRACKET’)
’{’
>>> unicodedata.name(’/’)
’SOLIDUS’
>>> unicodedata.decimal(’9’)
9
>>> unicodedata.decimal(’a’)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
ValueError: not a decimal
>>> unicodedata.category(’A’) # ’L’etter, ’u’ppercase
’Lu’
>>> unicodedata.bidirectional(’\u0660’) # ’A’rabic, ’N’umber
’AN’

6.6 stringprep — Internet String Preparation
When identifying things (such as host names) in the internet, it is often necessary to compare such identifications
for “equality”. Exactly how this comparison is executed may depend on the application domain, e.g. whether it
should be case-insensitive or not. It may be also necessary to restrict the possible identifications, to allow only
identifications consisting of “printable” characters.
RFC 3454 defines a procedure for “preparing” Unicode strings in internet protocols. Before passing strings onto
the wire, they are processed with the preparation procedure, after which they have a certain normalized form. The
RFC defines a set of tables, which can be combined into profiles. Each profile must define which tables it uses, and
what other optional parts of the stringprep procedure are part of the profile. One example of a stringprep
profile is nameprep, which is used for internationalized domain names.
The module stringprep only exposes the tables from RFC 3454. As these tables would be very large to
represent them as dictionaries or lists, the module uses the Unicode character database internally. The module
source code itself was generated using the mkstringprep.py utility.
As a result, these tables are exposed as functions, not as data structures. There are two kinds of tables in the RFC:
sets and mappings. For a set, stringprep provides the “characteristic function”, i.e. a function that returns true
if the parameter is part of the set. For mappings, it provides the mapping function: given the key, it returns the
associated value. Below is a list of all functions available in the module.
stringprep.in_table_a1(code)
Determine whether code is in tableA.1 (Unassigned code points in Unicode 3.2).
stringprep.in_table_b1(code)
Determine whether code is in tableB.1 (Commonly mapped to nothing).
stringprep.map_table_b2(code)
Return the mapped value for code according to tableB.2 (Mapping for case-folding used with NFKC).
stringprep.map_table_b3(code)
Return the mapped value for code according to tableB.3 (Mapping for case-folding used with no normalization).
stringprep.in_table_c11(code)
Determine whether code is in tableC.1.1 (ASCII space characters).
stringprep.in_table_c12(code)
Determine whether code is in tableC.1.2 (Non-ASCII space characters).
stringprep.in_table_c11_c12(code)
Determine whether code is in tableC.1 (Space characters, union of C.1.1 and C.1.2).
stringprep.in_table_c21(code)
Determine whether code is in tableC.2.1 (ASCII control characters).
112

Chapter 6. Text Processing Services


stringprep.in_table_c22(code)
    Determine whether code is in tableC.2.2 (Non-ASCII control characters).

stringprep.in_table_c21_c22(code)
    Determine whether code is in tableC.2 (Control characters, union of C.2.1 and C.2.2).

stringprep.in_table_c3(code)
    Determine whether code is in tableC.3 (Private use).

stringprep.in_table_c4(code)
    Determine whether code is in tableC.4 (Non-character code points).

stringprep.in_table_c5(code)
    Determine whether code is in tableC.5 (Surrogate codes).

stringprep.in_table_c6(code)
    Determine whether code is in tableC.6 (Inappropriate for plain text).

stringprep.in_table_c7(code)
    Determine whether code is in tableC.7 (Inappropriate for canonical representation).

stringprep.in_table_c8(code)
    Determine whether code is in tableC.8 (Change display properties or are deprecated).

stringprep.in_table_c9(code)
    Determine whether code is in tableC.9 (Tagging characters).

stringprep.in_table_d1(code)
    Determine whether code is in tableD.1 (Characters with bidirectional property “R” or “AL”).

stringprep.in_table_d2(code)
    Determine whether code is in tableD.2 (Characters with bidirectional property “L”).

6.7 readline — GNU readline interface

Platforms: Unix

The readline module defines a number of functions to facilitate completion and reading/writing of history files from the Python interpreter. This module can be used directly or via the rlcompleter module. Settings made using this module affect the behaviour of both the interpreter’s interactive prompt and the prompts offered by the built-in input() function.

Note: On MacOS X the readline module can be implemented using the libedit library instead of GNU readline.

The configuration file for libedit is different from that of GNU readline. If you programmatically load configuration strings you can check for the text “libedit” in readline.__doc__ to differentiate between GNU readline and libedit.

The readline module defines the following functions:

readline.parse_and_bind(string)
    Parse and execute single line of a readline init file.

readline.get_line_buffer()
    Return the current contents of the line buffer.

readline.insert_text(string)
    Insert text into the command line.

readline.read_init_file(filename)
    Parse a readline initialization file. The default filename is the last filename used.

6.7. readline — GNU readline interface 113
readline.read_history_file([filename])
    Load a readline history file. The default filename is ~/.history.

readline.write_history_file([filename])
    Save a readline history file. The default filename is ~/.history.

readline.clear_history()
    Clear the current history. (Note: this function is not available if the installed version of GNU readline
    doesn’t support it.)

readline.get_history_length()
    Return the desired length of the history file. Negative values imply unlimited history file size.

readline.set_history_length(length)
    Set the number of lines to save in the history file. write_history_file() uses this value to truncate
    the history file when saving. Negative values imply unlimited history file size.

readline.get_current_history_length()
    Return the number of lines currently in the history. (This is different from get_history_length(),
    which returns the maximum number of lines that will be written to a history file.)

readline.get_history_item(index)
    Return the current contents of history item at index.

readline.remove_history_item(pos)
    Remove history item specified by its position from the history.

readline.replace_history_item(pos, line)
    Replace history item specified by its position with the given line.

readline.redisplay()
    Change what’s displayed on the screen to reflect the current contents of the line buffer.

readline.set_startup_hook([function])
    Set or remove the startup_hook function. If function is specified, it will be used as the new startup_hook
    function; if omitted or None, any hook function already installed is removed. The startup_hook function is
    called with no arguments just before readline prints the first prompt.

readline.set_pre_input_hook([function])
    Set or remove the pre_input_hook function. If function is specified, it will be used as the new
    pre_input_hook function; if omitted or None, any hook function already installed is removed. The
    pre_input_hook function is called with no arguments after the first prompt has been printed and just before
    readline starts reading input characters.

readline.set_completer([function])
    Set or remove the completer function. If function is specified, it will be used as the new completer function;
    if omitted or None, any completer function already installed is removed. The completer function is called
    as function(text, state), for state in 0, 1, 2, ..., until it returns a non-string value. It should return
    the next possible completion starting with text.

readline.get_completer()
    Get the completer function, or None if no completer function has been set.

readline.get_completion_type()
    Get the type of completion being attempted.

readline.get_begidx()
    Get the beginning index of the readline tab-completion scope.

readline.get_endidx()
    Get the ending index of the readline tab-completion scope.

readline.set_completer_delims(string)
    Set the readline word delimiters for tab-completion.

readline.get_completer_delims()
    Get the readline word delimiters for tab-completion.
readline.set_completion_display_matches_hook([function])

Set or remove the completion display function. If function is specified, it will be used as the new completion display function; if omitted or None, any completion display function already installed is removed. The completion display function is called as function(substitution, [matches], longest_match_length) once each time matches need to be displayed.

readline.add_history(line)

Append a line to the history buffer, as if it was the last line typed.

See Also:
Module rlcompleter Completion of Python identifiers at the interactive prompt.

6.7.1 Example

The following example demonstrates how to use the readline module’s history reading and writing functions to automatically load and save a history file named .pyhist from the user’s home directory. The code below would normally be executed automatically during interactive sessions from the user’s PYTHONSTARTUP file.

```python
import os
import readline
histfile = os.path.join(os.path.expanduser("~"), ".pyhist")
try:
    readline.read_history_file(histfile)
except FileNotFoundError:
    pass
import atexit
atexit.register(readline.write_history_file, histfile)
del os, histfile
```

The following example extends the code.InteractiveConsole class to support history save/restore.

```python
import os
import readline
import atexit
import os

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/.console-history")):
        code.InteractiveConsole.__init__(self, locals, filename)
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except FileNotFoundError:
                pass
        atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
        readline.write_history_file(histfile)
```

6.7. readline — GNU readline interface
6.8 rlcompleter — Completion function for GNU readline

Source code: Lib/rlcompleter.py

The rlcompleter module defines a completion function suitable for the readline module by completing valid Python identifiers and keywords.

When this module is imported on a Unix platform with the readline module available, an instance of the Completer class is automatically created and its complete() method is set as the readline completer.

Example:
```python
>>> import rlcompleter
>>> import readline

>>> readline.parse_and_bind("tab: complete")

>>> readline.<TAB PRESSED>
```

The rlcompleter module is designed for use with Python’s interactive mode. A user can add the following lines to his or her initialization file (identified by the PYTHONSTARTUP environment variable) to get automatic Tab completion:
```
try:
    import readline
except ImportError:
    print("Module readline not available.")
else:
    import rlcompleter
    readline.parse_and_bind("tab: complete")
```

On platforms without readline, the Completer class defined by this module can still be used for custom purposes.

6.8.1 Completer Objects

Completer objects have the following method:

Completer.complete(text, state)

Return the state th completion for text.

If called for text that doesn’t include a period character (‘.’), it will complete from names currently defined in __main__, builtins and keywords (as defined by the keyword module).

If called for a dotted name, it will try to evaluate anything without obvious side-effects (functions will not be evaluated, but it can generate calls to __getattr__() up to the last part, and find matches for the rest via the dir() function. Any exception raised during the evaluation of the expression is caught, silenced and None is returned.
The modules described in this chapter provide some basic services operations for manipulation of binary data. Other operations on binary data, specifically in relation to file formats and network protocols, are described in the relevant sections.

Some libraries described under *Text Processing Services* also work with either ASCII-compatible binary formats (for example, `re`) or all binary data (for example, `difflib`).

In addition, see the documentation for Python’s built-in binary data types in *Binary Sequence Types — bytes, bytearray, memoryview*.

### 7.1 struct — Interpret bytes as packed binary data

This module performs conversions between Python values and C structs represented as Python `bytes` objects. This can be used in handling binary data stored in files or from network connections, among other sources. It uses *Format Strings* as compact descriptions of the layout of the C structs and the intended conversion to/from Python values.

**Note:** By default, the result of packing a given C struct includes pad bytes in order to maintain proper alignment for the C types involved; similarly, alignment is taken into account when unpacking. This behavior is chosen so that the bytes of a packed struct correspond exactly to the layout in memory of the corresponding C struct. To handle platform-independent data formats or omit implicit pad bytes, use `standard` size and alignment instead of `native` size and alignment: see *Byte Order, Size, and Alignment* for details.

#### 7.1.1 Functions and Exceptions

The module defines the following exception and functions:

**exception struct.error**

Exception raised on various occasions; argument is a string describing what is wrong.

**struct.pack(fmt, v1, v2, ...)**

Return a bytes object containing the values `v1`, `v2`, ... packed according to the format string `fmt`. The arguments must match the values required by the format exactly.

**struct.pack_into(fmt, buffer, offset, v1, v2, ...)**

Pack the values `v1`, `v2`, ... according to the format string `fmt` and write the packed bytes into the writable buffer `buffer` starting at position `offset`. Note that `offset` is a required argument.

**struct.unpack(fmt, buffer)**

Unpack from the buffer `buffer` (presumably packed by `struct.pack(fmt, ...)`) according to the format string `fmt`. The result is a tuple even if it contains exactly one item. The buffer must contain exactly the amount of data required by the format (`len(buffer)` must equal `struct.calcsize(fmt)`).
struct.unpack_from(fmt, buffer, offset=0)

Unpack from buffer starting at position offset, according to the format string fmt. The result is a tuple
even if it contains exactly one item. buffer must contain at least the amount of data required by the format
(len(buffer[offset:]) must be at least calcsize(fmt)).

struct.calcsize(fmt)

Return the size of the struct (and hence of the bytes object produced by pack(fmt, ...)) corresponding
to the format string fmt.

7.1.2 Format Strings

Format strings are the mechanism used to specify the expected layout when packing and unpacking data. They
are built up from Format Characters, which specify the type of data being packed/unpacked. In addition, there are
special characters for controlling the Byte Order, Size, and Alignment.

Byte Order, Size, and Alignment

By default, C types are represented in the machine’s native format and byte order, and properly aligned by skipping
pad bytes if necessary (according to the rules used by the C compiler).

Alternatively, the first character of the format string can be used to indicate the byte order, size and alignment of
the packed data, according to the following table:

<table>
<thead>
<tr>
<th>Character</th>
<th>Byte order</th>
<th>Size</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>native</td>
<td>native</td>
<td>native</td>
</tr>
<tr>
<td>=</td>
<td>native</td>
<td>standard</td>
<td>none</td>
</tr>
<tr>
<td>&lt;</td>
<td>little-endian</td>
<td>standard</td>
<td>none</td>
</tr>
<tr>
<td>&gt;</td>
<td>big-endian</td>
<td>standard</td>
<td>none</td>
</tr>
<tr>
<td>!</td>
<td>network (= big-endian)</td>
<td>standard</td>
<td>none</td>
</tr>
</tbody>
</table>

If the first character is not one of these, ‘@’ is assumed.

Native byte order is big-endian or little-endian, depending on the host system. For example, Intel x86 and AMD64
(x86-64) are little-endian; Motorola 68000 and PowerPC G5 are big-endian; ARM and Intel Itanium feature
switchable endianness (bi-endian). Use sys.byteorder to check the endianness of your system.

Native size and alignment are determined using the C compiler’s sizeof expression. This is always combined
with native byte order.

Standard size depends only on the format character; see the table in the Format Characters section.

Note the difference between ‘@’ and ‘=’: both use native byte order, but the size and alignment of the latter is
standardized.

The form ‘!’ is available for those poor souls who claim they can’t remember whether network byte order is
big-endian or little-endian.

There is no way to indicate non-native byte order (force byte-swapping); use the appropriate choice of ‘<’ or
’>’.

Notes:

1. Padding is only automatically added between successive structure members. No padding is added at the
beginning or the end of the encoded struct.

2. No padding is added when using non-native size and alignment, e.g. with ‘<’, ‘>’, ‘=’, and ‘!’.

3. To align the end of a structure to the alignment requirement of a particular type, end the format with the
code for that type with a repeat count of zero. See Examples.
Format Characters

Format characters have the following meaning; the conversion between C and Python values should be obvious given their types. The 'Standard size' column refers to the size of the packed value in bytes when using standard size; that is, when the format string starts with one of '<', '>', '!' or '='. When using native size, the size of the packed value is platform-dependent.

<table>
<thead>
<tr>
<th>Format</th>
<th>C Type</th>
<th>Python type</th>
<th>Standard size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>pad byte</td>
<td>no value</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>char</td>
<td>bytes of length 1</td>
<td>1</td>
<td>(1),(3)</td>
</tr>
<tr>
<td>b</td>
<td>signed char</td>
<td>integer</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>unsigned char</td>
<td>integer</td>
<td>1</td>
<td>(3)</td>
</tr>
<tr>
<td>?</td>
<td>_Bool</td>
<td>bool</td>
<td>1</td>
<td>(1)</td>
</tr>
<tr>
<td>h</td>
<td>short</td>
<td>integer</td>
<td>2</td>
<td>(3)</td>
</tr>
<tr>
<td>H</td>
<td>unsigned short</td>
<td>integer</td>
<td>2</td>
<td>(3)</td>
</tr>
<tr>
<td>i</td>
<td>int</td>
<td>integer</td>
<td>4</td>
<td>(3)</td>
</tr>
<tr>
<td>I</td>
<td>unsigned int</td>
<td>integer</td>
<td>4</td>
<td>(3)</td>
</tr>
<tr>
<td>l</td>
<td>long</td>
<td>integer</td>
<td>4</td>
<td>(3)</td>
</tr>
<tr>
<td>L</td>
<td>unsigned long</td>
<td>integer</td>
<td>8</td>
<td>(2), (3)</td>
</tr>
<tr>
<td>q</td>
<td>long long</td>
<td>integer</td>
<td>8</td>
<td>(2), (3)</td>
</tr>
<tr>
<td>Q</td>
<td>unsigned long long</td>
<td>integer</td>
<td>8</td>
<td>(2), (3)</td>
</tr>
<tr>
<td>n</td>
<td>ssize_t</td>
<td>integer</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>size_t</td>
<td>integer</td>
<td>4</td>
<td>(4)</td>
</tr>
<tr>
<td>f</td>
<td>float</td>
<td>float</td>
<td>4</td>
<td>(5)</td>
</tr>
<tr>
<td>d</td>
<td>double</td>
<td>float</td>
<td>8</td>
<td>(5)</td>
</tr>
<tr>
<td>s</td>
<td>char[]</td>
<td>bytes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>char[]</td>
<td>bytes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>void +</td>
<td>integer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Changed in version 3.3: Added support for the 'n' and 'N' formats. Notes:

1. The '?' conversion code corresponds to the _Bool type defined by C99. If this type is not available, it is simulated using a char. In standard mode, it is always represented by one byte.

2. The 'q' and 'Q' conversion codes are available in native mode only if the platform C compiler supports C long long, or, on Windows, __int64. They are always available in standard modes.

3. When attempting to pack a non-integer using any of the integer conversion codes, if the non-integer has a __index__() method then that method is called to convert the argument to an integer before packing. Changed in version 3.2: Use of the __index__() method for non-integers is new in 3.2.

4. The 'n' and 'N' conversion codes are only available for the native size (selected as the default or with the '@' byte order character). For the standard size, you can use whichever of the other integer formats fits your application.

5. For the 'f' and 'd' conversion codes, the packed representation uses the IEEE 754 binary32 (for 'f') or binary64 (for 'd') format, regardless of the floating-point format used by the platform.

6. The 'P' format character is only available for the native byte ordering (selected as the default or with the '@' byte order character). The byte order character '=' chooses to use little- or big-endian ordering based on the host system. The struct module does not interpret this as native ordering, so the 'P' format is not available.

A format character may be preceded by an integral repeat count. For example, the format string '4h' means exactly the same as 'hhhh'.

Whitespace characters between formats are ignored; a count and its format must not contain whitespace though.

For the 's' format character, the count is interpreted as the length of the bytes, not a repeat count like for the other format characters; for example, '10s' means a single 10-byte string, while '10c' means 10 characters. If a count is not given, it defaults to 1. For packing, the string is truncated or padded with null bytes as appropriate to make it fit. For unpacking, the resulting bytes object always has exactly the specified number of bytes. As a special case, '0s' means a single, empty string (while '0c' means 0 characters).
When packing a value \( x \) using one of the integer formats (‘b’, ‘B’, ‘h’, ‘H’, ‘i’, ‘I’, ‘l’, ‘L’, ‘q’, ‘Q’), if \( x \) is outside the valid range for that format then \texttt{struct.error} is raised. Changed in version 3.1: In 3.0, some of the integer formats wrapped out-of-range values and raised \texttt{DeprecationWarning} instead of \texttt{struct.error}. The ‘p’ format character encodes a “Pascal string”, meaning a short variable-length string stored in a \textit{fixed number of bytes}, given by the count. The first byte stored is the length of the string, or 255, whichever is smaller. The bytes of the string follow. If the string passed in to \texttt{pack()} is too long (longer than the count minus 1), only the leading \( \text{count-1} \) bytes of the string are stored. If the string is shorter than \( \text{count-1} \), it is padded with null bytes so that exactly count bytes in all are used. Note that for \texttt{unpack()}, the ‘p’ format character consumes count bytes, but that the string returned can never contain more than 255 bytes.

For the ‘?’ format character, the return value is either \texttt{True} or \texttt{False}. When packing, the truth value of the argument object is used. Either 0 or 1 in the native or standard bool representation will be packed, and any non-zero value will be True when unpacking.

**Examples**

**Note:** All examples assume a native byte order, size, and alignment with a big-endian machine.

A basic example of packing/unpacking three integers:

```python
>>> from struct import *
>>> pack('hhl', 1, 2, 3)
\x00\x01\x00\x02\x00\x00\x00\x03
>>> unpack('hhl', b'\x00\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('hhl')
8
```

Unpacked fields can be named by assigning them to variables or by wrapping the result in a named tuple:

```python
>>> record = b'raymond \x32\x12\x08\x01\x08'
>>> name, serialnum, school, gradelevel = unpack('<10sHHb', record)
>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name serialnum school gradelevel')
>>> Student._make(unpack('<10sHHb', record))
Student(name=b'raymond ', serialnum=4658, school=264, gradelevel=8)
```

The ordering of format characters may have an impact on size since the padding needed to satisfy alignment requirements is different:

```python
>>> pack('ci', b'*', 0x12131415)
b'*\x00\x00\x00\x12\x13\x14\x15'
>>> pack('ic', 0x12131415, b'*')
b'\x12\x13\x14\x15'*
>>> calcsize('ci')
8
>>> calcsize('ic')
5
```

The following format ‘llh0l’ specifies two pad bytes at the end, assuming longs are aligned on 4-byte boundaries:

```python
>>> pack('llh0l', 1, 2, 3)
b'\x00\x00\x00\x00\x01\x00\x00\x00\x15\x13\x14\x15'
```

This only works when native size and alignment are in effect; standard size and alignment does not enforce any alignment.

**See Also:**

*Module array* Packed binary storage of homogeneous data.
Module `xdrlib` Packing and unpacking of XDR data.

### 7.1.3 Classes

The `struct` module also defines the following type:

```python
class struct.Struct(format)
```

Return a new Struct object which writes and reads binary data according to the format string `format`. Creating a Struct object once and calling its methods is more efficient than calling the `struct` functions with the same format since the format string only needs to be compiled once.

Compiled Struct objects support the following methods and attributes:

- `pack(v1, v2, ...)`
  Identical to the `pack()` function, using the compiled format. (`len(result)` will equal `self.size`).

- `pack_into(buffer, offset, v1, v2, ...)`
  Identical to the `pack_into()` function, using the compiled format.

- `unpack(buffer)`
  Identical to the `unpack()` function, using the compiled format. (`len(buffer)` must equal `self.size`).

- `unpack_from(buffer, offset=0)`
  Identical to the `unpack_from()` function, using the compiled format. (`len(buffer[offset:])` must be at least `self.size`).

- `format`
  The format string used to construct this Struct object.

- `size`
  The calculated size of the struct (and hence of the bytes object produced by the `pack()` method) corresponding to `format`.

### 7.2 `codecs` — Codec registry and base classes

This module defines base classes for standard Python codecs (encoders and decoders) and provides access to the internal Python codec registry which manages the codec and error handling lookup process.

It defines the following functions:

- `codecs.encode(obj, encoding='utf-8', errors='strict')`
  Encodes `obj` using the codec registered for `encoding`. Errors may be given to set the desired error handling scheme. The default error handler is `strict` meaning that encoding errors raise `ValueError` (or a more codec specific subclass, such as `UnicodeEncodeError`). Refer to `Codec Base Classes` for more information on codec error handling.

- `codecs.decode(obj, encoding='utf-8', errors='strict')`
  Decodes `obj` using the codec registered for `encoding`. Errors may be given to set the desired error handling scheme. The default error handler is `strict` meaning that decoding errors raise `ValueError` (or a more codec specific subclass, such as `UnicodeDecodeError`). Refer to `Codec Base Classes` for more information on codec error handling.

- `codecs.register(search_function)`
  Register a codec search function. Search functions are expected to take one argument, the encoding name in all lower case letters, and return a `CodecInfo` object having the following attributes:

  - `name` The name of the encoding;
  - `encode` The stateless encoding function;
• decode The stateless decoding function;
• incrementalencoder An incremental encoder class or factory function;
• incrementaldecoder An incremental decoder class or factory function;
• streamwriter A stream writer class or factory function;
• streamreader A stream reader class or factory function.

The various functions or classes take the following arguments:

encode and decode: These must be functions or methods which have the same interface as the encode() / decode() methods of Codec instances (see Codec Interface). The functions/methods are expected to work in a stateless mode.

incrementalencoder and incrementaldecoder: These have to be factory functions providing the following interface:

    factory(errors='strict')

The factory functions must return objects providing the interfaces defined by the base classes IncrementalEncoder and IncrementalDecoder, respectively. Incremental codecs can maintain state.

streamreader and streamwriter: These have to be factory functions providing the following interface:

    factory(stream, errors='strict')

The factory functions must return objects providing the interfaces defined by the base classes StreamReader and StreamWriter, respectively. Stream codecs can maintain state.

Possible values for errors are

• ‘strict’: raise an exception in case of an encoding error
• ‘replace’: replace malformed data with a suitable replacement marker, such as ‘?’ or ‘\ufffd’
• ‘ignore’: ignore malformed data and continue without further notice
• ‘xmlcharrefreplace’: replace with the appropriate XML character reference (for encoding only)
• ‘backslashreplace’: replace with backslashed escape sequences (for encoding only)
• ‘surrogateescape’: on decoding, replace with code points in the Unicode Private Use Area ranging from U+DC80 to U+DCFF. These private code points will then be turned back into the same bytes when the surrogateescape error handler is used when encoding the data. (See PEP 383 for more.)

as well as any other error handling name defined via register_error().

In case a search function cannot find a given encoding, it should return None.

codecs.lookup(encoding)

Looks up the codec info in the Python codec registry and returns a CodecInfo object as defined above.

Encodings are first looked up in the registry’s cache. If not found, the list of registered search functions is scanned. If no CodecInfo object is found, a LookupError is raised. Otherwise, the CodecInfo object is stored in the cache and returned to the caller.

To simplify access to the various codecs, the module provides these additional functions which use lookup() for the codec lookup:

codecs.getencoder(encoding)

Look up the codec for the given encoding and return its encoder function.

Raising a LookupError in case the encoding cannot be found.
codecs.getdecoder(encoding)
    Look up the codec for the given encoding and return its decoder function.
    Raises a LookupError in case the encoding cannot be found.

codecs.getincrementalencoder(encoding)
    Look up the codec for the given encoding and return its incremental encoder class or factory function.
    Raises a LookupError in case the encoding cannot be found or the codec doesn’t support an incremental encoder.

codecs.getincrementaldecoder(encoding)
    Look up the codec for the given encoding and return its incremental decoder class or factory function.
    Raises a LookupError in case the encoding cannot be found or the codec doesn’t support an incremental decoder.

codecs.getreader(encoding)
    Look up the codec for the given encoding and return its StreamReader class or factory function.
    Raises a LookupError in case the encoding cannot be found.

codecs.getwriter(encoding)
    Look up the codec for the given encoding and return its StreamWriter class or factory function.
    Raises a LookupError in case the encoding cannot be found.

codecs.register_error(name, error_handler)
    Register the error handling function error_handler under the name name. error_handler will be called
during encoding and decoding in case of an error, when name is specified as the errors parameter.

    For encoding error_handler will be called with a UnicodeEncodeError instance, which contains in-
formation about the location of the error. The error handler must either raise this or a different exception or
return a tuple with a replacement for the unencodable part of the input and a position where encoding should
continue. The replacement may be either str or bytes. If the replacement is bytes, the encoder will sim-
ply copy them into the output buffer. If the replacement is a string, the encoder will encode the replacement.
Encoding continues on original input at the specified position. Negative position values will be treated as
being relative to the end of the input string. If the resulting position is out of bound an IndexError will
be raised.

    Decoding and translating works similar, except UnicodeDecodeError or
UnicodeTranslateError will be passed to the handler and that the replacement from the error
handler will be put into the output directly.

codecs.lookup_error(name)
    Return the error handler previously registered under the name name.
    Raises a LookupError in case the handler cannot be found.

codecs.strict_errors(exception)
    Implements the strict error handling: each encoding or decoding error raises a UnicodeError.

codecs.replace_errors(exception)
    Implements the replace error handling: malformed data is replaced with a suitable replacement character
such as ‘?’ in bytestrings and ‘\ufffd’ in Unicode strings.

codecs.ignore_errors(exception)
    Implements the ignore error handling: malformed data is ignored and encoding or decoding is continued
without further notice.

codecs.xmlcharrefreplace_errors(exception)
    Implements the xmlcharrefreplace error handling (for encoding only): the unencodable character is
replaced by an appropriate XML character reference.

codecs.backslashreplace_errors(exception)
    Implements the backslashreplace error handling (for encoding only): the unencodable character is
replaced by a backslashed escape sequence.
To simplify working with encoded files or streams, the module also defines these utility functions:

```python
codecs.open(filename, mode[, encoding[, errors[, buffering ]]])
```

Open an encoded file using the given `mode` and return a wrapped version providing transparent encoding/decoding. The default file mode is `'r'` meaning to open the file in read mode.

**Note:** The wrapped version’s methods will accept and return strings only. Bytes arguments will be rejected.

**Note:** Files are always opened in binary mode, even if no binary mode was specified. This is done to avoid data loss due to encodings using 8-bit values. This means that no automatic conversion of `b'
'` is done on reading and writing.

- `encoding` specifies the encoding which is to be used for the file.
- `errors` may be given to define the error handling. It defaults to `'strict'` which causes a `ValueError` to be raised in case an encoding error occurs.
- `buffering` has the same meaning as for the built-in `open()` function. It defaults to line buffered.

```python
codecs.EncodedFile(file, data_encoding, file_encoding=None, errors='strict')
```

Return a wrapped version of file which provides transparent encoding translation.

Bytes written to the wrapped file are interpreted according to the given `data_encoding` and then written to the original file as bytes using the `file_encoding`.

If `file_encoding` is not given, it defaults to `data_encoding`.

- `errors` may be given to define the error handling. It defaults to `'strict'`, which causes `ValueError` to be raised in case an encoding error occurs.

```python
codecs.iterencode(iterator, encoding, errors='strict', **kwargs)
```

Uses an incremental encoder to iteratively encode the input provided by `iterator`. This function is a generator. `errors` (as well as any other keyword argument) is passed through to the incremental encoder.

```python
codecs.iterdecode(iterator, encoding, errors='strict', **kwargs)
```

Uses an incremental decoder to iteratively decode the input provided by `iterator`. This function is a generator. `errors` (as well as any other keyword argument) is passed through to the incremental decoder.

The module also provides the following constants which are useful for reading and writing to platform dependent files:

```python
codecs.BOM
endcodecs.BOM_BE
endcodecs.BOM_LE
endcodecs.BOM_UTF8
endcodecs.BOM_UTF16
endcodecs.BOM_UTF16_BE
endcodecs.BOM_UTF16_LE
endcodecs.BOM_UTF32
endcodecs.BOM_UTF32_BE
endcodecs.BOM_UTF32_LE
```

These constants define various encodings of the Unicode byte order mark (BOM) used in UTF-16 and UTF-32 data streams to indicate the byte order used in the stream or file and in UTF-8 as a Unicode signature. `BOM_UTF16` is either `BOM_UTF16_BE` or `BOM_UTF16_LE` depending on the platform’s native byte order. `BOM` is an alias for `BOM_UTF16`. `BOM_BE` for `BOM_UTF16_BE` and `BOM_LE` for `BOM_UTF16_LE`. The others represent the BOM in UTF-8 and UTF-32 encodings.

### 7.2.1 Codec Base Classes

The `codecs` module defines a set of base classes which define the interface and can also be used to easily write your own codecs for use in Python.
Each codec has to define four interfaces to make it usable as codec in Python: stateless encoder, stateless decoder, stream reader and stream writer. The stream reader and writers typically reuse the stateless encoder/decoder to implement the file protocols.

The Codec class defines the interface for stateless encoders/decoders.

To simplify and standardize error handling, the `encode()` and `decode()` methods may implement different error handling schemes by providing the `errors` string argument. The following string values are defined and implemented by all standard Python codecs:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>'strict'</code></td>
<td>Raise <code>UnicodeError</code> (or a subclass); this is the default.</td>
</tr>
<tr>
<td><code>'ignore'</code></td>
<td>Ignore the character and continue with the next.</td>
</tr>
<tr>
<td><code>'replace'</code></td>
<td>Replace with a suitable replacement character; Python will use the official U+FFFD REPLACEMENT CHARACTER for the built-in Unicode codecs on decoding and ‘?’ on encoding.</td>
</tr>
<tr>
<td><code>'xmlcharrefreplace'</code></td>
<td>Replace with the appropriate XML character reference (only for encoding).</td>
</tr>
<tr>
<td><code>'backslashreplace'</code></td>
<td>Replace with backslashed escape sequences (only for encoding).</td>
</tr>
<tr>
<td><code>'surrogateescape'</code></td>
<td>Replace byte with surrogate U+DCxx, as defined in PEP 383.</td>
</tr>
</tbody>
</table>

In addition, the following error handlers are specific to a single codec:

<table>
<thead>
<tr>
<th>Value</th>
<th>Codec</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>'surrogatepass'</code></td>
<td>utf-8</td>
<td>Allow encoding and decoding of surrogate codes in UTF-8.</td>
</tr>
</tbody>
</table>

New in version 3.1: The `'surrogateescape'` and `'surrogatepass'` error handlers. The set of allowed values can be extended via `register_error()`.

**Codec Objects**

The Codec class defines these methods which also define the function interfaces of the stateless encoder and decoder:

```python
codec.encode(input[, errors])
```

Encodes the object `input` and returns a tuple (output object, length consumed). Encoding converts a string object to a bytes object using a particular character set encoding (e.g., `cp1252` or `iso-8859-1`).

`errors` defines the error handling to apply. It defaults to `'strict'` handling.

The method may not store state in the Codec instance. Use `StreamCodec` for codecs which have to keep state in order to make encoding/decoding efficient.

The encoder must be able to handle zero length input and return an empty object of the output object type in this situation.

```python
codec.decode(input[, errors])
```

Decodes the object `input` and returns a tuple (output object, length consumed). Decoding converts a bytes object encoded using a particular character set encoding to a string object.

`input` must be a bytes object or one which provides the read-only character buffer interface – for example, buffer objects and memory mapped files.

`errors` defines the error handling to apply. It defaults to `'strict'` handling.

The method may not store state in the Codec instance. Use `StreamCodec` for codecs which have to keep state in order to make encoding/decoding efficient.

The decoder must be able to handle zero length input and return an empty object of the output object type in this situation.

The `IncrementalEncoder` and `IncrementalDecoder` classes provide the basic interface for incremental encoding and decoding. Encoding/decoding the input isn’t done with one call to the stateless encoder/decoder function, but with multiple calls to the `encode()`/`decode()` method of the incremental encoder/decoder. The incremental encoder/decoder keeps track of the encoding/decoding process during method calls.
The joined output of calls to the `encode()`/`decode()` method is the same as if all the single inputs were joined into one, and this input was encoded/decoded with the stateless encoder/decoder.

**IncrementalEncoder Objects**

The `IncrementalEncoder` class is used for encoding an input in multiple steps. It defines the following methods which every incremental encoder must define in order to be compatible with the Python codec registry.

```python
class codecs.IncrementalEncoder([errors])
    Constructor for an IncrementalEncoder instance.
```

All incremental encoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The `IncrementalEncoder` may implement different error handling schemes by providing the `errors` keyword argument. These parameters are predefined:

- `'strict'` Raise `ValueError` (or a subclass); this is the default.
- `'ignore'` Ignore the character and continue with the next.
- `'replace'` Replace with a suitable replacement character
- `'xmlcharrefreplace'` Replace with the appropriate XML character reference
- `'backslashreplace'` Replace with backslashed escape sequences.

The `errors` argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `IncrementalEncoder` object.

The set of allowed values for the `errors` argument can be extended with `register_error()`.

```python
encode([object[, final]])
    Encodes object (taking the current state of the encoder into account) and returns the resulting encoded object. If this is the last call to `encode()` `final` must be true (the default is false).
```

```python
reset()
    Reset the encoder to the initial state. The output is discarded: call `.encode('', final=True)` to reset the encoder and to get the output.
```

```python
IncrementalEncoder.getstate()
    Return the current state of the encoder which must be an integer. The implementation should make sure that 0 is the most common state. (States that are more complicated than integers can be converted into an integer by marshaling/pickling the state and encoding the bytes of the resulting string into an integer).
```

```python
IncrementalEncoder.setstate(state)
    Set the state of the encoder to state. state must be an encoder state returned by `getstate()`.
```

**IncrementalDecoder Objects**

The `IncrementalDecoder` class is used for decoding an input in multiple steps. It defines the following methods which every incremental decoder must define in order to be compatible with the Python codec registry.

```python
class codecs.IncrementalDecoder([errors])
    Constructor for an IncrementalDecoder instance.
```

All incremental decoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The `IncrementalDecoder` may implement different error handling schemes by providing the `errors` keyword argument. These parameters are predefined:

- `'strict'` Raise `ValueError` (or a subclass); this is the default.
- `'ignore'` Ignore the character and continue with the next.
• ‘replace’ Replace with a suitable replacement character.

The errors argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the IncrementalDecoder object.

The set of allowed values for the errors argument can be extended with register_error().

\[
\text{decode (object[, final])}
\]

Decodes object (taking the current state of the decoder into account) and returns the resulting decoded object. If this is the last call to decode() final must be true (the default is false). If final is true the decoder must decode the input completely and must flush all buffers. If this isn’t possible (e.g. because of incomplete byte sequences at the end of the input) it must initiate error handling just like in the stateless case (which might raise an exception).

\[
\text{reset ()}
\]

Reset the decoder to the initial state.

\[
\text{getstate ()}
\]

Return the current state of the decoder. This must be a tuple with two items, the first must be the buffer containing the still undecoded input. The second must be an integer and can be additional state info. (The implementation should make sure that 0 is the most common additional state info.) If this additional state info is 0 it must be possible to set the decoder to the state which has no input buffered and 0 as the additional state info, so that feeding the previously buffered input to the decoder returns it to the previous state without producing any output. (Additional state info that is more complicated than integers can be converted into an integer by marshaling/pickling the info and encoding the bytes of the resulting string into an integer.)

\[
\text{setstate (state)}
\]

Set the state of the encoder to state. state must be a decoder state returned by getstate().

The StreamWriter and StreamReader classes provide generic working interfaces which can be used to implement new encoding submodules very easily. See encodings.utf_8 for an example of how this is done.

**StreamWriter Objects**

The StreamWriter class is a subclass of Codec and defines the following methods which every stream writer must define in order to be compatible with the Python codec registry.

\[
\text{class codecs.StreamWriter (stream[, errors])}
\]

Constructor for a StreamWriter instance.

All stream writers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

stream must be a file-like object open for writing binary data.

The StreamWriter may implement different error handling schemes by providing the errors keyword argument. These parameters are predefined:

• ‘strict’ Raise ValueError (or a subclass); this is the default.
• ‘ignore’ Ignore the character and continue with the next.
• ‘replace’ Replace with a suitable replacement character
• ‘xmlcharrefreplace’ Replace with the appropriate XML character reference
• ‘backslashreplace’ Replace with backslashed escape sequences.

The errors argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the StreamWriter object.

The set of allowed values for the errors argument can be extended with register_error().

---

7.2. **codecs — Codec registry and base classes**
**write** *(object)*  
Writes the object’s contents encoded to the stream.

**writelines** *(list)*  
Writes the concatenated list of strings to the stream (possibly by reusing the **write**() method).

**reset** ()  
Flushes and resets the codec buffers used for keeping state.  
Calling this method should ensure that the data on the output is put into a clean state that allows appending of new fresh data without having to rescan the whole stream to recover state.

In addition to the above methods, the **StreamWriter** must also inherit all other methods and attributes from the underlying stream.

### StreamReader Objects

The **StreamReader** class is a subclass of **Codec** and defines the following methods which every stream reader must define in order to be compatible with the Python codec registry.

**class** codecs.**StreamReader** *(stream[, errors]*)  
Constructor for a **StreamReader** instance.  
All stream readers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

*stream* must be a file-like object open for reading (binary) data.

The **StreamReader** may implement different error handling schemes by providing the **errors** keyword argument. These parameters are defined:

- `'strict'` Raise **ValueError** (or a subclass); this is the default.
- `'ignore'` Ignore the character and continue with the next.
- `'replace'` Replace with a suitable replacement character.

The **errors** argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the **StreamReader** object.

The set of allowed values for the **errors** argument can be extended with **register_error()**.

**read** *(size[, chars[, firstline]]]*)  
Decodes data from the stream and returns the resulting object.  
*chars* indicates the number of characters to read from the stream. **read()** will never return more than *chars* characters, but it might return less, if there are not enough characters available.

*size* indicates the approximate maximum number of bytes to read from the stream for decoding purposes. The decoder can modify this setting as appropriate. The default value -1 indicates to read and decode as much as possible. *size* is intended to prevent having to decode huge files in one step.

*firstline* indicates that it would be sufficient to only return the first line, if there are decoding errors on later lines.

The method should use a greedy read strategy meaning that it should read as much data as is allowed within the definition of the encoding and the given *size*, e.g. if optional encoding endings or state markers are available on the stream, these should be read too.

**readline** *(size[, keepends]*)  
Read one line from the input stream and return the decoded data.

*size*, if given, is passed as size argument to the stream’s **read()** method.

If **keepends** is false line-endings will be stripped from the lines returned.
readlines ([sizehint[, keepends ]])
Read all lines available on the input stream and return them as a list of lines.
Line-endings are implemented using the codec’s decoder method and are included in the list entries if keepends is true.
sizehint, if given, is passed as the size argument to the stream’s read() method.
reset ()
Resets the codec buffers used for keeping state.
Note that no stream repositioning should take place. This method is primarily intended to be able to recover from decoding errors.
In addition to the above methods, the StreamReader must also inherit all other methods and attributes from the underlying stream.
The next two base classes are included for convenience. They are not needed by the codec registry, but may provide useful in practice.

StreamReaderWriter Objects

The StreamReaderWriter allows wrapping streams which work in both read and write modes.
The design is such that one can use the factory functions returned by the lookup() function to construct the instance.

class codecs.StreamReaderWriter (stream, Reader, Writer, errors)
Creates a StreamReaderWriter instance. stream must be a file-like object. Reader and Writer must be factory functions or classes providing the StreamReader and StreamWriter interface resp. Error handling is done in the same way as defined for the stream readers and writers.
StreamReaderWriter instances define the combined interfaces of StreamReader and StreamWriter classes. They inherit all other methods and attributes from the underlying stream.

StreamRecoder Objects

The StreamRecoder provide a frontend - backend view of encoding data which is sometimes useful when dealing with different encoding environments.
The design is such that one can use the factory functions returned by the lookup() function to construct the instance.

class codecs.StreamRecoder (stream, encode, decode, Reader, Writer, errors)
Creates a StreamRecoder instance which implements a two-way conversion: encode and decode work on the frontend (the input to read() and output of write()) while Reader and Writer work on the backend (reading and writing to the stream).
You can use these objects to do transparent direct recodings from e.g. Latin-1 to UTF-8 and back.
stream must be a file-like object.
encode, decode must adhere to the Codec interface. Reader, Writer must be factory functions or classes providing objects of the StreamReader and StreamWriter interface respectively.
encode and decode are needed for the frontend translation, Reader and Writer for the backend translation.
Error handling is done in the same way as defined for the stream readers and writers.
StreamRecoder instances define the combined interfaces of StreamReader and StreamWriter classes. They inherit all other methods and attributes from the underlying stream.
7.2.2 Encodings and Unicode

Strings are stored internally as sequences of codepoints in range 0 – 10FFFF (see PEP 393 for more details about the implementation). Once a string object is used outside of CPU and memory, CPU endianness and how these arrays are stored as bytes become an issue. Transforming a string object into a sequence of bytes is called encoding and recreating the string object from the sequence of bytes is known as decoding. There are many different methods for how this transformation can be done (these methods are also called encodings). The simplest method is to map the codepoints 0-255 to the bytes 0x00-0xff. This means that a string object that contains codepoints above U+00FF can’t be encoded with this method (which is called ‘latin-1’ or ‘iso-8859-1’). str.encode() will raise a UnicodeEncodeError that looks like this: UnicodeEncodeError: ‘latin-1’ codec can’t encode character ‘\u1234’ in position 3: ordinal not in range(256).

There’s another group of encodings (the so called charmap encodings) that choose a different subset of all Unicode code points and how these codepoints are mapped to the bytes 0x00-0xff. To see how this is done simply open e.g. encodings/cp1252.py (which is an encoding that is used primarily on Windows). There’s a string constant with 256 characters that shows you which character is mapped to which byte value.

All of these encodings can only encode 256 of the 1114112 codepoints defined in Unicode. A simple and straightforward way that can store each Unicode code point, is to store each codepoint as four consecutive bytes. There are two possibilities: store the bytes in big endian or in little endian order. These two encodings are called UTF-32-BE and UTF-32-LE respectively. Their disadvantage is that if e.g. you use UTF-32-BE on a little endian machine you will always have to swap bytes on encoding and decoding. UTF-32 avoids this problem: bytes will always be in natural endianness. When these bytes are read by a CPU with a different endianness, then bytes have to be swapped though. To be able to detect the endianness of a UTF-16 or UTF-32 byte sequence, there’s the so called BOM (“Byte Order Mark”). This is the Unicode character U+FEFF. This character can be prepended to every UTF-16 or UTF-32 byte sequence. The byte swapped version of this character (0xfffe) is an illegal character that may not appear in a Unicode text. So when the first character in an UTF-16 or UTF-32 byte sequence appears to be a U+FEFE the bytes have to be swapped on decoding. Unfortunately the character U+FEFF had a second purpose as a ZERO WIDTH NO-BREAK SPACE: a character that has no width and doesn’t allow a word to be split. It can e.g. be used to give hints to a ligature algorithm. With Unicode 4.0 using U+FEFF as a ZERO WIDTH NO-BREAK SPACE has been deprecated (with U+2060 (WORD JOINER) assuming this role). Nevertheless Unicode software still must be able to handle U+FEFF in both roles: as a BOM it’s a device to determine the storage layout of the encoded bytes, and vanishes once the byte sequence has been decoded into a string; as a ZERO WIDTH NO-BREAK SPACE it’s a normal character that will be decoded like any other.

There’s another encoding that is able to encoding the full range of Unicode characters: UTF-8. UTF-8 is an 8-bit encoding, which means there are no issues with byte order in UTF-8. Each byte in a UTF-8 byte sequence consists of two parts: marker bits (the most significant bits) and payload bits. The marker bits are a sequence of zero to four 1 bits followed by a 0 bit. Unicode characters are encoded like this (with x being payload bits, which when concatenated give the Unicode character):

<table>
<thead>
<tr>
<th>Range</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>U-000000000 ... U-0000000FF</td>
<td>0xxxxxxx</td>
</tr>
<tr>
<td>U-000000080 ... U-00000007FF</td>
<td>110xxxx 10xxxxxx</td>
</tr>
<tr>
<td>U-000000080 ... U-0000000FF</td>
<td>1110xxx 10xxxxxx 10xxxxxx</td>
</tr>
<tr>
<td>U-000100000 ... U-0001000FF</td>
<td>11110xxx 10xxxxxx 10xxxxxx 10xxxxxx</td>
</tr>
</tbody>
</table>

The least significant bit of the Unicode character is the rightmost x bit.

As UTF-8 is an 8-bit encoding no BOM is required and any U+FEFF character in the decoded string (even if it’s the first character) is treated as a ZERO WIDTH NO-BREAK SPACE.

Without external information it’s impossible to reliably determine which encoding was used for encoding a string. Each charmap encoding can decode any random byte sequence. However that’s not possible with UTF-8, as UTF-8 byte sequences have a structure that doesn’t allow arbitrary byte sequences. To increase the reliability with which a UTF-8 encoding can be detected, Microsoft invented a variant of UTF-8 (that Python 2.5 calls "utf-8-sig") for its Notepad program: Before any of the Unicode characters is written to the file, a UTF-8 encoded BOM (which looks like this as a byte sequence: 0xef, 0xbb, 0xbf) is written. As it’s rather improbable that any charmap encoded file starts with these byte values (which would e.g. map to...
in iso-8859-1), this increases the probability that a \texttt{utf-8-sig} encoding can be correctly guessed from the byte sequence. So here the BOM is not used to be able to determine the byte order used for generating the byte sequence, but as a signature that helps in guessing the encoding. On encoding the \texttt{utf-8-sig} codec will write 0xef, 0xbb, 0xbf as the first three bytes to the file. On decoding \texttt{utf-8-sig} will skip those three bytes if they appear as the first three bytes in the file. In UTF-8, the use of the BOM is discouraged and should generally be avoided.

### 7.2.3 Standard Encodings

Python comes with a number of codecs built-in, either implemented as C functions or with dictionaries as mapping tables. The following table lists the codecs by name, together with a few common aliases, and the languages for which the encoding is likely used. Neither the list of aliases nor the list of languages is meant to be exhaustive. Notice that spelling alternatives that only differ in case or use a hyphen instead of an underscore are also valid aliases; therefore, e.g. \texttt{utf-8} is a valid alias for the \texttt{utf_8} codec.

**CPython implementation detail:** Some common encodings can bypass the codecs lookup machinery to improve performance. These optimization opportunities are only recognized by CPython for a limited set of aliases: utf-8, utf8, latin-1, latin1, iso-8859-1, mbcs (Windows only), ascii, utf-16, and utf-32. Using alternative spellings for these encodings may result in slower execution.

Many of the character sets support the same languages. They vary in individual characters (e.g. whether the EURO SIGN is supported or not), and in the assignment of characters to code positions. For the European languages in particular, the following variants typically exist:

- an ISO 8859 codeset
- a Microsoft Windows code page, which is typically derived from a 8859 codeset, but replaces control characters with additional graphic characters
- an IBM EBCDIC code page
- an IBM PC code page, which is ASCII compatible

<table>
<thead>
<tr>
<th>Codec</th>
<th>Aliases</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii</td>
<td>646, us-ascii</td>
<td>English</td>
</tr>
<tr>
<td>big5</td>
<td>big5-tw, csbig5</td>
<td>Traditional Chinese</td>
</tr>
<tr>
<td>big5hkscs</td>
<td>big5-hkscs, hkscs</td>
<td>Traditional Chinese</td>
</tr>
<tr>
<td>cp037</td>
<td>IBM037, IBM039</td>
<td>English</td>
</tr>
<tr>
<td>cp424</td>
<td>EBCDIC-CP-HE, IBM424</td>
<td>Hebrew</td>
</tr>
<tr>
<td>cp437</td>
<td>437, IBM437</td>
<td>English</td>
</tr>
<tr>
<td>cp500</td>
<td>EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500</td>
<td>Western Europe</td>
</tr>
<tr>
<td>cp720</td>
<td>IBM775</td>
<td>Arabic</td>
</tr>
<tr>
<td>cp737</td>
<td>850, IBM850</td>
<td>Greek</td>
</tr>
<tr>
<td>cp775</td>
<td>852, IBM852</td>
<td>Baltic languages</td>
</tr>
<tr>
<td>cp850</td>
<td>855, IBM855</td>
<td>Bulgarian, Byelorussian, Macedonian, Russian, Serbian</td>
</tr>
<tr>
<td>cp855</td>
<td>857, IBM857</td>
<td>Hebrew</td>
</tr>
<tr>
<td>cp856</td>
<td>858, IBM858</td>
<td>Turkish</td>
</tr>
<tr>
<td>cp857</td>
<td>860, IBM860</td>
<td>Western Europe</td>
</tr>
<tr>
<td>cp858</td>
<td>861, CP-IS, IBM861</td>
<td>Portuguese</td>
</tr>
<tr>
<td>cp860</td>
<td>862, IBM862</td>
<td>Icelandic</td>
</tr>
<tr>
<td>cp861</td>
<td>863, IBM863</td>
<td>Hebrew</td>
</tr>
<tr>
<td>cp862</td>
<td>864, IBM864</td>
<td>Arabic</td>
</tr>
</tbody>
</table>

Continued on next page
Table 7.1 – continued from previous page

<table>
<thead>
<tr>
<th>Code</th>
<th>Characters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cp865</td>
<td>865, IBM865</td>
<td>Danish, Norwegian</td>
</tr>
<tr>
<td>cp866</td>
<td>866, IBM866</td>
<td>Russian</td>
</tr>
<tr>
<td>cp869</td>
<td>869, CP-GR, IBM869</td>
<td>Greek</td>
</tr>
<tr>
<td>cp874</td>
<td>949, ms949, uhc</td>
<td>Thai</td>
</tr>
<tr>
<td>cp875</td>
<td>950, ms950</td>
<td>Greek</td>
</tr>
<tr>
<td>cp932</td>
<td>932, ms932, mskanji, ms-kanji</td>
<td>Japanese</td>
</tr>
<tr>
<td>cp949</td>
<td>869, cp869</td>
<td>Korean</td>
</tr>
<tr>
<td>cp950</td>
<td>949, cp949</td>
<td>Traditional Chinese</td>
</tr>
<tr>
<td>cp1006</td>
<td>ibm1026</td>
<td>Urdu</td>
</tr>
<tr>
<td>cp1026</td>
<td>ibm1140</td>
<td>Turkish</td>
</tr>
<tr>
<td>cp1140</td>
<td>windows-1250</td>
<td>Central and Eastern Europe</td>
</tr>
<tr>
<td>cp1250</td>
<td>windows-1251</td>
<td>Bulgarian, Byelorussian, Macedonian, Russian, Serbian</td>
</tr>
<tr>
<td>cp1252</td>
<td>windows-1252</td>
<td>Western Europe</td>
</tr>
<tr>
<td>cp1253</td>
<td>windows-1253</td>
<td>Greek</td>
</tr>
<tr>
<td>cp1254</td>
<td>windows-1254</td>
<td>Turkish</td>
</tr>
<tr>
<td>cp1255</td>
<td>windows-1255</td>
<td>Hebrew</td>
</tr>
<tr>
<td>cp1256</td>
<td>windows-1256</td>
<td>Arabic</td>
</tr>
<tr>
<td>cp1257</td>
<td>windows-1257</td>
<td>Baltic languages</td>
</tr>
<tr>
<td>cp1258</td>
<td>windows-1258</td>
<td>Vietnamese</td>
</tr>
<tr>
<td>cp65001</td>
<td>windows-1251</td>
<td>Simplified Chinese</td>
</tr>
<tr>
<td>gb2312</td>
<td>chinese, csiso58gb231280</td>
<td>Unified Chinese</td>
</tr>
<tr>
<td>gbk</td>
<td>936, cp936, ms936</td>
<td>Unified Chinese</td>
</tr>
<tr>
<td>gb18030</td>
<td>gb18030-2000</td>
<td>Simplified Chinese</td>
</tr>
<tr>
<td>hz</td>
<td>hzgb, kz-gb, hz-gb-2312</td>
<td>Japanese</td>
</tr>
<tr>
<td>iso2022_jp</td>
<td>csiso2022jp, iso2022jp, iso-2022-jp</td>
<td>Japanese</td>
</tr>
<tr>
<td>iso2022_jp_1</td>
<td>iso2022jp-1, iso-2022-jp-1</td>
<td>Japanese, Korean, Simplified Chinese,</td>
</tr>
<tr>
<td>iso2022_jp_2</td>
<td>iso2022jp-2, iso-2022-jp-2</td>
<td>Chinese, Western Europe, Greek</td>
</tr>
<tr>
<td>iso2022_jp_3</td>
<td>iso2022jp-3, iso-2022-jp-3</td>
<td>Japanese</td>
</tr>
<tr>
<td>iso2022_jp_ext</td>
<td>iso2022jp-ext, iso-2022-jp-ext</td>
<td>Japanese</td>
</tr>
<tr>
<td>latin_1</td>
<td>iso-8859-1, iso8859-1, 8859</td>
<td>West Europe</td>
</tr>
<tr>
<td>iso8859_2</td>
<td>iso-8859-2, latin2, L2</td>
<td>Central and Eastern Europe</td>
</tr>
<tr>
<td>iso8859_3</td>
<td>iso-8859-3, latin3, L3</td>
<td>Esperanto, Maltese</td>
</tr>
<tr>
<td>iso8859_4</td>
<td>iso-8859-4, latin4, L4</td>
<td>Baltic languages</td>
</tr>
<tr>
<td>iso8859_5</td>
<td>iso-8859-5, cyrillic</td>
<td>Bulgarian, Byelorussian, Macedonian, Russian, Serbian</td>
</tr>
<tr>
<td>iso8859_6</td>
<td>iso-8859-6, arabic</td>
<td>Arabic</td>
</tr>
<tr>
<td>iso8859_7</td>
<td>iso-8859-7, greek, greek8</td>
<td>Greek</td>
</tr>
</tbody>
</table>

continued on next page
<table>
<thead>
<tr>
<th>Encoding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iso8859_8</td>
<td>iso-8859-8, hebrew</td>
</tr>
<tr>
<td>iso8859_9</td>
<td>iso-8859-9, latin5, L5</td>
</tr>
<tr>
<td>iso8859_10</td>
<td>iso-8859-10, latin6, L6</td>
</tr>
<tr>
<td>iso8859_13</td>
<td>iso-8859-13, latin7, L7</td>
</tr>
<tr>
<td>iso8859_14</td>
<td>iso-8859-14, latin8, L8</td>
</tr>
<tr>
<td>iso8859_15</td>
<td>iso-8859-15, latin9, L9</td>
</tr>
<tr>
<td>iso8859_16</td>
<td>iso-8859-16, latin10, L10</td>
</tr>
<tr>
<td>johab</td>
<td>cp1361, ms1361</td>
</tr>
<tr>
<td>koi8_r</td>
<td>maccyrillic</td>
</tr>
<tr>
<td>koi8_u</td>
<td></td>
</tr>
<tr>
<td>mac_cyrillic</td>
<td>maccyrillic</td>
</tr>
<tr>
<td>mac_greek</td>
<td>macgreek</td>
</tr>
<tr>
<td>mac_iceland</td>
<td>maciceland</td>
</tr>
<tr>
<td>mac_latin2</td>
<td>maclatin2, maccentraleurope</td>
</tr>
<tr>
<td>mac_roman</td>
<td>macroman, macintosh</td>
</tr>
<tr>
<td>mac_turkish</td>
<td>macturkish</td>
</tr>
<tr>
<td>ptcp154</td>
<td>csptcp154, pt154, cp154, cyrillic-asian</td>
</tr>
<tr>
<td>shiftjis</td>
<td>csshiftjis, shiftjis, sjis, s_jis</td>
</tr>
<tr>
<td>shiftjisx0213</td>
<td>shiftjisx0213, sjisx0213, s_jisx0213</td>
</tr>
<tr>
<td>utf_32</td>
<td>U32, utf32</td>
</tr>
<tr>
<td>utf_32_be</td>
<td>UTF-32BE</td>
</tr>
<tr>
<td>utf_32_le</td>
<td>UTF-32LE</td>
</tr>
<tr>
<td>utf_16</td>
<td>U16, utf16</td>
</tr>
<tr>
<td>utf_16_be</td>
<td>UTF-16BE</td>
</tr>
<tr>
<td>utf_16_le</td>
<td>UTF-16LE</td>
</tr>
<tr>
<td>utf_7</td>
<td>U7, unicode-1-1-utf-7</td>
</tr>
<tr>
<td>utf_8</td>
<td>U8, UTF, utf8</td>
</tr>
<tr>
<td>utf_8_sig</td>
<td>all languages</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Table 7.1 – continued from previous page</strong></td>
</tr>
</tbody>
</table>

7.2.4 Python Specific Encodings

A number of predefined codecs are specific to Python, so their codec names have no meaning outside Python. These are listed in the tables below based on the expected input and output types (note that while text encodings are the most common use case for codecs, the underlying codec infrastructure supports arbitrary data transforms rather than just text encodings). For asymmetric codecs, the stated purpose describes the encoding direction.

The following codecs provide `str` to `bytes` encoding and `bytes-like object` to `str` decoding, similar to the Unicode text encodings.
<table>
<thead>
<tr>
<th>Codec</th>
<th>Aliases</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>idna</td>
<td></td>
<td>Implements <a href="https://tools.ietf.org/html/rfc3490">RFC 3490</a>, see also <a href="https://docs.python.org/3/library/encodings.html">encodings.idna</a></td>
</tr>
<tr>
<td>mbcs</td>
<td>dbcs</td>
<td>Windows only: Encode operand according to the ANSI codepage (CP_ACP)</td>
</tr>
<tr>
<td>palmos</td>
<td></td>
<td>Encoding of PalmOS 3.5</td>
</tr>
<tr>
<td>punycode</td>
<td></td>
<td>Implements <a href="https://tools.ietf.org/html/rfc3492">RFC 3492</a></td>
</tr>
<tr>
<td>raw_unicode_escape</td>
<td></td>
<td>Produce a string that is suitable as raw Unicode literal in Python source code</td>
</tr>
<tr>
<td>undefined</td>
<td></td>
<td>Raise an exception for all conversions. Can be used as the system encoding if no automatic coercion between byte and Unicode strings is desired.</td>
</tr>
<tr>
<td>unicode_escape</td>
<td></td>
<td>Produce a string that is suitable as Unicode literal in Python source code.</td>
</tr>
<tr>
<td>unicode_internal</td>
<td></td>
<td>Return the internal representation of the operand. Deprecated since version 3.3</td>
</tr>
</tbody>
</table>

The following codecs provide *bytes-like object* to *bytes* mappings.
<table>
<thead>
<tr>
<th>Codec</th>
<th>Purpose/encoder/decoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>base64_codec</td>
<td>Convert operand to MIME base64 (the result always includes a trailing '\n')</td>
</tr>
<tr>
<td>bz2_codec</td>
<td>Compress the operand using bz2.compress(), bz2.decompress()</td>
</tr>
<tr>
<td>hex_codec</td>
<td>Convert operand to hexadecimal representation with two digits per byte</td>
</tr>
<tr>
<td>quopri_codec</td>
<td>Convert operand to MIME quoted printable</td>
</tr>
<tr>
<td>uu_codec</td>
<td>Convert the operand using uu.encode(), uu.decode()</td>
</tr>
</tbody>
</table>

7.2. codecs — Codec registry and base classes

Rather than accepting any bytes-like object, 'base64_codec' accepts only bytes and bytearray for encoding and only bytes, 7.2.4 codecs — Codec registry and base classes 135
7.2.5 encodings.idna — Internationalized Domain Names in Applications

This module implements RFC 3490 (Internationalized Domain Names in Applications) and RFC 3492 (Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN)). It builds upon the punycode encoding and stringprep.

These RFCs together define a protocol to support non-ASCII characters in domain names. A domain name containing non-ASCII characters (such as www.Alliancefrançaise.nu) is converted into an ASCII-compatible encoding (ACE, such as www.xn--alliancefranaise-npb.nu). The ACE form of the domain name is then used in all places where arbitrary characters are not allowed by the protocol, such as DNS queries, HTTP Host fields, and so on. This conversion is carried out in the application; if possible invisible to the user. The application should transparently convert Unicode domain labels to IDNA on the wire, and convert back ACE labels to Unicode before presenting them to the user.

Python supports this conversion in several ways: the idna codec performs conversion between Unicode and ACE, separating an input string into labels based on the separator characters defined in section 3.1 (1) of RFC 3490 and converting each label to ACE as required, and conversely separating an input byte string into labels based on the . separator and converting any ACE labels found into unicode. Furthermore, the socket module transparently converts Unicode host names to ACE, so that applications need not be concerned about converting host names themselves when they pass them to the socket module. On top of that, modules that have host names as function parameters, such as http.client and ftplib, accept Unicode host names (http.client then also transparently sends an IDNA hostname in the Host field if it sends that field at all).

When receiving host names from the wire (such as in reverse name lookup), no automatic conversion to Unicode is performed: Applications wishing to present such host names to the user should decode them to Unicode.

The module encodings.idna also implements the nameprep procedure, which performs certain normalizations on host names, to achieve case-insensitivity of international domain names, and to unify similar characters. The nameprep functions can be used directly if desired.

```python
encodings.idna.nameprep(label)

Return the nameprepped version of label. The implementation currently assumes query strings, so AllowUnassigned is true.
```

```python
encodings.idna.ToASCII(label)

Convert a label to ASCII, as specified in RFC 3490. UseSTD3ASCIIRules is assumed to be false.
```

```python
encodings.idna.ToUnicode(label)

Convert a label to Unicode, as specified in RFC 3490.
```

7.2.6 encodings.mbcsl — Windows ANSI codepage

Encode operand according to the ANSI codepage (CP_ACP).

Availability: Windows only. Changed in version 3.3: Support any error handler. Changed in version 3.2: Before 3.2, the errors argument was ignored; ‘replace’ was always used to encode, and ‘ignore’ to decode.

7.2.7 encodings.utf_8_sig — UTF-8 codec with BOM signature

This module implements a variant of the UTF-8 codec: On encoding a UTF-8 encoded BOM will be prepended to the UTF-8 encoded bytes. For the stateful encoder this is only done once (on the first write to the byte stream). For decoding an optional UTF-8 encoded BOM at the start of the data will be skipped.
The modules described in this chapter provide a variety of specialized data types such as dates and times, fixed-type arrays, heap queues, synchronized queues, and sets.

Python also provides some built-in data types, in particular, `dict`, `list`, `set` and `frozenset`, and `tuple`. The `str` class is used to hold Unicode strings, and the `bytes` class is used to hold binary data.

The following modules are documented in this chapter:

### 8.1 `datetime` — Basic date and time types

The `datetime` module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient attribute extraction for output formatting and manipulation. For related functionality, see also the `time` and `calendar` modules.

There are two kinds of date and time objects: “naive” and “aware”.

An aware object has sufficient knowledge of applicable algorithmic and political time adjustments, such as time zone and daylight saving time information, to locate itself relative to other aware objects. An aware object is used to represent a specific moment in time that is not open to interpretation.\(^1\)

A naive object does not contain enough information to unambiguously locate itself relative to other date/time objects. Whether a naive object represents Coordinated Universal Time (UTC), local time, or time in some other timezone is purely up to the program, just like it is up to the program whether a particular number represents metres, miles, or mass. Naive objects are easy to understand and to work with, at the cost of ignoring some aspects of reality.

For applications requiring aware objects, `datetime` and `time` objects have an optional time zone information attribute, `tzinfo`, that can be set to an instance of a subclass of the abstract `tzinfo` class. These `tzinfo` objects capture information about the offset from UTC time, the time zone name, and whether Daylight Saving Time is in effect. Note that only one concrete `tzinfo` class, the `timezone` class, is supplied by the `datetime` module. The `timezone` class can represent simple timezones with fixed offset from UTC, such as UTC itself or North American EST and EDT timezones. Supporting timezones at deeper levels of detail is up to the application. The rules for time adjustment across the world are more political than rational, change frequently, and there is no standard suitable for every application aside from UTC.

The `datetime` module exports the following constants:

```python
datetime.MINYEAR
```

The smallest year number allowed in a `date` or `datetime` object. `MINYEAR` is 1.

```python
datetime.MAXYEAR
```

The largest year number allowed in a `date` or `datetime` object. `MAXYEAR` is 9999.

See Also:

- **Module `calendar`** General calendar related functions.

---

\(^1\) If, that is, we ignore the effects of Relativity
Module `time` Time access and conversions.

### 8.1.1 Available Types

- **class datetime.date**
  An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Attributes: `year`, `month`, and `day`.

- **class datetime.time**
  An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds (there is no notion of “leap seconds” here). Attributes: `hour`, `minute`, `second`, `microsecond`, and `tzinfo`.

- **class datetime.datetime**
  A combination of a date and a time. Attributes: `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond`, and `tzinfo`.

- **class datetime.timedelta**
  A duration expressing the difference between two `date`, `time`, or `datetime` instances to microsecond resolution.

- **class datetime.tzinfo**
  An abstract base class for time zone information objects. These are used by the `datetime` and `time` classes to provide a customizable notion of time adjustment (for example, to account for time zone and/or daylight saving time).

- **class datetime.timezone**
  A class that implements the `tzinfo` abstract base class as a fixed offset from the UTC. New in version 3.2.

Objects of these types are immutable.

Objects of the `date` type are always naive.

An object of type `time` or `datetime` may be naive or aware. A `datetime` object `d` is aware if `d.tzinfo` is not `None` and `d.tzinfo.utcoffset(d)` does not return `None`. If `d.tzinfo` is `None`, or if `d.tzinfo` is not `None` but `d.tzinfo.utcoffset(d)` returns `None`, `d` is naive. A `time` object `t` is aware if `t.tzinfo` is not `None` and `t.tzinfo.utcoffset(None)` does not return `None`. Otherwise, `t` is naive.

The distinction between naive and aware doesn’t apply to `timedelta` objects.

Subclass relationships:

```
object
timedelta
tzinfo
timezone
time
date
datetime
```

### 8.1.2 timedelta Objects

A `timedelta` object represents a duration, the difference between two dates or times.

- **class datetime.timedelta**
  ```python
days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0
  ```
  All arguments are optional and default to 0. Arguments may be integers or floats, and may be positive or negative.

  Only `days`, `seconds` and `microseconds` are stored internally. Arguments are converted to those units:
  
  - A millisecond is converted to 1000 microseconds.
  - A minute is converted to 60 seconds.
• An hour is converted to 3600 seconds.
• A week is converted to 7 days.

and days, seconds and microseconds are then normalized so that the representation is unique, with

• \(0 \leq \text{microseconds} < 1000000\)
• \(0 \leq \text{seconds} < 3600 \times 24\) (the number of seconds in one day)
• \(-999999999 \leq \text{days} \leq 999999999\)

If any argument is a float and there are fractional microseconds, the fractional microseconds left over from all arguments are combined and their sum is rounded to the nearest microsecond. If no argument is a float, the conversion and normalization processes are exact (no information is lost).

If the normalized value of days lies outside the indicated range, `OverflowError` is raised.

Note that normalization of negative values may be surprising at first. For example,

```python
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

Class attributes are:

`timedelta.min`

The most negative `timedelta` object, `timedelta(-999999999)`.

`timedelta.max`

The most positive `timedelta` object, `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

`timedelta.resolution`

The smallest possible difference between non-equal `timedelta` objects, `timedelta(microseconds=1)`.

Note that, because of normalization, `timedelta.max > -timedelta.min`. `-timedelta.max` is not representable as a `timedelta` object.

Instance attributes (read-only):

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>days</td>
<td>Between -999999999 and 999999999 inclusive</td>
</tr>
<tr>
<td>seconds</td>
<td>Between 0 and 86399 inclusive</td>
</tr>
<tr>
<td>microseconds</td>
<td>Between 0 and 999999 inclusive</td>
</tr>
</tbody>
</table>

Supported operations:
<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 = t_2 + t_3 )</td>
<td>Sum of ( t_2 ) and ( t_3 ). Afterwards ( t_1 - t_2 = t_3 ) and ( t_1 - t_3 = t_2 ) are true. (1)</td>
</tr>
<tr>
<td>( t_1 = t_2 - t_3 )</td>
<td>Difference of ( t_2 ) and ( t_3 ). Afterwards ( t_1 = t_2 - t_3 ) and ( t_2 = t_1 + t_3 ) are true. (1)</td>
</tr>
<tr>
<td>( t_1 = t_2 \times i ) or ( t_1 = i \times t_2 )</td>
<td>Delta multiplied by an integer. Afterwards ( t_1 \div i = t_2 ) is true, provided ( i \neq 0 ). In general, ( t_1 \times i = t_1 \times (i-1) + t_1 ) is true. (1)</td>
</tr>
<tr>
<td>( t_1 = t_2 \div t_3 )</td>
<td>Division (3) of ( t_2 ) by ( t_3 ). Returns a float object.</td>
</tr>
<tr>
<td>( t_1 = t_2 \div f ) or ( t_1 = t_2 \div i )</td>
<td>Delta divided by a float or an int. The result is rounded to the nearest multiple of ( \text{timedelta}.\text{resolution} ) using round-half-to-even.</td>
</tr>
<tr>
<td>( t_1 = t_2 \mod t_3 )</td>
<td>The remainder is computed as a timedelta object. (3)</td>
</tr>
<tr>
<td>( q, r = \text{divmod}(t_1, t_2) )</td>
<td>Computes the quotient and the remainder; ( q = t_1 \div t_2 ) and ( r = t_1 \mod t_2 ). ( q ) is an integer and ( r ) is a timedelta object.</td>
</tr>
<tr>
<td>( +t_1 )</td>
<td>Returns a timedelta object with the same value. (2)</td>
</tr>
<tr>
<td>( -t_1 )</td>
<td>Equivalent to ( \text{timedelta}((-t_1).\text{days}, -t_1.\text{seconds}, -t_1.\text{microseconds}) ), and to ( t_1^*^{-1} ). (1)(4)</td>
</tr>
<tr>
<td>abs(t)</td>
<td>Equivalent to (+t) when ( t.\text{days} \geq 0 ), and to (-t) when ( t.\text{days} &lt; 0 ). (2)</td>
</tr>
<tr>
<td>str(t)</td>
<td>Returns a string in the form (<a href="">D \text{ day[s]}, </a>, where ( D ) is negative for negative ( t ). (5)</td>
</tr>
<tr>
<td>repr(t)</td>
<td>Returns a string in the form ( \text{datetime.timedelta}(D[, S[, U]]) ), where ( D ) is negative for negative ( t ). (5)</td>
</tr>
</tbody>
</table>

Notes:

1. This is exact, but may overflow.
2. This is exact, and cannot overflow.
3. Division by 0 raises \text{ZeroDivisionError}. (3)
4. -\text{timedelta}.\text{max} is not representable as a timedelta object.
5. String representations of timedelta objects are normalized similarly to their internal representation. This leads to somewhat unusual results for negative time deltas. For example:

```python
>>> timedelta(hours=-5)
datetime.timedelta(hours=-5)
>>> print(timedelta(hours=-5))
-1 day, 19:00:00
```

In addition to the operations listed above timedelta objects support certain additions and subtractions with date and datetime objects (see below). Changed in version 3.2: Floor division and true division of a timedelta object by another timedelta object are now supported, as are remainder operations and the \text{divmod()} function. True division and multiplication of a timedelta object by a float object are now supported. Comparisons of timedelta objects are supported with the timedelta object representing the smaller duration considered to be the smaller timedelta. In order to stop mixed-type comparisons from falling back to the default comparison by object address, when a timedelta object is compared to an object of a different type, TypeError is raised unless the comparison is == or !=. The latter cases return False or True, respectively. timedelta objects are hashable (usable as dictionary keys), support efficient pickling, and in Boolean contexts, a timedelta object is considered to be true if and only if it isn’t equal to timedelta(0).

Instance methods:

- \text{timedelta}.\text{total_seconds}()
  Return the total number of seconds contained in the duration. Equivalent to \( t_1 \div \text{timedelta}(\text{seconds}=1) \).

  Note that for very large time intervals (greater than 270 years on most platforms) this method will lose microsecond accuracy. New in version 3.2.
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23, ...
... minutes=50, seconds=600)  # adds up to 365 days
>>> year.total_seconds()
31536000.0
>>> year == another_year
True
>>> ten_years = 10 * year
>>> ten_years, ten_years.days // 365
(datetime.timedelta(3650), 10)
>>> nine_years = ten_years - year
>>> nine_years, nine_years.days // 365
(datetime.timedelta(3285), 9)
>>> three_years = nine_years // 3;
>>> three_years, three_years.days // 365
(datetime.timedelta(1095), 3)
>>> abs(three_years - ten_years) == 2 * three_years + year
True

8.1.3 date Objects

A date object represents a date (year, month and day) in an idealized calendar, the current Gregorian calendar indefinitely extended in both directions. January 1 of year 1 is called day number 1, January 2 of year 1 is called day number 2, and so on. This matches the definition of the “proleptic Gregorian” calendar in Dershowitz and Reingold’s book Calendrical Calculations, where it’s the base calendar for all computations. See the book for algorithms for converting between proleptic Gregorian ordinals and many other calendar systems.

class datetime.date(year, month, day)
    All arguments are required. Arguments may be integers, in the following ranges:
    • MINYEAR <= year <= MAXYEAR
    • 1 <= month <= 12
    • 1 <= day <= number of days in the given month and year

    If an argument outside those ranges is given, ValueError is raised.

Other constructors, all class methods:
classmethod date.today()
    Return the current local date. This is equivalent to date.fromtimestamp(time.time()).

classmethod date.fromtimestamp(timestamp)
    Return the local date corresponding to the POSIX timestamp, such as is returned by time.time(). This may raise OverflowError, if the timestamp is out of the range of values supported by the platform C localtime() function, and OSError on localtime() failure. It’s common for this to be restricted to years from 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by fromtimestamp(). Changed in version 3.3: Raise OverflowError instead of ValueError if the timestamp is out of the range of values supported by the platform C localtime() function. Raise OSError instead of ValueError on localtime() failure.

classmethod date.fromordinal(ordinal)
    Return the date corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. ValueError is raised unless 1 <= ordinal <= date.max.toordinal(). For any date d, date.fromordinal(d.toordinal()) == d.

Class attributes:
date.min
    The earliest representable date.date(MINYEAR, 1, 1).
The latest representable date, `date(MINYEAR, 12, 31)`. (1)

The smallest possible difference between non-equal date objects, `timedelta(days=1)`. (2)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>date2 = date1 + timedelta</code></td>
<td><code>date2</code> is <code>timedelta.days</code> days removed from <code>date1</code>. (1)</td>
</tr>
<tr>
<td><code>date2 = date1 - timedelta</code></td>
<td>Computes <code>date2</code> such that <code>date2 + timedelta == date1</code>. (2)</td>
</tr>
<tr>
<td><code>timedelta = date1 - date2</code></td>
<td>(3)</td>
</tr>
<tr>
<td><code>date1 &lt; date2</code></td>
<td><code>date1</code> is considered less than <code>date2</code> when <code>date1</code> precedes <code>date2</code> in time. (4)</td>
</tr>
</tbody>
</table>

Notes:

1. `date2` is moved forward in time if `timedelta.days > 0`, or backward if `timedelta.days < 0`. Afterward `date2 - date1 == timedelta.days`. `timedelta.seconds` and `timedelta.microseconds` are ignored. `OverflowError` is raised if `date2.year` would be smaller than `MINYEAR` or larger than `MAXYEAR`. (1)

2. This isn’t quite equivalent to `date1 + (-timedelta)`, because `-timedelta` in isolation can overflow in cases where `date1 - timedelta` does not. `timedelta.seconds` and `timedelta.microseconds` are ignored. (2)

3. This is exact, and cannot overflow. `timedelta.seconds` and `timedelta.microseconds` are 0, and `date2 + timedelta == date1` after. (3)

4. In other words, `date1 < date2` if and only if `date1.toordinal() < date2.toordinal()`. In order to stop comparison from falling back to the default scheme of comparing object addresses, date comparison normally raises `TypeError` if the other comparand isn’t also a `date` object. However, `NotImplemented` is returned instead if the other comparand has a `timetuple()` attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a `date` object is compared to an object of a different type, `TypeError` is raised unless the comparison is `==` or `!=`. The latter cases return `False` or `True`, respectively. (4)

Dates can be used as dictionary keys. In Boolean contexts, all `date` objects are considered to be true.

Instance methods:

`date.replace(year, month, day)`

Return a date with the same value, except for those parameters given new values by whichever keyword arguments are specified. For example, if `d == date(2002, 12, 31)`, then `d.replace(day=26) == date(2002, 12, 26)`. (5)

`date.timetuple()`

Return a `time.struct_time` such as returned by `time.localtime()`. The hours, minutes and seconds are 0, and the DST flag is -1. `d.timetuple()` is equivalent to `time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1))`, where `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` is the day number within the current year starting with 1 for January 1st. (6)

`date.toordinal()`

Return the proleptic Gregorian ordinal of the date, where January 1 of year 1 has ordinal 1. For any `date` object `d`, `date.fromordinal(d.toordinal()) == d`. (7)
date.weekday()
    Return the day of the week as an integer, where Monday is 0 and Sunday is 6. For example, date(2002, 12, 4).weekday() == 2, a Wednesday. See also isoweekday().

date.isoweekday()
    Return the day of the week as an integer, where Monday is 1 and Sunday is 7. For example, date(2002, 12, 4).isoweekday() == 3, a Wednesday. See also weekday(), isocalendar().

date.isocalendar()
    Return a 3-tuple, (ISO year, ISO week number, ISO weekday).
    The ISO calendar is a widely used variant of the Gregorian calendar. See http://www.phys.uu.nl/~vgent/calendar/isocalendar.htm for a good explanation.
    The ISO year consists of 52 or 53 full weeks, and where a week starts on a Monday and ends on a Sunday. The first week of an ISO year is the first (Gregorian) calendar week of a year containing a Thursday. This is called week number 1, and the ISO year of that Thursday is the same as its Gregorian year.

date.isoformat()
    Return a string representing the date in ISO 8601 format, 'YYYY-MM-DD'. For example, date(2002, 12, 4).isoformat() == '2002-12-04'.

date.__str__()
    For a date d, str(d) is equivalent to d.isoformat().

date.ctime()
    Return a string representing the date, for example date(2002, 12, 4).ctime() == ‘Wed Dec 4 00:00:00 2002’. d.ctime() is equivalent to time.ctime(time.mktime(d.timetuple())) on platforms where the native C ctime() function (which time.ctime() invokes, but which date.ctime() does not invoke) conforms to the C standard.

date.strftime(format)
    Return a string representing the date, controlled by an explicit format string. Format codes referring to hours, minutes or seconds will see 0 values. For a complete list of formatting directives, see strftime() and strptime() Behavior.

date.__format__(format)
    Same as date.strftime(). This makes it possible to specify format string for a date object when using str.format(). For a complete list of formatting directives, see strftime() and strptime() Behavior.

Example of counting days to an event:
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...    my_birthday = my_birthday.replace(year=today.year + 1)
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202

Example of working with date:
>>> from datetime import date
>>> d = date.fromordinal(730920)  # 730920th day after 1. 1. 0001
>>> d
datetime.date(2002, 3, 11)
>>> t = d.timetuple()
   ...
>>> print(t)
(2002, 3, 11, 0, 0, 0, 70, -1)
>>> ic = d.isocalendar()
>>> for i in ic:
   ...
>>> print(i)
(2002, 11, 1)
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'

8.1.4 datetime Objects

A datetime object is a single object containing all the information from a date object and a time object. Like a date object, datetime assumes the current Gregorian calendar extended in both directions; like a time object, datetime assumes there are exactly 3600*24 seconds in every day.

Constructor:
class datetime.datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None)
The year, month and day arguments are required. tzinfo may be None, or an instance of a tzinfo subclass. The remaining arguments may be integers, in the following ranges:
- MINYEAR <= year <= MAXYEAR
- 1 <= month <= 12
- 1 <= day <= number of days in the given month and year
- 0 <= hour < 24
- 0 <= minute < 60
- 0 <= second < 60
- 0 <= microsecond < 1000000

If an argument outside those ranges is given, ValueError is raised.

Other constructors, all class methods:
classmethod datetime.today()
Return the current local datetime, with tzinfo None. This is equivalent to datetime.fromtimestamp(time.time()). See also now(), fromtimestamp().

classmethod datetime.now(tz=None)
Return the current local date and time. If optional argument tz is None or not specified, this is like today(), but, if possible, supplies more precision than can be gotten from going through a time.time() timestamp (for example, this may be possible on platforms supplying the C gettimeofday() function).

Else tz must be an instance of a class tzinfo subclass, and the current date and time are converted to tz’s time zone. In this case the result is equivalent to tz.fromutc(datetime.utcnow().replace(tzinfo=tz)). See also today(), utcnow().

classmethod datetime.utcnow()
Return the current UTC date and time, with tzinfo None. This is like now(), but returns the current UTC date and time, as a naive datetime object. An aware current UTC datetime can be obtained by calling datetime.now(timezone.utc). See also now().

classmethod datetime.fromtimestamp(timestamp, tz=None)
Return the local date and time corresponding to the POSIX timestamp, such as is returned by time.time(). If optional argument tz is None or not specified, the timestamp is converted to the platform’s local date and time, and the returned datetime object is naive.

Else tz must be an instance of a class tzinfo subclass, and the timestamp is converted to tz’s time zone. In this case the result is equivalent to tz.fromutc(datetime.utcfromtimestamp(timestamp).replace(tzinfo=tz)). fromtimestamp() may raise OverflowError, if the timestamp is out of the range of values supported by the platform C localtime() or gmtime() functions, and OSError on localtime() or gmtime() failure. It’s common for this to be restricted to years in 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by fromtimestamp(), and then it’s possible to have two timestamps differing by a second that yield identical datetime objects. See also utcfromtimestamp(). Changed in version 3.3: Raise OverflowError instead of ValueError if the timestamp is out of the range of values supported by the platform C localtime() or gmtime() functions. Raise OSError instead of ValueError on localtime() or gmtime() failure.

classmethod datetime.utcfromtimestamp(timestamp)
Return the UTC datetime corresponding to the POSIX timestamp, with tzinfo None. This may raise OverflowError, if the timestamp is out of the range of values supported by the platform C gmtime() function, and OSError on gmtime() failure. It’s common for this to be restricted to years in 1970 through 2038. See also fromtimestamp().

On the POSIX compliant platforms, utcfromtimestamp(timestamp) is equivalent to the following expression:

datetime(1970, 1, 1) + timedelta(seconds=timestamp)

Changed in version 3.3: Raise OverflowError instead of ValueError if the timestamp is out of the range of values supported by the platform C gmtime() function. Raise OSError instead of ValueError on gmtime() failure.

classmethod datetime.fromordinal(ordinal)
Return the datetime corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. ValueError is raised unless 1 <= ordinal <= datetime.max.toordinal(). The hour, minute, second and microsecond of the result are all 0, and tzinfo is None.

classmethod datetime.combine(date, time)
Return a new datetime object whose date components are equal to the given date object’s, and whose time components and tzinfo attributes are equal to the given time object’s. For any datetime object d, d == datetime.combine(d.date(), d.timetz()). If date is a datetime object, its time components and tzinfo attributes are ignored.
classmethod datetime.strptime(date_string, format)

Return a datetime corresponding to date_string, parsed according to format. This is equivalent to
datetime(*time.strptime(date_string, format)[0:6]). ValueError is raised if
the date_string and format can’t be parsed by time.strptime() or if it returns a value which isn’t a
time tuple. For a complete list of formatting directives, see strftime() and strptime() Behavior.

Class attributes:
datetime.min
The earliest representable datetime.datetime(MINYEAR, 1, 1, tzinfo=None).
datetime.max
The latest representable datetime.datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999,
tzinfo=None).
datetime.resolution
The smallest possible difference between non-equal datetime objects,
timedelta(microseconds=1).

Instance attributes (read-only):
datetime.year
Between MINYEAR and MAXYEAR inclusive.
datetime.month
Between 1 and 12 inclusive.
datetime.day
Between 1 and the number of days in the given month of the given year.
datetime.hour
In range(24).
datetime.minute
In range(60).
datetime.second
In range(60).
datetime.microsecond
In range(1000000).
datetime.tzinfo
The object passed as the tzinfo argument to the datetime constructor, or None if none was passed.

Supported operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>datetime2 = datetimel1 + timedelta</td>
<td>(1)</td>
</tr>
<tr>
<td>datetime2 = datetimel1 - timedelta</td>
<td>(2)</td>
</tr>
<tr>
<td>timedelta = datetimel1 - datetimel2</td>
<td>(3)</td>
</tr>
<tr>
<td>datetimel1 &lt; datetimel2</td>
<td>Compares datetime to datetimel(4)</td>
</tr>
</tbody>
</table>

1. datetime2 is a duration of timedelta removed from datetimel1, moving forward in time if
timedelta.days > 0, or backward if timedelta.days < 0. The result has the same tzinfo at-
ttribute as the input datetime, and datetime2 - datetimel1 == timedelta after. OverflowError is raised if
datetimel1.year would be smaller than MINYEAR or larger than MAXYEAR. Note that no time zone adjust-
ments are done even if the input is an aware object.

2. Computes the datetime2 such that datetime2 + timedelta == datetimel1. As for addition, the result has the
same tzinfo attribute as the input datetime, and no time zone adjustments are done even if the input is
aware. This isn’t quite equivalent to datetimel1 + (-timedelta), because -timedelta in isolation can overflow
in cases where datetimel1 - timedelta does not.

3. Subtraction of a datetime from a datetime is defined only if both operands are naive, or if both are
aware. If one is aware and the other is naive, TypeError is raised.
If both are naive, or both are aware and have the same `tzinfo` attribute, the `tzinfo` attributes are ignored, and the result is a `timedelta` object \( t \) such that \( \text{datetime2} + t = \text{datetime1} \). No time zone adjustments are done in this case.

If both are aware and have different `tzinfo` attributes, \( a-b \) acts as if \( a \) and \( b \) were first converted to naive UTC datetimes first. The result is \( (a.\text{replace}(\text{tzinfo}=\text{None}) - a.\text{utcoffset}) - (b.\text{replace}(\text{tzinfo}=\text{None}) - b.\text{utcoffset}) \) except that the implementation never overflows.

4. `datetime1` is considered less than `datetime2` when `datetime1` precedes `datetime2` in time.

If one comparand is naive and the other is aware, `TypeError` is raised if an order comparison is attempted. For equality comparisons, naive instances are never equal to aware instances.

If both comparands are aware, and have the same `tzinfo` attribute, the common `tzinfo` attribute is ignored and the base datetimes are compared. If both comparands are aware and have different `tzinfo` attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from `self.utcoffset()`). Changed in version 3.3: Equality comparisons between naive and aware `datetime` instances don’t raise `TypeError`.

**Note:** In order to stop comparison from falling back to the default scheme of comparing object addresses, `datetime` comparison normally raises `TypeError` if the other comparand isn’t also a `datetime` object. However, `NotImplemented` is returned instead if the other comparand has a `timetuple()` attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a `datetime` object is compared to an object of a different type, `TypeError` is raised unless the comparison is `==` or `!=`. The latter cases return `False` or `True`, respectively.

`datetime` objects can be used as dictionary keys. In Boolean contexts, all `datetime` objects are considered to be true.

Instance methods:

`datetime.date()`
- Return `date` object with same year, month and day.

`datetime.time()`
- Return `time` object with same hour, minute, second and microsecond. `tzinfo` is `None`. See also method `timetz()`.

`datetime.timetz()`
- Return `time` object with same hour, minute, second, microsecond, and `tzinfo` attributes. See also method `time()`.

`datetime.replace(year, month, day, hour, minute, second, microsecond, tzinfo=None)`
- Return a `datetime` with the same attributes, except for those attributes given new values by whichever keyword arguments are specified. Note that `tzinfo=None` can be specified to create a naive `datetime` from an aware `datetime` with no conversion of date and time data.

`datetime.astimezone(tz=None)`
- Return a `datetime` object with new `tzinfo` attribute \( tz \), adjusting the date and time data so the result is the same UTC time as `self`, but in \( tz \)'s local time.

  If provided, `tz` must be an instance of a `tzinfo` subclass, and its `utcoffset()` and `dst()` methods must not return `None`. `self` must be aware (`self.tzinfo` must not be `None`, and `self.utcoffset()` must not return `None`).

  If called without arguments (or with `tz=None`) the system local timezone is assumed. The `tzinfo` attribute of the converted `datetime` instance will be set to an instance of `timezone` with the zone name and offset obtained from the OS.

  If `self.tzinfo` is `tz`, `self.astimezone(tz)` is equal to `self`: no adjustment of date or time data is performed. Else the result is local time in time zone `tz`, representing the same UTC time as `self`: after `astz = dt.astimezone(tz), astz - astz.utcoffset()` will usually have the same date and
time data as \( dt - dt.utcoffset() \). The discussion of class \( tzinfo \) explains the cases at Daylight Saving Time transition boundaries where this cannot be achieved (an issue only if \( tz \) models both standard and daylight time).

If you merely want to attach a time zone object \( tz \) to a datetime \( dt \) without adjustment of date and time data, use \( dt.replace(tzinfo=tz) \). If you merely want to remove the time zone object from an aware datetime \( dt \) without conversion of date and time data, use \( dt.replace(tzinfo=None) \).

Note that the default \( tzinfo.fromutc() \) method can be overridden in a \( tzinfo \) subclass to affect the result returned by \( astimezone() \). Ignoring error cases, \( astimezone() \) acts like:

```python
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new time zone object.
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz’s local time.
    return tz.fromutc(utc)
```

Changed in version 3.3: \( tz \) now can be omitted.

datetime.\texttt{utcoffset}()

If \( tzinfo \) is None, returns None, else returns \( self.tzinfo.utcoffset() \), and raises an exception if the latter doesn’t return None, or a \texttt{timedelta} object representing a whole number of minutes with magnitude less than one day.

datetime.\texttt{dst}()

If \( tzinfo \) is None, returns None, else returns \( self.tzinfo.dst() \), and raises an exception if the latter doesn’t return None, or a \texttt{timedelta} object representing a whole number of minutes with magnitude less than one day.

datetime.\texttt{tzname}()

If \( tzinfo \) is None, returns None, else returns \( self.tzinfo.tzname() \), raises an exception if the latter doesn’t return None or a string object.

datetime.\texttt{timetuple}()

Return a \texttt{time.struct_time} such as returned by \( time.localtime() \). \( d.timetuple() \) is equivalent to \( time.struct_time((d.year, d.month, d.day, d.hour, d.minute, d.second, d.weekday(), yday, dst)) \), where \( yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1 \) is the day number within the current year starting with 1 for January 1st. The \texttt{tm_isdst} flag of the result is set according to the \( dst() \) method: \texttt{tzinfo} is None or \texttt{dst()} returns None, \texttt{tm_isdst} is set to -1; else if \texttt{dst()} returns a non-zero value, \texttt{tm_isdst} is set to 1; else \texttt{tm_isdst} is set to 0.

datetime.\texttt{utctimetuple}()

If datetime instance \( d \) is naive, this is the same as \( d.timetuple() \) except that \texttt{tm_isdst} is forced to 0 regardless of what \texttt{d.dst()} returns. \( DST \) is never in effect for a UTC time.

If \( d \) is aware, \( d \) is normalized to UTC time, by subtracting \( d.utcoffset() \), and a \texttt{time.struct_time} for the normalized time is returned. \texttt{tm_isdst} is forced to 0. Note that an \texttt{OverflowError} may be raised if \( d.year \) was \texttt{MINYEAR} or \texttt{MAXYEAR} and UTC adjustment spills over a year boundary.

datetime.\texttt{toordinal}()

Return the proleptic Gregorian ordinal of the date. The same as \( self.date().toordinal() \).

datetime.\texttt{timestamp}()

Return POSIX timestamp corresponding to the datetime instance. The return value is a \texttt{float} similar to that returned by \( time.time() \).

Naive datetime instances are assumed to represent local time and this method relies on the platform \texttt{C} \texttt{mktime()} function to perform the conversion. Since datetime supports wider range of values than \texttt{mktime()} on many platforms, this method may raise \texttt{OverflowError} for times far in the past or far in the future.
For aware `datetime` instances, the return value is computed as:

```python
(dt - datetime(1970, 1, 1, tzinfo=timezone.utc)).total_seconds()
```

New in version 3.3.

**Note:** There is no method to obtain the POSIX timestamp directly from a naive `datetime` instance representing UTC time. If your application uses this convention and your system timezone is not set to UTC, you can obtain the POSIX timestamp by supplying `tzinfo=timezone.utc`:

```python
timestamp = dt.replace(tzinfo=timezone.utc).timestamp()
```

or by calculating the timestamp directly:

```python
timestamp = (dt - datetime(1970, 1, 1)) / timedelta(seconds=1)
```

datetime.**weekday**

Return the day of the week as an integer, where Monday is 0 and Sunday is 6. The same as `self.date().weekday()`. See also `isoweekday()`.

datetime.**isoweekday**

Return the day of the week as an integer, where Monday is 1 and Sunday is 7. The same as `self.date().isoweekday()`. See also `weekday()`, `isocalendar()`.

datetime.**isocalendar**

Return a 3-tuple, (ISO year, ISO week number, ISO weekday). The same as `self.date().isocalendar()`.

datetime.**isoformat** *(sep='T')*

Return a string representing the date and time in ISO 8601 format, YYYY-MM-DDTHH:MM:SS.mmmmmm or, if `microsecond` is 0, YYYY-MM-DDTHH:MM:SS

If `utcoffset()` does not return None, a 6-character string is appended, giving the UTC offset in (signed) hours and minutes: YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM or, if `microsecond` is 0 YYYY-MM-DDTHH:MM:SS+HH:MM

The optional argument `sep` (default ‘T’) is a one-character separator, placed between the date and time portions of the result. For example,

```python
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(minutes=-399)
...     ...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
```

datetime.**str**

For a `datetime` instance `d`, `str(d)` is equivalent to `d.isoformat(' ' `).

datetime.**ctime**

Return a string representing the date and time, for example `datetime(2002, 12, 4, 20, 30, 40).ctime() == ‘Wed Dec 4 20:30:40 2002’`. `d.ctime()` is equivalent to `time.ctime(time.mktime(d.timetuple()))` on platforms where the native C `ctime()` function (which `time.ctime()` invokes, but which `datetime.ctime()` does not invoke) conforms to the C standard.

datetime.**strftime** *(format)*

Return a string representing the date and time, controlled by an explicit format string. For a complete list of formatting directives, see `strftime()` and `strptime()` Behavior.
datetime.__format__(format)

Same as datetime.strftime(). This makes it possible to specify format string for a datetime object when using str.format(). For a complete list of formatting directives, see strftime() and strptime().

Behavior.

Examples of working with datetime objects:

```python
>>> from datetime import datetime, date, time

# Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)

# Using datetime.now() or datetime.utcnow()
>>> datetime.datetime(2005, 7, 14, 12, 30)
>>> datetime.datetime.now()
datetime.datetime(2005, 7, 14, 12, 30)
>>> datetime.datetime.utcnow()
datetime.datetime(2005, 7, 14, 11, 30)

# Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)

# Using datetime.timetuple() to get tuple of all attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print(it)
...     2006 # year
     11 # month
     21 # day
     16 # hour
     30 # minute
     0 # second
     1 # weekday (0 = Monday)
     325 # number of days since 1st January
     -1 # dst - method tzinfo.dst() returned None

# Date in ISO format
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print(it)
...     2006 # ISO year
     47 # ISO week
     2 # ISO weekday

# Formatting datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}''.format(dt, "day", "month", "time")
'The day is 21, the month is November, the time is 04:30PM.'

Using datetime with tzinfo:

```
d = datetime(dt.year, 11, 1)
self.dstoff = d - timedelta(days=d.weekday() + 1)
if self.dston <= dt.replace(tzinfo=None) < self.dstoff:
    return timedelta(hours=1)
else:
    return timedelta(0)
def tzname(self,dt):
    return "GMT +1"

>>> class GMT2(tzinfo):
    def utcoffset(self, dt):
        return timedelta(hours=2) + self.dst(dt)
    def dst(self, dt):
        d = datetime(dt.year, 4, 1)
        self.dston = d - timedelta(days=d.weekday() + 1)
        d = datetime(dt.year, 11, 1)
        self.dstoff = d - timedelta(days=d.weekday() + 1)
        if self.dston <= dt.replace(tzinfo=None) < self.dstoff:
            return timedelta(hours=1)
        else:
            return timedelta(0)
def tzname(self,dt):
    return "GMT +2"

>>> gmt1 = GMT1()
>>> # Daylight Saving Time
>>> dt1 = datetime(2006, 11, 21, 16, 30, tzinfo=gmt1)
>>> dt1.dst()  # The Daylight Saving adjustment
datetime.timedelta(0)
>>> dt1.utcoffset()  # The time zone offset
datetime.timedelta(0, 3600)
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=gmt1)
>>> dt2.dst()  # No Daylight Saving adjustment
datetime.timedelta(0, 7200)
>>> dt2.utcoffset()  # The time zone offset
datetime.timedelta(0, 7200)
>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(GMT2())
>>> print(dt3)
datetime.datetime(2006, 6, 14, 14, 0, tzinfo=<GMT2 object at 0x...>)
>>> dt3.utctimetuple() == dt2.utctimetuple()
True

8.1.5 time Objects

A time object represents a (local) time of day, independent of any particular day, and subject to adjustment via a tzinfo object.

class datetime.time(hour=0, minute=0, second=0, microsecond=0, tzinfo=None)

All arguments are optional. tzinfo may be None, or an instance of a tzinfo subclass. The remaining arguments may be integers, in the following ranges:

- 0 <= hour < 24
- 0 <= minute < 60
- 0 <= second < 60

8.1. datetime — Basic date and time types 151
• \(0 \leq \text{microsecond} < 1000000\).

If an argument outside those ranges is given, \texttt{ValueError} is raised. All default to 0 except \texttt{tzinfo}, which defaults to \texttt{None}.

Class attributes:

\texttt{time.min}

The earliest representable time, \texttt{time(0, 0, 0, 0)}.

\texttt{time.max}

The latest representable time, \texttt{time(23, 59, 59, 999999)}.

\texttt{time.resolution}

The smallest possible difference between non-equal time objects, \texttt{timedelta(microseconds=1)}, although note that arithmetic on time objects is not supported.

Instance attributes (read-only):

\texttt{time.hour}

In range(24).

\texttt{time.minute}

In range(60).

\texttt{time.second}

In range(60).

\texttt{time.microsecond}

In range(1000000).

\texttt{time.tzinfo}

The object passed as the tzinfo argument to the \texttt{time} constructor, or \texttt{None} if none was passed.

Supported operations:

• comparison of \texttt{time} to \texttt{time}, where \(a\) is considered less than \(b\) when \(a\) precedes \(b\) in time. If one comparator is naive and the other is aware, \texttt{TypeError} is raised if an order comparison is attempted. For equality comparisons, naive instances are never equal to aware instances.

If both comparands are aware, and have the same \texttt{tzinfo} attribute, the common \texttt{tzinfo} attribute is ignored and the base times are compared. If both comparands are aware and have different \texttt{tzinfo} attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from \texttt{self.utcoffset()}). In order to stop mixed-type comparisons from falling back to the default comparison by object address, when a \texttt{time} object is compared to an object of a different type, \texttt{TypeError} is raised unless the comparison is == or !=. The latter cases return \texttt{False} or \texttt{True}, respectively. Changed in version 3.3: Equality comparisons between naive and aware \texttt{time} instances don’t raise \texttt{TypeError}.

• hash, use as dict key

• efficient pickling

• in Boolean contexts, a \texttt{time} object is considered to be true if and only if, after converting it to minutes and subtracting \texttt{utcoffset()} (or 0 if that’s \texttt{None}), the result is non-zero.

Instance methods:

\texttt{time.replace([hour[, minute[, second[, microsecond[, tzinfo]]]]])}

Return a \texttt{time} with the same value, except for those attributes given new values by whichever keyword arguments are specified. Note that \texttt{tzinfo=None} can be specified to create a naive \texttt{time} from an aware \texttt{time}, without conversion of the time data.

\texttt{time.isoformat()}

Return a string representing the time in ISO 8601 format, HH:MM:SS.mmmmm or, if self.microsecond is 0, HH:MM:SS If \texttt{utcoffset()} does not return \texttt{None}, a 6-character string is appended, giving the UTC offset in (signed) hours and minutes: HH:MM:SS.mmmmm+HH:MM or, if self.microsecond is 0, HH:MM:SS+HH:MM
The Python Library Reference, Release 3.3.3

```python
def __str__(self):
    return t.isoformat()

def strftime(self, format):
    return t.isoformat() + '00'

def __format__(self, format):
    return t.isoformat() + '00'

def utcoffset(self):
    return self.tzinfo.utcoffset() + '00'

def dst(self):
    return self.tzinfo.dst() + '00'

def tzname(self):
    return self.tzinfo.tzname() + '00'
```

Example:
```python
>>> from datetime import time, tzinfo
>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self, dt):
...         return "Europe/Prague"
...
>>> t = time(12, 10, 30, tzinfo=GMT1())
>>> t
datetime.time(12, 10, 30, tzinfo=<GMT1 object at 0x...>)
>>> gmt = GMT1()
>>> t.isoformat()
'12:10:30+01:00'
>>> t dst()
datetime.timedelta(0)
>>> t.tzname()
'Europe/Prague'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 Europe/Prague'
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'
```

8.1.6 tzinfo Objects

tzinfo is an abstract base class, meaning that this class should not be instantiated directly. You need to derive a concrete subclass, and (at least) supply implementations of the standard tzinfo methods needed by the datetime methods you use. The datetime module supplies a simple concrete subclass of timezone which can represent timezones with fixed offset from UTC such as UTC itself or North American EST and EDT.

An instance of (a concrete subclass of) tzinfo can be passed to the constructors for datetime and time objects. The latter objects view their attributes as being in local time, and the tzinfo object supports methods
revealing offset of local time from UTC, the name of the time zone, and DST offset, all relative to a date or time object passed to them.

Special requirement for pickling: A `tzinfo` subclass must have an `__init__()` method that can be called with no arguments, else it can be pickled but possibly not unpickled again. This is a technical requirement that may be relaxed in the future.

A concrete subclass of `tzinfo` may need to implement the following methods. Exactly which methods are needed depends on the uses made of aware `datetime` objects. If in doubt, simply implement all of them.

**tzinfo.**`utcoffset(dt)`
Return offset of local time from UTC, in minutes east of UTC. If local time is west of UTC, this should be negative. Note that this is intended to be the total offset from UTC; for example, if a `tzinfo` object represents both time zone and DST adjustments, `utcoffset()` should return their sum. If the UTC offset isn’t known, return `None`. Else the value returned must be a `timedelta` object specifying a whole number of minutes in the range -1439 to 1439 inclusive (1440 = 24*60; the magnitude of the offset must be less than one day). Most implementations of `utcoffset()` will probably look like one of these two:

```python
return CONSTANT  # fixed-offset class
return CONSTANT + self.dst(dt)  # daylight-aware class
```

If `utcoffset()` does not return `None`, `dst()` should not return `None` either.

The default implementation of `utcoffset()` raises `NotImplementedError`.

**tzinfo.**`dst(dt)`
Return the daylight saving time (DST) adjustment, in minutes east of UTC, or `None` if DST information isn’t known. Return `timedelta(0)` if DST is not in effect. If DST is in effect, return the offset as a `timedelta` object (see `utcoffset()` for details). Note that DST offset, if applicable, has already been added to the UTC offset returned by `utcoffset()`, so there’s no need to consult `dst()` unless you’re interested in obtaining DST info separately. For example, `datetime.timetuple()` calls its `tzinfo` attribute’s `dst()` method to determine how the `tm_isdst` flag should be set, and `tzinfo.fromutc()` calls `dst()` to account for DST changes when crossing time zones.

An instance `tz` of a `tzinfo` subclass that models both standard and daylight times must be consistent in this sense:

```python
tz.utcoffset(dt) - tz.dst(dt)
```

must return the same result for every `datetime` `dt` with `dt.tzinfo == tz` For sane `tzinfo` subclasses, this expression yields the time zone’s “standard offset”, which should not depend on the date or the time, but only on geographic location. The implementation of `datetime.astimezone()` relies on this, but cannot detect violations; it’s the programmer’s responsibility to ensure it. If a `tzinfo` subclass cannot guarantee this, it may be able to override the default implementation of `tzinfo.fromutc()` to work correctly with `astimezone()` regardless.

Most implementations of `dst()` will probably look like one of these two:

```python
def dst(self, dt):
    # a fixed-offset class: doesn’t account for DST
    return timedelta(0)

or

def dst(self, dt):
    # Code to set dston and dstoff to the time zone’s DST
    # transition times based on the input dt.year, and expressed
    # in standard local time. Then
    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```
The default implementation of `dst()` raises `NotImplementedError`.

```python
tzinfo.tzname(dt)
```

Return the time zone name corresponding to the `datetime` object `dt`, as a string. Nothing about string names is defined by the `datetime` module, and there’s no requirement that it mean anything in particular. For example, “GMT”, “UTC”, “-500”, “-5:00”, “EDT”, “US/Eastern”, “America/New York” are all valid replies. Return `None` if a string name isn’t known. Note that this is a method rather than a fixed string primarily because some `tzinfo` subclasses will wish to return different names depending on the specific value of `dt` passed, especially if the `tzinfo` class is accounting for daylight time.

The default implementation of `tzname()` raises `NotImplementedError`.

These methods are called by a `datetime` or `time` object, in response to their methods of the same names. A `datetime` object passes itself as the argument, and a `time` object passes `None` as the argument. A `tzinfo` subclass’s methods should therefore be prepared to accept a `dt` argument of `None`, or of class `datetime`.

When `None` is passed, it’s up to the class designer to decide the best response. For example, returning `None` is appropriate if the class wishes to say that time objects don’t participate in the `tzinfo` protocols. It may be more useful for `utcoffset(None)` to return the standard UTC offset, as there is no other convention for discovering the standard offset.

When a `datetime` object is passed in response to a `datetime` method, `dt.tzinfo` is the same object as `self.tzinfo` methods can rely on this, unless user code calls `tzinfo` methods directly. The intent is that the `tzinfo` methods interpret `dt` as being in local time, and not need worry about objects in other timezones.

There is one more `tzinfo` method that a subclass may wish to override:

```python
tzinfo.fromutc(dt)
```

This is called from the default `datetime.astimezone()` implementation. When called from that, `dt.tzinfo` is `self`, and `dt`’s date and time data are to be viewed as expressing a UTC time. The purpose of `fromutc()` is to adjust the date and time data, returning an equivalent `datetime` in `self`’s local time.

Most `tzinfo` subclasses should be able to inherit the default `fromutc()` implementation without problems. It’s strong enough to handle fixed-offset time zones, and time zones accounting for both standard and daylight time, and the latter even if the DST transition times differ in different years. An example of a time zone the default `fromutc()` implementation may not handle correctly in all cases is one where the standard offset (from UTC) depends on the specific date and time passed, which can happen for political reasons. The default implementations of `astimezone()` and `fromutc()` may not produce the result you want if the result is one of the hours straddling the moment the standard offset changes.

Skipping code for error cases, the default `fromutc()` implementation acts like:

```python
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst  # this is self's standard offset
    if delta:
        dt += delta       # convert to standard local time
        dtdst = dt.dst()  # raise ValueError if dtdst is None
        if dtdst:
            return dt + dtdst
    else:
        return dt
```

Example `tzinfo` classes:

```python
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)
```

---

8.1. `datetime` — Basic date and time types 155
# A UTC class.

class UTC(tzinfo):
    """UTC""
    def utcoffset(self, dt):
        return ZERO
    def tzname(self, dt):
        return "UTC"
    def dst(self, dt):
        return ZERO
utc = UTC()

# A class building tzinfo objects for fixed-offset time zones.
# Note that FixedOffset(0, "UTC") is a different way to build a
# UTC tzinfo object.

class FixedOffset(tzinfo):
    """Fixed offset in minutes east from UTC.""
    def __init__(self, offset, name):
        self.__offset = timedelta(minutes=offset)
        self.__name = name
    def utcoffset(self, dt):
        return self.__offset
    def tzname(self, dt):
        return self.__name
    def dst(self, dt):
        return ZERO

# A class capturing the platform’s idea of local time.

import time as _time
STDOFFSET = timedelta(seconds = _time.timezone)
if _time.daylight:
    DSTOFFSET = timedelta(seconds = _time.altzone)
else:
    DSTOFFSET = STDOFFSET
DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):
    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET
    def dst(self, dt):
if self._isdst(dt):
    return DSTDIFF
else:
    return ZERO

def tzname(self, dt):
    return _time.tzname[self._isdst(dt)]

def _isdst(self, dt):
    tt = (dt.year, dt.month, dt.day,
         dt.hour, dt.minute, dt.second,
         dt.weekday(), 0, 0)
    stamp = _time.mktime(tt)
    tt = _time.localtime(stamp)
    return tt.tm_isdst > 0

Local = LocalTimezone()

# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt

# US DST Rules
# # This is a simplified (i.e., wrong for a few cases) set of rules for US
# # DST start and end times. For a complete and up-to-date set of DST rules
# # and timezone definitions, visit the Olson Database (or try pytz):
# # http://www.twinsun.com/tz/tz-link.htm
# # http://sourceforge.net/projects/pytz/ (might not be up-to-date)
# # In the US, since 2007, DST starts at 2am (standard time) on the second
# # Sunday in March, which is the first Sunday on or after Mar 8.
# DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time; 1am standard time) on the first Sunday of Nov.
# DSTEND_2007 = datetime(1, 11, 1, 1)
# # From 1987 to 2006, DST used to start at 2am (standard time) on the first
# # Sunday in April and to end at 2am (DST time; 1am standard time) on the last
# # Sunday of October, which is the first Sunday on or after Oct 25.
# DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
# DSTEND_1987_2006 = datetime(1, 10, 25, 1)
# # From 1967 to 1986, DST used to start at 2am (standard time) on the last
# # Sunday in April (the one on or after April 24) and to end at 2am (DST time;
# # 1am standard time) on the last Sunday of October, which is the first Sunday
# # on or after Oct 25.
# DSTSTART_1967_1986 = datetime(1, 4, 24, 2)

class USTimeZone(tzinfo):
    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
The Python Library Reference, Release 3.3.3

self.stdname = stdname
self.dstname = dstname

def __repr__(self):
    return self.reprname
def tzname(self, dt):
    if self.dst(dt):
        return self.dstname
    else:
        return self.stdname
def utcoffset(self, dt):
    return self.stdoffset + self.dst(dt)
def dst(self, dt):
    if dt is None or dt.tzinfo is None:
        # An exception may be sensible here, in one or both cases.
        # It depends on how you want to treat them. The default
        # fromutc() implementation (called by the default astimezone()
        # implementation) passes a datetime with dt.tzinfo is self.
        return ZERO
    assert dt.tzinfo is self
    # Find start and end times for US DST. For years before 1967, return
    # ZERO for no DST.
    if 2006 < dt.year:
        dststart, dstend = DSTSTART_2007, DSTEND_2007
    elif 1986 < dt.year < 2007:
    elif 1966 < dt.year < 1987:
    else:
        return ZERO
    start = first_sunday_on_or_after(dststart.replace(year=dt.year))
    end = first_sunday_on_or_after(dstend.replace(year=dt.year))

    # Can’t compare naive to aware objects, so strip the timezone from
    # dt first.
    if start <= dt.replace(tzinfo=None) < end:
        return HOUR
    else:
        return ZERO

Eastern = USTimeZone(-5, "Eastern", "EST", "EDT")
Central = USTimeZone(-6, "Central", "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific = USTimeZone(-8, "Pacific", "PST", "PDT")

Note that there are unavoidable subtleties twice per year in a tzinfo subclass accounting for both standard and
daylight time, at the DST transition points. For concreteness, consider US Eastern (UTC -0500), where EDT
begins the minute after 1:59 (EST) on the second Sunday in March, and ends the minute after 1:59 (EDT) on the
first Sunday in November:

When DST starts (the “start” line), the local wall clock leaps from 1:59 to 3:00. A wall time of the form 2:MM doesn’t really make sense on that day, so \texttt{astimezone(Eastern)} won’t deliver a result with \texttt{hour == 2} on the day DST begins. In order for \texttt{astimezone()} to make this guarantee, the \texttt{tzinfo.dst()} method must consider times in the “missing hour” (2:MM for Eastern) to be in daylight time.

When DST ends (the “end” line), there’s a potentially worse problem: there’s an hour that can’t be spelled unambiguously in local wall time: the last hour of daylight time. In Eastern, that’s times of the form 5:MM UTC on the day daylight time ends. The local wall clock leaps from 1:59 (daylight time) back to 1:00 (standard time) again. Local times of the form 1:MM are ambiguous. \texttt{astimezone()} mimics the local clock’s behavior by mapping two adjacent UTC hours into the same local hour then. In the Eastern example, UTC times of the form 5:MM and 6:MM both map to 1:MM when converted to Eastern. In order for \texttt{astimezone()} to make this guarantee, the \texttt{tzinfo.dst()} method must consider times in the “repeated hour” to be in standard time. This is easily arranged, as in the example, by expressing DST switch times in the time zone’s standard local time.

Applications that can’t bear such ambiguities should avoid using hybrid \texttt{tzinfo} subclasses; there are no ambiguities when using \texttt{timezone}, or any other fixed-offset \texttt{tzinfo} subclass (such as a class representing only EST (fixed offset -5 hours), or only EDT (fixed offset -4 hours)).

See Also:

\texttt{pytz} The standard library has no \texttt{tzinfo} instances except for UTC, but there exists a third-party library which brings the IANA timezone database (also known as the Olson database) to Python: \texttt{pytz}.

\texttt{pytz} contains up-to-date information and its usage is recommended.

\texttt{IANA timezone database} The Time Zone Database (often called tz or zoneinfo) contains code and data that represent the history of local time for many representative locations around the globe. It is updated periodically to reflect changes made by political bodies to time zone boundaries, UTC offsets, and daylight-saving rules.

\section*{8.1.7 timezone Objects}

The \texttt{timezone} class is a subclass of \texttt{tzinfo}, each instance of which represents a timezone defined by a fixed offset from UTC. Note that objects of this class cannot be used to represent timezone information in the locations where different offsets are used in different days of the year or where historical changes have been made to civil time.

\begin{verbatim}
class datetime.timezone(offset[, name])
The offset argument must be specified as a timedelta object representing the difference between the local time and UTC. It must be strictly between \texttt{-timedelta(hours=24)} and \texttt{timedelta(hours=24)} and represent a whole number of minutes, otherwise ValueError is raised.

The name argument is optional. If specified it must be a string that is used as the value returned by the tzname(dt) method. Otherwise, tzname(dt) returns a string ‘UTCsHH:MM’, where s is the sign of offset, HH and MM are two digits of offset.hours and offset.minutes respectively.

timezone.utcoffset(dt) Return the fixed value specified when the timezone instance is constructed. The dt argument is ignored. The return value is a timedelta instance equal to the difference between the local time and UTC.

timezone.tzname(dt) Return the fixed value specified when the timezone instance is constructed or a string ‘UTCsHH:MM’, where s is the sign of offset, HH and MM are two digits of offset.hours and offset.minutes respectively.

timezone.dst(dt) Always returns None.

timezone.fromutc(dt) Return dt + offset. The dt argument must be an aware datetime instance, with tzinfo set to self.
\end{verbatim}

8.1. \texttt{datetime} — Basic date and time types
Class attributes:

timezone.utc

    The UTC timezone, timezone(timedelta(0)).

8.1.8 strftime() and strptime() Behavior

date, datetime, and time objects all support a strftime(format) method, to create a string representing
the time under the control of an explicit format string. Broadly speaking, d.strftime(fmt) acts like the time
module's time.strftime(fmt, d.timetuple()) although not all objects support a timetuple() method.

Conversely, the datetime.strptime() class method creates a datetime object from a string representing
a date and time and a corresponding format string. datetime.strptime(date_string, format) is
equivalent to datetime(*(time.strptime(date_string, format)[0:6])).

For time objects, the format codes for year, month, and day should not be used, as time objects have no such
values. If they're used anyway, 1900 is substituted for the year, and 1 for the month and day.

For date objects, the format codes for hours, minutes, seconds, and microseconds should not be used, as date
objects have no such values. If they're used anyway, 0 is substituted for them.

The full set of format codes supported varies across platforms, because Python calls the platform C library’s
strftime() function, and platform variations are common. To see the full set of format codes supported on
your platform, consult the strftime(3) documentation.

The following is a list of all the format codes that the C standard (1989 version) requires, and these work on all
platforms with a standard C implementation. Note that the 1999 version of the C standard added additional format
codes.
<table>
<thead>
<tr>
<th>Directive</th>
<th>Meaning</th>
<th>Example</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Weekday as locale’s abbreviated name.</td>
<td>Sun, Mon, ..., Sat (en_US); So, Mo, ..., Sa (de_DE)</td>
<td>(1)</td>
</tr>
<tr>
<td>%A</td>
<td>Weekday as locale’s full name.</td>
<td>Sunday, Monday, ..., Saturday (en_US); Sonntag, Montag, ..., Samstag (de_DE)</td>
<td>(1)</td>
</tr>
<tr>
<td>%w</td>
<td>Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.</td>
<td>0, 1, ..., 6</td>
<td></td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month as a zero-padded decimal number.</td>
<td>01, 02, ..., 31</td>
<td>(1)</td>
</tr>
<tr>
<td>%b</td>
<td>Month as locale’s abbreviated name.</td>
<td>Jan, Feb, ..., Dec (en_US); Jan, Feb, ..., Dez (de_DE)</td>
<td></td>
</tr>
<tr>
<td>%B</td>
<td>Month as locale’s full name.</td>
<td>January, February, ..., December (en_US); Januar, Februar, ..., Dezember (de_DE)</td>
<td>(1)</td>
</tr>
<tr>
<td>%m</td>
<td>Month as a zero-padded decimal number.</td>
<td>01, 02, ..., 12</td>
<td></td>
</tr>
<tr>
<td>%y</td>
<td>Year without century as a zero-padded decimal number.</td>
<td>00, 01, ..., 99</td>
<td></td>
</tr>
<tr>
<td>%Y</td>
<td>Year with century as a decimal number.</td>
<td>0001, 0002, ..., 2013, 2014, ..., 9998, 9999</td>
<td>(2)</td>
</tr>
<tr>
<td>%H</td>
<td>Hour (24-hour clock) as a zero-padded decimal number.</td>
<td>00, 01, ..., 23</td>
<td></td>
</tr>
<tr>
<td>%I</td>
<td>Hour (12-hour clock) as a zero-padded decimal number.</td>
<td>01, 02, ..., 12</td>
<td></td>
</tr>
<tr>
<td>%p</td>
<td>Locale’s equivalent of either AM or PM.</td>
<td>AM, PM (en_US); am, pm (de_DE)</td>
<td>(1), (3)</td>
</tr>
<tr>
<td>%M</td>
<td>Minute as a zero-padded decimal number.</td>
<td>00, 01, ..., 59</td>
<td></td>
</tr>
<tr>
<td>%S</td>
<td>Second as a zero-padded decimal number.</td>
<td>00, 01, ..., 59</td>
<td>(4)</td>
</tr>
<tr>
<td>%f</td>
<td>Microsecond as a decimal number, zero-padded on the left.</td>
<td>000000, 000001, ..., 999999</td>
<td>(5)</td>
</tr>
<tr>
<td>%z</td>
<td>UTC offset in the form +HHMM or -HHMM (empty string if the object is naive).</td>
<td>(empty), +0000, -0400, +1030</td>
<td>(6)</td>
</tr>
<tr>
<td>%j</td>
<td>Day of the year as a zero-padded decimal number.</td>
<td>001, 002, ..., 366</td>
<td></td>
</tr>
</tbody>
</table>
Notes:

1. Because the format depends on the current locale, care should be taken when making assumptions about the output value. Field orderings will vary (for example, “month/day/year” versus “day/month/year”), and the output may contain Unicode characters encoded using the locale’s default encoding (for example, if the current locale is jps_JP, the default encoding could be any one of eucJP, SJIS, or utf-8; use \texttt{locale.getlocale()} to determine the current locale’s encoding).

2. The \texttt{strptime()} method can parse years in the full [1, 9999] range, but years < 1000 must be zero-filled to 4-digit width. Changed in version 3.2: In previous versions, \texttt{strftime()} method was restricted to years \geq 1900. Changed in version 3.3: In version 3.2, \texttt{strftime()} method was restricted to years \geq 1000.

3. When used with the \texttt{strptime()} method, the \%p directive only affects the output hour field if the \%I directive is used to parse the hour.

4. Unlike the \texttt{time} module, the \texttt{datetime} module does not support leap seconds.

5. When used with the \texttt{strptime()} method, the \%f directive accepts from one to six digits and zero pads on the right. \%f is an extension to the set of format characters in the C standard (but implemented separately in \texttt{datetime} objects, and therefore always available).

6. For a naive object, the \%z and \%Z format codes are replaced by empty strings.

   For an aware object:
   
   \%z \texttt{utcoffset()} is transformed into a 5-character string of the form +HHMM or -HHMM, where HH is a 2-digit string giving the number of UTC offset hours, and MM is a 2-digit string giving the number of UTC offset minutes. For example, if \texttt{utcoffset()} returns \texttt{timedelta(hours=-3, minutes=-30)}, \%z is replaced with the string ‘-0330’.

   \%Z If \texttt{tzname()} returns None, \%z is replaced by an empty string. Otherwise \%z is replaced by the returned value, which must be a string.

   Changed in version 3.2: When the \%z directive is provided to the \texttt{strptime()} method, an aware \texttt{datetime} object will be produced. The \texttt{tzinfo} of the result will be set to a \texttt{timezone} instance.

7. When used with the \texttt{strptime()} method, \%U and \%W are only used in calculations when the day of the week and the year are specified.

### 8.2 \texttt{calendar} — General calendar-related functions

\textbf{Source code:} Lib/calendar.py

This module allows you to output calendars like the Unix \texttt{cal} program, and provides additional useful functions related to the calendar. By default, these calendars have Monday as the first day of the week, and Sunday as the last (the European convention). Use \texttt{setfirstweekday()} to set the first day of the week to Sunday (6) or to any other weekday. Parameters that specify dates are given as integers. For related functionality, see also the \texttt{datetime} and \texttt{time} modules.

Most of these functions and classes rely on the \texttt{datetime} module which uses an idealized calendar, the current Gregorian calendar extended in both directions. This matches the definition of the “proleptic Gregorian” calendar in Dershowitz and Reingold’s book “Calendrical Calculations”, where it’s the base calendar for all computations.

\textbf{class} \texttt{calendar.Calendar} (\texttt{firstweekday=0})

 Creates a \texttt{Calendar} object. \texttt{firstweekday} is an integer specifying the first day of the week. 0 is Monday (the default), 6 is Sunday.

A \texttt{Calendar} object provides several methods that can be used for preparing the calendar data for formatting. This class doesn’t do any formatting itself. This is the job of subclasses.

\texttt{Calendar} instances have the following methods:
iterweekdays()
Return an iterator for the week day numbers that will be used for one week. The first value from the
iterator will be the same as the value of the firstweekday property.

itermonthdates (year, month)
Return an iterator for the month month (1-12) in the year year. This iterator will return all days (as
datetime.date objects) for the month and all days before the start of the month or after the end
of the month that are required to get a complete week.

itermonthdays (year, month)
Return an iterator for the month month in the year year similar to itermonthdates(). Days
returned will be tuples consisting of a day number and a week day number.

monthdatescalendar (year, month)
Return a list of the weeks in the month month of the year as full weeks. Weeks are lists of seven
datetime.date objects.

monthdays2calendar (year, month)
Return a list of the weeks in the month month of the year as full weeks. Weeks are lists of seven tuples
of day numbers and weekday numbers.

monthdayscalendar (year, month)
Return a list of the weeks in the month month of the year as full weeks. Weeks are lists of seven day
numbers.

yeardatescalendar (year, width=3)
Return the data for the specified year ready for formatting. The return value is a list of month rows.
Each month row contains up to width months (defaulting to 3). Each month contains between 4 and 6
weeks and each week contains 1–7 days. Days are datetime.date objects.

yeardays2calendar (year, width=3)
Return the data for the specified year ready for formatting (similar to yeardatescalendar()). Entries in the week lists are tuples of day numbers and weekday numbers. Day numbers outside this
month are zero.

yeardayscalendar (year, width=3)
Return the data for the specified year ready for formatting (similar to yeardatescalendar()). Entries in the week lists are day numbers. Day numbers outside this month are zero.

class calendar.TextCalendar (firstweekday=0)
This class can be used to generate plain text calendars.

TextCalendar instances have the following methods:

formatmonth (theyear, themonth, w=0, l=0)
Return a month’s calendar in a multi-line string. If w is provided, it specifies the width of the date
columns, which are centered. If l is given, it specifies the number of lines that each week will use.
Depends on the first weekday as specified in the constructor or set by the setfirstweekday() method.

prmonth (theyear, themonth, w=0, l=0)
Print a month’s calendar as returned by formatmonth().

formatyear (theyear, w=2, l=1, c=6, m=3)
Return a m-column calendar for an entire year as a multi-line string. Optional parameters w, l, and c
are for date column width, lines per week, and number of spaces between month columns, respectively.
Depends on the first weekday as specified in the constructor or set by the setfirstweekday() method. The earliest year for which a calendar can be generated is platform-dependent.

pryear (theyear, w=2, l=1, c=6, m=3)
Print the calendar for an entire year as returned by formatyear().
class calendar.HTMLCalendar (firstweekday=0)
This class can be used to generate HTML calendars.

HTMLCalendar instances have the following methods:

formatmonth (theyear, themonth, withyear=True)
Return a month’s calendar as an HTML table. If withyear is true the year will be included in
the header, otherwise just the month name will be used.

formatyear (theyear, width=3)
Return a year’s calendar as an HTML table. width (defaulting to 3) specifies the number of months
per row.

formatyearpage (theyear, width=3, css='calendar.css', encoding=None)
Return a year’s calendar as a complete HTML page. width (defaulting to 3) specifies the number of
months per row. css is the name for the cascading style sheet to be used. None can be passed if no
style sheet should be used. encoding specifies the encoding to be used for the output (defaulting to
the system default encoding).

class calendar.LocaleTextCalendar (firstweekday=0, locale=None)
This subclass of TextCalendar can be passed a locale name in the constructor and will return month
and weekday names in the specified locale. If this locale includes an encoding all strings containing month
and weekday names will be returned as unicode.

class calendar.LocaleHTMLCalendar (firstweekday=0, locale=None)
This subclass of HTMLCalendar can be passed a locale name in the constructor and will return month
and weekday names in the specified locale. If this locale includes an encoding all strings containing month
and weekday names will be returned as unicode.

Note: The formatweekday() and formatmonthname() methods of these two classes temporarily change
the current locale to the given locale. Because the current locale is a process-wide setting, they are not thread-safe.

For simple text calendars this module provides the following functions.

calendar.setfirstweekday (weekday)
Sets the weekday (0 is Monday, 6 is Sunday) to start each week. The values MONDAY, TUESDAY,
WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, and SUNDAY are provided for convenience. For ex-
ample, to set the first weekday to Sunday:

import calendar
calendar.setfirstweekday(calendar.SUNDAY)

calendar.firstweekday ()
Returns the current setting for the weekday to start each week.

calendar.isleap (year)
Returns True if year is a leap year, otherwise False.

calendar.leapdays (y1, y2)
Returns the number of leap years in the range from y1 to y2 (exclusive), where y1 and y2 are years.
This function works for ranges spanning a century change.

calendar.weekday (year, month, day)
Returns the day of the week (0 is Monday) for year (1970–...), month (1–12), day (1–31).

calendar.weekheader (n)
Return a header containing abbreviated weekday names. n specifies the width in characters for one weekday.

calendar.monthrange (year, month)
Returns weekday of first day of the month and number of days in month, for the specified year and month.
The Python Library Reference, Release 3.3.3

calendar.monthcalendar (year, month)
Returns a matrix representing a month’s calendar. Each row represents a week; days outside of the month are represented by zeros. Each week begins with Monday unless set by setfirstweekday().

calendar.prmonth (theyear, themonth, w=0, l=0)
Prints a month’s calendar as returned by month().

calendar.month (theyear, themonth, w=0, l=0)
Returns a month’s calendar in a multi-line string using the formatmonth() of the TextCalendar class.

calendar.prcal (year, w=0, l=0, c=6, m=3)
Prints the calendar for an entire year as returned by calendar().

calendar.calendar (year, w=2, l=1, c=6, m=3)
Returns a 3-column calendar for an entire year as a multi-line string using the formatyear() of the TextCalendar class.

calendar.timegm (tuple)
An unrelated but handy function that takes a time tuple such as returned by the gmtime() function in the time module, and returns the corresponding Unix timestamp value, assuming an epoch of 1970, and the POSIX encoding. In fact, time.gmtime() and timegm() are each others’ inverse.

The calendar module exports the following data attributes:

calendar.day_name
An array that represents the days of the week in the current locale.

calendar.day_abbr
An array that represents the abbreviated days of the week in the current locale.

calendar.month_name
An array that represents the months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and month_name[0] is the empty string.

calendar.month_abbr
An array that represents the abbreviated months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and month_abbr[0] is the empty string.

See Also:
Module datetime Object-oriented interface to dates and times with similar functionality to the time module.
Module time Low-level time related functions.

8.3 collections — Container datatypes

Source code: Lib/collections/__init__.py

This module implements specialized container datatypes providing alternatives to Python’s general purpose built-in containers, dict, list, set, and tuple.

| namedtuple() | factory function for creating tuple subclasses with named fields |
| deque | list-like container with fast appends and pops on either end |
| ChainMap | dict-like class for creating a single view of multiple mappings |
| Counter | dict subclass for counting hashable objects |
| OrderedDict | dict subclass that remembers the order entries were added |
| defaultdict | dict subclass that calls a factory function to supply missing values |
| UserDict | wrapper around dictionary objects for easier dict subclassing |
| UserList | wrapper around list objects for easier list subclassing |
| UserString | wrapper around string objects for easier string subclassing |
Changed in version 3.3: Moved *Collections Abstract Base Classes* to the *collections.abc* module. For backwards compatibility, they continue to be visible in this module as well.

## 8.3.1 ChainMap objects

New in version 3.3. A *ChainMap* class is provided for quickly linking a number of mappings so they can be treated as a single unit. It is often much faster than creating a new dictionary and running multiple *update()* calls.

The class can be used to simulate nested scopes and is useful in templating.

```python
class collections.ChainMap(*maps)

A ChainMap groups multiple dicts or other mappings together to create a single, updateable view. If no maps are specified, a single empty dictionary is provided so that a new chain always has at least one mapping.

The underlying mappings are stored in a list. That list is public and can accessed or updated using the maps attribute. There is no other state.

Lookups search the underlying mappings successively until a key is found. In contrast, writes, updates, and deletions only operate on the first mapping.

A ChainMap incorporates the underlying mappings by reference. So, if one of the underlying mappings gets updated, those changes will be reflected in ChainMap.

All of the usual dictionary methods are supported. In addition, there is a maps attribute, a method for creating new subcontexts, and a property for accessing all but the first mapping:

```
maps
```

A user updateable list of mappings. The list is ordered from first-searched to last-searched. It is the only stored state and can be modified to change which mappings are searched. The list should always contain at least one mapping.

```
new_child()
```

Returns a new ChainMap containing a new dict followed by all of the maps in the current instance. A call to d.new_child() is equivalent to: ChainMap({}, *d.maps). This method is used for creating subcontexts that can be updated without altering values in any of the parent mappings.

```
parents
```

Property returning a new ChainMap containing all of the maps in the current instance except the first one. This is useful for skipping the first map in the search. Use cases are similar to those for the nonlocal keyword used in nested scopes. The use cases also parallel those for the built-in super() function. A reference to d.parents is equivalent to: ChainMap(*d.maps[1:]).

**See Also:**

- The MultiContext class in the Enthought CodeTools package has options to support writing to any mapping in the chain.
- Django’s Context class for templating is a read-only chain of mappings. It also features pushing and popping of contexts similar to the new_child() method and the parents() property.
- The Nested Contexts recipe has options to control whether writes and other mutations apply only to the first mapping or to any mapping in the chain.
- A greatly simplified read-only version of Chainmap.

### ChainMap Examples and Recipes

This section shows various approaches to working with chained maps.

Example of simulating Python’s internal lookup chain:

```python
import builtins
def pylookup:
    pylookup = ChainMap(locals(), globals(), vars(builtins))
```
Example of letting user specified command-line arguments take precedence over environment variables which in turn take precedence over default values:

```python
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k:v for k, v in vars(namespace).items() if v}

combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

Example patterns for using the `ChainMap` class to simulate nested contexts:

```python
c = ChainMap()  # Create root context
d = c.new_child()  # Create nested child context
e = c.new_child()  # Child of c, independent from d

e.maps[0]  # Current context dictionary -- like Python’s locals()
e.maps[-1]  # Root context -- like Python’s globals()
e.parents  # Enclosing context chain -- like Python’s nonlocals

d['x']  # Get first key in the chain of contexts
d['x'] = 1  # Set value in current context
def d['x']  # Delete from current context
list(d)  # All nested values
k in d  # Check all nested values
len(d)  # Number of nested values
d.items()  # All nested items
dict(d)  # Flatten into a regular dictionary
```

The `ChainMap` class only makes updates (writes and deletions) to the first mapping in the chain while lookups will search the full chain. However, if deep writes and deletions are desired, it is easy to make a subclass that updates keys found deeper in the chain:

```python
class DeepChainMap(ChainMap):
    'Variant of ChainMap that allows direct updates to inner scopes'

    def __setitem__(self, key, value):
        for mapping in self.maps:
            if key in mapping:
                mapping[key] = value
        return

    def __delitem__(self, key):
        for mapping in self.maps:
            if key in mapping:
                del mapping[key]
        raise KeyError(key)
```

```python
>>> d = DeepChainMap({‘zebra’: ‘black’}, {‘elephant’: ‘blue’}, {‘lion’: ‘yellow’})
>>> d[‘lion’] = ‘orange’  # update an existing key two levels down
>>> d[‘snake’] = ‘red’  # new keys get added to the topmost dict
>>> del d[‘elephant’]  # remove an existing key one level down
```

DeepChainMap({‘zebra’: ‘black’, ‘snake’: ‘red’}, {}, {‘lion’: ‘orange’})
8.3.2 Counter objects

A counter tool is provided to support convenient and rapid tallies. For example:

```python
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})
```

```python
>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631), ('you', 554), ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

```python
class collections.Counter([iterable-or-mapping])
```

A Counter is a dict subclass for counting hashable objects. It is an unordered collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The Counter class is similar to bags or multisets in other languages.

Elements are counted from an iterable or initialized from another mapping (or counter):

```python
>>> c = Counter()  # a new, empty counter
>>> c = Counter('gallahad')  # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2})  # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)  # a new counter from keyword args
```

Counter objects have a dictionary interface except that they return a zero count for missing items instead of raising a KeyError:

```python
>>> c = Counter({'eggs', 'ham'})
>>> c['bacon']  # count of a missing element is zero
0
```

Setting a count to zero does not remove an element from a counter. Use del to remove it entirely:

```python
>>> c['sausage'] = 0  # counter entry with a zero count
>>> del c['sausage']  # del actually removes the entry
```

New in version 3.1. Counter objects support three methods beyond those available for all dictionaries:

`elements()`
Return an iterator over elements repeating each as many times as its count. Elements are returned in arbitrary order. If an element’s count is less than one, elements() will ignore it.

```python
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> list(c.elements())
['a', 'a', 'a', 'b', 'b']
```

`most_common([n])`
Return a list of the n most common elements and their counts from the most common to the least. If n is not specified, most_common() returns all elements in the counter. Elements with equal counts are ordered arbitrarily:

```python
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('r', 2), ('b', 2)]
```
subtract \([\text{iterable-or-mapping}]\)

Elements are subtracted from an iterable or from another mapping (or counter). Like `dict.update()` but subtracts counts instead of replacing them. Both inputs and outputs may be zero or negative.

```python
c = Counter(a=4, b=2, c=0, d=-2)
d = Counter(a=1, b=2, c=3, d=4)
c.subtract(d)
c
```

```
Counter({‘a’: 3, ‘b’: 0, ‘c’: -3, ‘d’: -6})
```

New in version 3.2.

The usual dictionary methods are available for `Counter` objects except for two which work differently for counters.

**fromkeys** \([\text{iterable}]\)

This class method is not implemented for `Counter` objects.

**update** \([\text{iterable-or-mapping}]\)

Elements are counted from an iterable or added-in from another mapping (or counter). Like `dict.update()` but adds counts instead of replacing them. Also, the iterable is expected to be a sequence of elements, not a sequence of (key, value) pairs.

Common patterns for working with `Counter` objects:

```python
sum(c.values())  # total of all counts
c.clear()  # reset all counts
list(c)  # list unique elements
set(c)  # convert to a set
dict(c)  # convert to a regular dictionary
c.items()  # convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs))  # convert from a list of (elem, cnt) pairs
c.most_common()[::-n-1:-1]  # n least common elements
+c  # remove zero and negative counts
```

Several mathematical operations are provided for combining `Counter` objects to produce multisets (counters that have counts greater than zero). Addition and subtraction combine counters by adding or subtracting the counts of corresponding elements. Intersection and union return the minimum and maximum of corresponding counts. Each operation can accept inputs with signed counts, but the output will exclude results with counts of zero or less.

```python
c = Counter(a=3, b=1)
d = Counter(a=1, b=2)
c + d  # add two counters together: c[x] + d[x]
c - d  # subtract (keeping only positive counts)
c & d  # intersection: min(c[x], d[x])
c | d  # union: max(c[x], d[x])
```

Unary addition and substraction are shortcuts for adding an empty counter or subtracting from an empty counter.

```python
c = Counter(a=2, b=-4)
+c
counter({‘a’: 2})
-c
counter({‘b’: 4})
```

New in version 3.3: Added support for unary plus, unary minus, and in-place multiset operations.

8.3. collections — Container datatypes 169
Note: Counters were primarily designed to work with positive integers to represent running counts; however, care was taken to not unnecessarily preclude use cases needing other types or negative values. To help with those use cases, this section documents the minimum range and type restrictions.

- The Counter class itself is a dictionary subclass with no restrictions on its keys and values. The values are intended to be numbers representing counts, but you could store anything in the value field.

- The most_common() method requires only that the values be orderable.

- For in-place operations such as c[key] += 1, the value type need only support addition and subtraction. So fractions, floats, and decimals would work and negative values are supported. The same is also true for update() and subtract() which allow negative and zero values for both inputs and outputs.

- The multiset methods are designed only for use cases with positive values. The inputs may be negative or zero, but only outputs with positive values are created. There are no type restrictions, but the value type needs to support addition, subtraction, and comparison.

- The elements() method requires integer counts. It ignores zero and negative counts.

See Also:

- Counter class adapted for Python 2.5 and an early Bag recipe for Python 2.4.

- Bag class in Smalltalk.

- Wikipedia entry for Multisets.

- C++ multisets tutorial with examples.

- For mathematical operations on multisets and their use cases, see Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19.

- To enumerate all distinct multisets of a given size over a given set of elements, see itertools.combinations_with_replacement().

  map(Counter, combinations_with_replacement('ABC', 2)) -> AA AB AC BB BC CC

### 8.3.3 deque objects

```python
class collections.deque([iterable, maxlen])
```

Returns a new deque object initialized left-to-right (using append()) with data from iterable. If iterable is not specified, the new deque is empty.

Deques are a generalization of stacks and queues (the name is pronounced “deck” and is short for “double-ended queue”). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same O(1) performance in either direction.

Though list objects support similar operations, they are optimized for fast fixed-length operations and incur O(n) memory movement costs for pop(0) and insert(0, v) operations which change both the size and position of the underlying data representation.

If maxlen is not specified or is None, deques may grow to an arbitrary length. Otherwise, the deque is bounded to the specified maximum length. Once a bounded length deque is full, when new items are added, a corresponding number of items are discarded from the opposite end. Bounded length deques provide functionality similar to the tail filter in Unix. They are also useful for tracking transactions and other pools of data where only the most recent activity is of interest.

Deque objects support the following methods:

- **append(x)**
  Add x to the right side of the deque.

- **appendleft(x)**
  Add x to the left side of the deque.
clear()
    Remove all elements from the deque leaving it with length 0.

count(x)
    Count the number of deque elements equal to x. New in version 3.2.

extend(iterable)
    Extend the right side of the deque by appending elements from the iterable argument.

extendleft(iterable)
    Extend the left side of the deque by appending elements from iterable. Note, the series of left appends results in reversing the order of elements in the iterable argument.

pop()
    Remove and return an element from the right side of the deque. If no elements are present, raises an IndexError.

popleft()
    Remove and return an element from the left side of the deque. If no elements are present, raises an IndexError.

remove(value)
    Removed the first occurrence of value. If not found, raises a ValueError.

reverse()
    Reverse the elements of the deque in-place and then return None. New in version 3.2.

rotate(n)
    Rotate the deque n steps to the right. If n is negative, rotate to the left. Rotating one step to the right is equivalent to: d.appendleft(d.pop()).

maxlen
    Maximum size of a deque or None if unbounded. New in version 3.1.

In addition to the above, deques support iteration, pickling, len(d), reversed(d), copy.copy(d), copy.deepcopy(d), membership testing with the in operator, and subscript references such as d[-1]. Indexed access is O(1) at both ends but slows to O(n) in the middle. For fast random access, use lists instead.

Example:

```python
>>> from collections import deque
>>> d = deque('ghi')  # make a new deque with three items
>>> for elem in d:    # iterate over the deque’s elements
...     print(elem.upper())
G
H
I

>>> d.append('j')    # add a new entry to the right side
>>> d.appendleft('f') # add a new entry to the left side
>>> d               # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()          # return and remove the rightmost item
'j'

>>> d.popleft()      # return and remove the leftmost item
'f'

>>> list(d)          # list the contents of the deque
['g', 'h', 'i']

>>> d[0]             # peek at leftmost item
'g'

>>> d[-1]            # peek at rightmost item
'i'
```

8.3. collections — Container datatypes 171
>>> list(reversed(d))  # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d  # search the deque
True
>>> d.extend('jkl')  # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)  # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)  # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))  # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])

>>> d.clear()  # empty the deque
>>> d.pop()  # cannot pop from an empty deque
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
d.pop()
  IndexError: pop from an empty deque

>>> d.extendleft('abc')  # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])

**deque Recipes**

This section shows various approaches to working with deques.

Bounded length deques provide functionality similar to the `tail` filter in Unix:

```python
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return deque(f, n)
```

Another approach to using deques is to maintain a sequence of recently added elements by appending to the right and popping to the left:

```python
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # http://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / n
```

The `rotate()` method provides a way to implement deque slicing and deletion. For example, a pure Python implementation of `del d[n]` relies on the `rotate()` method to position elements to be popped:

```python
def delete_nth(d, n):
    d.rotate(-n)
```
d.popleft()
d.rotate(n)

To implement deque slicing, use a similar approach applying rotate() to bring a target element to the left side of the deque. Remove old entries with popleft(), add new entries with extend(), and then reverse the rotation. With minor variations on that approach, it is easy to implement Forth style stack manipulations such as dup, drop, swap, over, pick, rot, and roll.

8.3.4 defaultdict objects

class collections.defaultdict ([default_factory,[...]])

Returns a new dictionary-like object. defaultdict is a subclass of the built-in dict class. It overrides one method and adds one writable instance variable. The remaining functionality is the same as for the dict class and is not documented here.

The first argument provides the initial value for the default_factory attribute; it defaults to None. All remaining arguments are treated the same as if they were passed to the dict constructor, including keyword arguments.

defaultdict objects support the following method in addition to the standard dict operations:

__missing__(key)

If the default_factory attribute is None, this raises a KeyError exception with the key as argument.

If default_factory is not None, it is called without arguments to provide a default value for the given key; this value is inserted in the dictionary for the key, and returned.

If calling default_factory raises an exception this exception is propagated unchanged.

This method is called by the __getitem__() method of the dict class when the requested key is not found; whatever it returns or raises is then returned or raised by __getitem__().

Note that __missing__() is not called for any operations besides __getitem__(). This means that get() will, like normal dictionaries, return None as a default rather than using default_factory.

defaultdict objects support the following instance variable:

default_factory

This attribute is used by the __missing__() method; it is initialized from the first argument to the constructor, if present, or to None, if absent.

defaultdict Examples

Using list as the default_factory, it is easy to group a sequence of key-value pairs into a dictionary of lists:

>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> list(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]

When each key is encountered for the first time, it is not already in the mapping; so an entry is automatically created using the default_factory function which returns an empty list. The list.append() operation then attaches the value to the new list. When keys are encountered again, the look-up proceeds normally (returning the list for that key) and the list.append() operation adds another value to the list. This technique is simpler and faster than an equivalent technique using dict.setdefault():

8.3. collections — Container datatypes 173
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
... >>> list(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]

Setting the default_factory to int makes the defaultdict useful for counting (like a bag or multiset in other languages):

>>> s = ‘mississippi’
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
... >>> list(d.items())
[('i', 4), ('p', 2), ('s', 4), ('m', 1)]

When a letter is first encountered, it is missing from the mapping, so the default_factory function calls int() to supply a default count of zero. The increment operation then builds up the count for each letter.

The function int() which always returns zero is just a special case of constant functions. A faster and more flexible way to create constant functions is to use a lambda function which can supply any constant value (not just zero):

>>> def constant_factory(value):
...     return lambda: value
... >>> d = defaultdict(constant_factory('<missing>'))
... >>> d.update(name='John', action='ran')
... >>> ‘%(name)s %(action)s to %(object)s’ % d
'John ran to <missing>'

Setting the default_factory to set makes the defaultdict useful for building a dictionary of sets:

>>> s = [(’red’, 1), (’blue’, 2), (’red’, 3), (’blue’, 4), (’red’, 1), (’blue’, 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
... >>> list(d.items())
[('blue', {2, 4}), ('red', {1, 3})]

8.3.5 namedtuple() Factory Function for Tuples with Named Fields

Named tuples assign meaning to each position in a tuple and allow for more readable, self-documenting code. They can be used wherever regular tuples are used, and they add the ability to access fields by name instead of position index.

collections.namedtuple (typename, field_names, verbose=False, rename=False)

Returns a new tuple subclass named typename. The new subclass is used to create tuple-like objects that have fields accessible by attribute lookup as well as being indexable and iterable. Instances of the subclass also have a helpful docstring (with typename and field_names) and a helpful __repr__() method which lists the tuple contents in a name=value format.

The field_names are a single string with each fieldname separated by whitespace and/or commas, for example ‘x y’ or ‘x, y’. Alternatively, field_names can be a sequence of strings such as [’x’, ’y’].

Any valid Python identifier may be used for a fieldname except for names starting with an underscore. Valid identifiers consist of letters, digits, and underscores but do not start with a digit or underscore and cannot be a keyword such as class, for, return, global, pass, or raise.
If `rename` is true, invalid fieldnames are automatically replaced with positional names. For example, `['abc', 'def', 'ghi', 'abc']` is converted to `['abc', '_1', 'ghi', '_3']`, eliminating the keyword `def` and the duplicate fieldname `abc`.

If `verbose` is true, the class definition is printed after it is built. This option is outdated; instead, it is simpler to print the `_source` attribute.

Named tuple instances do not have per-instance dictionaries, so they are lightweight and require no more memory than regular tuples. Changed in version 3.1: Added support for `rename`.

```python
>>> # Basic example
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22)  # instantiate with positional or keyword arguments
>>> p[0] + p[1]         # indexable like the plain tuple (11, 22)
33
>>> x, y = p            # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y           # fields also accessible by name
33
>>> p                  # readable __repr__ with a name=value style
Point(x=11, y=22)
```

Named tuples are especially useful for assigning field names to result tuples returned by the `csv` or `sqlite3` modules:

```python
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, paygrade')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
    print(emp.name, emp.title)

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print(emp.name, emp.title)
```

In addition to the methods inherited from tuples, named tuples support three additional methods and two attributes. To prevent conflicts with field names, the method and attribute names start with an underscore.

```python
classmethod somenamedtuple._make(iterable)
    Class method that makes a new instance from an existing sequence or iterable.

>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

```python
classmethod somenamedtuple._asdict()
    Return a new OrderedDict which maps field names to their corresponding values. Note, this method is no longer needed now that the same effect can be achieved by using the built-in `vars()` function:

>>> vars(p)
OrderedDict([('x', 11), ('y', 22)])
```

Changed in version 3.1: Returns an `OrderedDict` instead of a regular `dict`.

```python
classmethod somenamedtuple._replace(kwargs)
    Return a new instance of the named tuple replacing specified fields with new values:
```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)

```python
for partnum, record in inventory.items():
... inventory[partnum] = record._replace(price=newprices[partnum], timestamp=time.now())
```

somenamedtuple._source
A string with the pure Python source code used to create the named tuple class. The source makes the named tuple self-documenting. It can be printed, executed using `exec()`, or saved to a file and imported. New in version 3.3.

somenamedtuple._fields
Tuple of strings listing the field names. Useful for introspection and for creating new named tuple types from existing named tuples.

```python
>>> p._fields
('x', 'y')
```

```python
Color = namedtuple('Color', 'red green blue')
Pixel = namedtuple('Pixel', Point._fields + Color._fields)
Pixel(11, 22, 128, 255, 0)
```

To retrieve a field whose name is stored in a string, use the `getattr()` function:

```python
>>> getattr(p, 'x')
11
```

To convert a dictionary to a named tuple, use the double-star-operator (as described in `tut-unpacking-arguments`):

```python
d = {'x': 11, 'y': 22}
Point(**d)
```

Since a named tuple is a regular Python class, it is easy to add or change functionality with a subclass. Here is how to add a calculated field and a fixed-width print format:

```python
class Point(namedtuple('Point', 'x y')):
    __slots__ = ()
    @property
def hypot(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5
def __str__(self):
    return 'Point: x=%6.3f y=%6.3f hypot=%6.3f' % (self.x, self.y, self.hypot)
```

```python
for p in Point(3, 4), Point(14, 5/7):
print(p)
```

```python
Point: x= 3.000 y= 4.000 hypot= 5.000
Point: x=14.000 y= 0.714 hypot=14.018
```

The subclass shown above sets `__slots__` to an empty tuple. This helps keep memory requirements low by preventing the creation of instance dictionaries.

Subclassing is not useful for adding new, stored fields. Instead, simply create a new named tuple type from the _fields attribute:

```python
Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

Default values can be implemented by using `_replace()` to customize a prototype instance:

```python
Account = namedtuple('Account', 'owner balance transaction_count')
default_account = Account('<owner name>', 0.0, 0)
```

176 Chapter 8. Data Types
Entrained constants can be implemented with named tuples, but it is simpler and more efficient to use a simple class declaration:

```python
>>> Status = namedtuple('Status', 'open pending closed')._make(range(3))
>>> Status.open, Status.pending, Status.closed
(0, 1, 2)
```

```python
>>> class Status:
    open, pending, closed = range(3)
```

See Also:
- Named tuple recipe adapted for Python 2.4.
- Recipe for named tuple abstract base class with a metaclass mix-in by Jan Kaliszewski. Besides providing an abstract base class for named tuples, it also supports an alternate metaclass-based constructor that is convenient for use cases where named tuples are being subclassed.

### 8.3.6 OrderedDict objects

Ordered dictionaries are just like regular dictionaries but they remember the order that items were inserted. When iterating over an ordered dictionary, the items are returned in the order their keys were first added.

```python
class collections.OrderedDict([items])
```

Return an instance of a dict subclass, supporting the usual `dict` methods. An `OrderedDict` is a dict that remembers the order that keys were first inserted. If a new entry overwrites an existing entry, the original insertion position is left unchanged. Deleting an entry and reinserting it will move it to the end. New in version 3.1.

```python
popitem(last=True)
```

The `popitem()` method for ordered dictionaries returns and removes a (key, value) pair. The pairs are returned in LIFO order if `last` is true or FIFO order if false.

```python
move_to_end(key, last=True)
```

Move an existing `key` to either end of an ordered dictionary. The item is moved to the right end if `last` is true (the default) or to the beginning if `last` is false. Raises `KeyError` if the `key` does not exist:

```python
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
'acdeb'
>>> d.move_to_end('b', last=False)
'bacde'
```

New in version 3.2.

In addition to the usual mapping methods, ordered dictionaries also support reverse iteration using `reversed()`.

Equality tests between `OrderedDict` objects are order-sensitive and are implemented as `list(od1.items())==list(od2.items())`. Equality tests between `OrderedDict` objects and other `Mapping` objects are order-insensitive like regular dictionaries. This allows `OrderedDict` objects to be substituted anywhere a regular dictionary is used.

The `OrderedDict` constructor and `update()` method both accept keyword arguments, but their order is lost because Python’s function call semantics pass-in keyword arguments using a regular unordered dictionary.

See Also:
- Equivalent `OrderedDict` recipe that runs on Python 2.4 or later.
OrderedDict Examples and Recipes

Since an ordered dictionary remembers its insertion order, it can be used in conjunction with sorting to make a sorted dictionary:

```python
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

>>> # dictionary sorted by value
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])

>>> # dictionary sorted by length of the key string
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2), ('banana', 3)])
```

The new sorted dictionaries maintain their sort order when entries are deleted. But when new keys are added, the keys are appended to the end and the sort is not maintained.

It is also straight-forward to create an ordered dictionary variant that remembers the order the keys were last inserted. If a new entry overwrites an existing entry, the original insertion position is changed and moved to the end:

```python
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        if key in self:
            del self[key]
        OrderedDict.__setitem__(self, key, value)
```

An ordered dictionary can be combined with the `Counter` class so that the counter remembers the order elements are first encountered:

```python
class OrderedCounter(Counter, OrderedDict):
    'Counter that remembers the order elements are first encountered'

    def __repr__(self):
        return '%s(%r)' % (self.__class__.__name__, OrderedDict(self))

    def __reduce__(self):
        return self.__class__, (OrderedDict(self),)
```

8.3.7 UserDict objects

The class, `UserDict` acts as a wrapper around dictionary objects. The need for this class has been partially supplanted by the ability to subclass directly from `dict`; however, this class can be easier to work with because the underlying dictionary is accessible as an attribute.

```python
class collections.UserDict([initialdata])
    Class that simulates a dictionary. The instance’s contents are kept in a regular dictionary, which is accessible via the data attribute of UserDict instances. If initialdata is provided, data is initialized with its contents; note that a reference to initialdata will not be kept, allowing it to be used for other purposes.

    In addition to supporting the methods and operations of mappings, UserDict instances provide the following attribute:
```
data
A real dictionary used to store the contents of the UserDict class.

8.3.8 UserList objects

This class acts as a wrapper around list objects. It is a useful base class for your own list-like classes which can inherit from them and override existing methods or add new ones. In this way, one can add new behaviors to lists. The need for this class has been partially supplanted by the ability to subclass directly from list; however, this class can be easier to work with because the underlying list is accessible as an attribute.

class collections.UserList([list])
Class that simulates a list. The instance’s contents are kept in a regular list, which is accessible via the data attribute of UserList instances. The instance’s contents are initially set to a copy of list, defaulting to the empty list[]. list can be any iterable, for example a real Python list or a UserList object.

In addition to supporting the methods and operations of mutable sequences, UserList instances provide the following attribute:

data
A real list object used to store the contents of the UserList class.

Subclassing requirements: Subclasses of UserList are expect to offer a constructor which can be called with either no arguments or one argument. List operations which return a new sequence attempt to create an instance of the actual implementation class. To do so, it assumes that the constructor can be called with a single parameter, which is a sequence object used as a data source.

If a derived class does not wish to comply with this requirement, all of the special methods supported by this class will need to be overridden; please consult the sources for information about the methods which need to be provided in that case.

8.3.9 UserString objects

The class, UserString acts as a wrapper around string objects. The need for this class has been partially supplanted by the ability to subclass directly from str; however, this class can be easier to work with because the underlying string is accessible as an attribute.

class collections.UserString([sequence])
Class that simulates a string or a Unicode string object. The instance’s content is kept in a regular string object, which is accessible via the data attribute of UserString instances. The instance’s contents are initially set to a copy of sequence. The sequence can be an instance of bytes, str, UserString (or a subclass) or an arbitrary sequence which can be converted into a string using the built-in str() function.

8.4 collections.abc — Abstract Base Classes for Containers

New in version 3.3: Formerly, this module was part of the collections module. Source code: Lib/collections/abc.py

This module provides abstract base classes that can be used to test whether a class provides a particular interface; for example, whether it is hashable or whether it is a mapping.

8.4.1 Collections Abstract Base Classes

The collections module offers the following ABCs:
### ABC Inheritance

<table>
<thead>
<tr>
<th>ABC</th>
<th>Inherits from</th>
<th>Abstract Methods</th>
<th>Mixin Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container</td>
<td></td>
<td><strong>contains</strong></td>
<td></td>
</tr>
<tr>
<td>Hashable</td>
<td></td>
<td><strong>hash</strong></td>
<td></td>
</tr>
<tr>
<td>Iterable</td>
<td></td>
<td><strong>iter</strong></td>
<td></td>
</tr>
<tr>
<td>Iterator</td>
<td>Iterable</td>
<td><strong>next</strong></td>
<td><strong>iter</strong></td>
</tr>
<tr>
<td>Sized</td>
<td></td>
<td><strong>len</strong></td>
<td></td>
</tr>
<tr>
<td>Callable</td>
<td></td>
<td><strong>call</strong></td>
<td></td>
</tr>
<tr>
<td>Sequence</td>
<td>Sized, Iterable</td>
<td><strong>getitem</strong>, <strong>contains</strong>, <strong>iter</strong>, <strong>len</strong>, <strong>reversed</strong>, index, and count</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>iter</strong></td>
<td></td>
</tr>
<tr>
<td>MutatableSequence</td>
<td>Sequence</td>
<td><strong>getitem</strong>, <strong>setitem</strong>, <strong>delitem</strong>, <strong>len</strong>, insert</td>
<td>Inherited Sequence methods and append, reverse, extend, pop, remove, and <strong>iadd</strong></td>
</tr>
<tr>
<td>Set</td>
<td>Sized, Iterable</td>
<td><strong>contains</strong>, <strong>len</strong>, <strong>le</strong>, <strong>lt</strong>, <strong>eq</strong>, <strong>ne</strong>, <strong>gt</strong>, <strong>ge</strong>, <strong>and</strong>, <strong>or</strong>, <strong>sub</strong>, <strong>xor</strong>, and isdisjoint</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Container</td>
<td><strong>iter</strong>, <strong>len</strong></td>
<td></td>
</tr>
<tr>
<td>MutatableSet</td>
<td>Set</td>
<td><strong>contains</strong>, <strong>iter</strong>, <strong>len</strong></td>
<td>Inherited Set methods and clear, pop, remove, <strong>ior</strong>, <strong>iand</strong>, <strong>ixor</strong>, and <strong>isub</strong></td>
</tr>
<tr>
<td>Mapping</td>
<td>Sized, Iterable</td>
<td><strong>getitem</strong>, <strong>contains</strong>, keys, items, values, get, <strong>eq</strong>, and <strong>ne</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Container</td>
<td><strong>iter</strong>, <strong>len</strong></td>
<td></td>
</tr>
<tr>
<td>MutatableMapping</td>
<td>Mapping</td>
<td><strong>getitem</strong>, <strong>setitem</strong>, <strong>delitem</strong>, <strong>iter</strong>, <strong>len</strong></td>
<td>Inherited Mapping methods and pop, popitem, clear, update, and setdefault</td>
</tr>
<tr>
<td>MappingView</td>
<td>Sized</td>
<td><strong>len</strong></td>
<td></td>
</tr>
<tr>
<td>ItemsView</td>
<td>MappingView, Set</td>
<td><strong>contains</strong>, <strong>iter</strong></td>
<td></td>
</tr>
<tr>
<td>KeysView</td>
<td>MappingView, Set</td>
<td><strong>contains</strong>, <strong>iter</strong></td>
<td></td>
</tr>
<tr>
<td>ValuesView</td>
<td>MappingView</td>
<td><strong>contains</strong>, <strong>iter</strong></td>
<td></td>
</tr>
</tbody>
</table>

**class collections.abc.Container**

ABCs for classes that provide respectively the methods __contains__(), __hash__(), __len__(), and __call__().

**class collections.abc.Hashable**

ABC for classes that provide the __hash__() method. See also the definition of iterable.

**class collections.abc.Sized**

ABC for classes that provide the __len__() method. See also the definition of iterable.

**class collections.abc.Callable**

ABC for classes that provide the __iter__() and __next__() methods. See also the definition of iterator.

**class collections.abc.Sequence**

ABCs for read-only and mutable sequences.

**class collections.abc.Set**

ABCs for read-only and mutable sets.

**class collections.abc.MutableSequence**

ABCs for read-only and mutable sequences.

**class collections.abc.MutableSet**

ABCs for read-only and mutable sets.

**class collections.abc.Mapping**

ABCs for read-only and mutable mappings.
class collections.abc.MappingView
class collections.abc.ItemsView
class collections.abc.KeysView
class collections.abc.ValuesView

ABCs for mapping, items, keys, and values views.

These ABCs allow us to ask classes or instances if they provide particular functionality, for example:

```python
size = None
if isinstance(myvar, collections.abc.Sized):
    size = len(myvar)
```

Several of the ABCs are also useful as mixins that make it easier to develop classes supporting container APIs. For example, to write a class supporting the full `Set` API, it only necessary to supply the three underlying abstract methods: `__contains__()`, `__iter__()`, and `__len__()`. The ABC supplies the remaining methods such as `__and__()` and `isdisjoint()`:

```python
class ListBasedSet(collections.abc.Set):
    ''' Alternate set implementation favoring space over speed
    and not requiring the set elements to be hashable. '''
    def __init__(self, iterable):
        self.elements = lst = []
        for value in iterable:
            if value not in lst:
                lst.append(value)
    def __iter__(self):
        return iter(self.elements)
    def __contains__(self, value):
        return value in self.elements
    def __len__(self):
        return len(self.elements)
```

```python
s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2  # The __and__() method is supported automatically
```

Notes on using `Set` and `MutableSet` as a mixin:

1. Since some set operations create new sets, the default mixin methods need a way to create new instances from an iterable. The class constructor is assumed to have a signature in the form `ClassName(iterable)`. That assumption is factored-out to an internal classmethod called `_from_iterable()` which calls `cls(iterable)` to produce a new set. If the `Set` mixin is being used in a class with a different constructor signature, you will need to override `_from_iterable()` with a classmethod that can construct new instances from an iterable argument.

2. To override the comparisons (presumably for speed, as the semantics are fixed), redefine `__le__()` and then the other operations will automatically follow suit.

3. The `Set` mixin provides a `_hash()` method to compute a hash value for the set; however, `__hash__()` is not defined because not all sets are hashable or immutable. To add set hashability using mixins, inherit from both `Set()` and `Hashable()`, then define `__hash__ = Set._hash`.

See Also:

- `OrderedSet` recipe for an example built on `MutableSet`.
- For more about ABCs, see the `abc` module and [PEP 3119](https://www.python.org/dev/peps/pep-0319/).

### 8.5 heapq — Heap queue algorithm

Source code: Lib/heapq.py
This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm. Heaps are binary trees for which every parent node has a value less than or equal to any of its children. This implementation uses arrays for which heap[k] <= heap[2*k+1] and heap[k] <= heap[2*k+2] for all k, counting elements from zero. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that its smallest element is always the root, heap[0].

The API below differs from textbook heap algorithms in two aspects: (a) We use zero-based indexing. This makes the relationship between the index for a node and the indexes for its children slightly less obvious, but is more suitable since Python uses zero-based indexing. (b) Our pop method returns the smallest item, not the largest (called a “min heap” in textbooks; a “max heap” is more common in texts because of its suitability for in-place sorting).

These two make it possible to view the heap as a regular Python list without surprises: heap[0] is the smallest item, and heap.sort() maintains the heap invariant!

To create a heap, use a list initialized to [], or you can transform a populated list into a heap via function heapify().

The following functions are provided:

- heapq.heappush(heap, item)
  Push the value item onto the heap, maintaining the heap invariant.

- heapq.heappop(heap)
  Pop and return the smallest item from the heap, maintaining the heap invariant. If the heap is empty, IndexError is raised.

- heapq.heappushpop(heap, item)
  Push item on the heap, then pop and return the smallest item from the heap. The combined action runs more efficiently than heappush() followed by a separate call to heappop().

- heapq.heapify(x)
  Transform list x into a heap, in-place, in linear time.

- heapq.heappop(heap, item)
  Pop and return the smallest item from the heap, and also push the new item. The heap size doesn’t change. If the heap is empty, IndexError is raised.

  This one step operation is more efficient than a heappop() followed by heappush() and can be more appropriate when using a fixed-size heap. The pop/push combination always returns an element from the heap and replaces it with item.

  The value returned may be larger than the item added. If that isn’t desired, consider using heappushpop() instead. Its push/pop combination returns the smaller of the two values, leaving the larger value on the heap.

The module also offers three general purpose functions based on heaps.

- heapq.merge(*iterables)
  Merge multiple sorted inputs into a single sorted output (for example, merge timestamped entries from multiple log files). Returns an iterator over the sorted values. Similar to sorted(itertools.chain(*iterables)) but returns an iterable, does not pull the data into memory all at once, and assumes that each of the input streams is already sorted (smallest to largest).

- heapq.nlargest(n, iterable, key=None)
  Return a list with the n largest elements from the dataset defined by iterable. key, if provided, specifies a function of one argument that is used to extract a comparison key from each element in the iterable: key=str.lower Equivalent to: sorted(iterable, key=key, reverse=True)[:]n

- heapq.nsmallest(n, iterable, key=None)
  Return a list with the n smallest elements from the dataset defined by iterable. key, if provided, specifies a function of one argument that is used to extract a comparison key from each element in the iterable: key=str.lower Equivalent to: sorted(iterable, key=key)[:n]
The latter two functions perform best for smaller values of \( n \). For larger values, it is more efficient to use the \( \text{sorted}\) function. Also, when \( n = 1 \), it is more efficient to use the built-in \( \text{min}() \) and \( \text{max}() \) functions.

### 8.5.1 Basic Examples

A `heapsort` can be implemented by pushing all values onto a heap and then popping off the smallest values one at a time:

```python
>>> def heapsort(iterable):
...     "Equivalent to \text{sorted}(iterable)"
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Heap elements can be tuples. This is useful for assigning comparison values (such as task priorities) alongside the main record being tracked:

```python
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, ‘write spec’)
```

### 8.5.2 Priority Queue Implementation Notes

A `priority queue` is common use for a heap, and it presents several implementation challenges:

- Sort stability: how do you get two tasks with equal priorities to be returned in the order they were originally added?
- Tuple comparison breaks for (priority, task) pairs if the priorities are equal and the tasks do not have a default comparison order.
- If the priority of a task changes, how do you move it to a new position in the heap?
- Or if a pending task needs to be deleted, how do you find it and remove it from the queue?

A solution to the first two challenges is to store entries as 3-element list including the priority, an entry count, and the task. The entry count serves as a tie-breaker so that two tasks with the same priority are returned in the order they were added. And since no two entry counts are the same, the tuple comparison will never attempt to directly compare two tasks.

The remaining challenges revolve around finding a pending task and making changes to its priority or removing it entirely. Finding a task can be done with a dictionary pointing to an entry in the queue.

Removing the entry or changing its priority is more difficult because it would break the heap structure invariants. So, a possible solution is to mark the entry as removed and add a new entry with the revised priority:

```python
pq = []
entry_finder = {} # mapping of tasks to entries
REMOVED = '<removed-task>' # placeholder for a removed task
counter = itertools.count() # unique sequence count

def add_task(task, priority=0):
    'Add a new task or update the priority of an existing task'
    if task in entry_finder:
```
remove_task(task)
count = next(counter)
entry = [priority, count, task]
entry_finder[task] = entry
heappush(pq, entry)

def remove_task(task):
    ’Mark an existing task as REMOVED. Raise KeyError if not found.’
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    ’Remove and return the lowest priority task. Raise KeyError if empty.’
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError(’pop from an empty priority queue’)

8.5.3 Theory

Heaps are arrays for which a[k] <= a[2*k+1] and a[k] <= a[2*k+2] for all k, counting elements from 0. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that a[0] is always its smallest element.

The strange invariant above is meant to be an efficient memory representation for a tournament. The numbers below are k, not a[k]:

```
    0
   / \   
 1   2
 / \   / \ 
3 4 5 6
/ \ / \ / \ / 
7 8 9 10 11 12 13 14
```

In the tree above, each cell k is topping 2*k+1 and 2*k+2. In an usual binary tournament we see in sports, each cell is the winner over the two cells it tops, and we can trace the winner down the tree to see all opponents s/he had. However, in many computer applications of such tournaments, we do not need to trace the history of a winner. To be more memory efficient, when a winner is promoted, we try to replace it by something else at a lower level, and the rule becomes that a cell and the two cells it tops contain three different items, but the top cell “wins” over the two topped cells.

If this heap invariant is protected at all time, index 0 is clearly the overall winner. The simplest algorithmic way to remove it and find the “next” winner is to move some loser (let’s say cell 30 in the diagram above) into the 0 position, and then percolate this new 0 down the tree, exchanging values, until the invariant is re-established. This is clearly logarithmic on the total number of items in the tree. By iterating over all items, you get an O(n log n) sort.

A nice feature of this sort is that you can efficiently insert new items while the sort is going on, provided that the inserted items are not “better” than the last 0’th element you extracted. This is especially useful in simulation contexts, where the tree holds all incoming events, and the “win” condition means the smallest scheduled time. When an event schedules other events for execution, they are scheduled into the future, so they can easily go into the heap. So, a heap is a good structure for implementing schedulers (this is what I used for my MIDI sequencer :-).

Various structures for implementing schedulers have been extensively studied, and heaps are good for this, as they
are reasonably speedy, the speed is almost constant, and the worst case is not much different than the average case. However, there are other representations which are more efficient overall, yet the worst cases might be terrible.

Heaps are also very useful in big disk sorts. You most probably all know that a big sort implies producing “runs” (which are pre-sorted sequences, which size is usually related to the amount of CPU memory), followed by a merging passes for these runs, which merging is often very cleverly organised. It is very important that the initial sort produces the longest runs possible. Tournaments are a good way to that. If, using all the memory available to hold a tournament, you replace and percolate items that happen to fit the current run, you’ll produce runs which are twice the size of the memory for random input, and much better for input fuzzily ordered.

Moreover, if you output the 0’t item on disk and get an input which may not fit in the current tournament (because the value “wins” over the last output value), it cannot fit in the heap, so the size of the heap decreases. The freed memory could be cleverly reused immediately for progressively building a second heap, which grows at exactly the same rate the first heap is melting. When the first heap completely vanishes, you switch heaps and start a new run. Clever and quite effective!

In a word, heaps are useful memory structures to know. I use them in a few applications, and I think it is good to keep a ‘heap’ module around. :-)

### 8.6 bisect — Array bisection algorithm

**Source code:** Lib/bisect.py

This module provides support for maintaining a list in sorted order without having to sort the list after each insertion. For long lists of items with expensive comparison operations, this can be an improvement over the more common approach. The module is called *bisect* because it uses a basic bisection algorithm to do its work. The source code may be most useful as a working example of the algorithm (the boundary conditions are already right!).

The following functions are provided:

- **bisect.bisect_left**(a, x, \(lo=0\), \(hi=len(a)\))
  
  Locate the insertion point for \(x\) in \(a\) to maintain sorted order. The parameters \(lo\) and \(hi\) may be used to specify a subset of the list which should be considered; by default the entire list is used. If \(x\) is already present in \(a\), the insertion point will be before (to the left of) any existing entries. The return value is suitable for use as the first parameter to \(list.insert()\) assuming that \(a\) is already sorted.

  The returned insertion point \(i\) partitions the array \(a\) into two halves so that \(all(val < x \text{ for } val \in a[lo:i])\) for the left side and \(all(val >= x \text{ for } val \in a[i:hi])\) for the right side.

- **bisect.bisect_right**(a, x, \(lo=0\), \(hi=len(a)\))

- **bisect.bisect**(a, x, \(lo=0\), \(hi=len(a)\))

  Similar to **bisect.bisect_left()**, but returns an insertion point which comes after (to the right of) any existing entries of \(x\) in \(a\).

  The returned insertion point \(i\) partitions the array \(a\) into two halves so that \(all(val <= x \text{ for } val \in a[lo:i])\) for the left side and \(all(val > x \text{ for } val \in a[i:hi])\) for the right side.

- **bisect.insort_left**(a, x, \(lo=0\), \(hi=len(a)\))

  Insert \(x\) in \(a\) in sorted order. This is equivalent to \(a.insert(bisect.bisect_left(a, x, lo, hi), x)\) assuming that \(a\) is already sorted. Keep in mind that the \(O(\log n)\) search is dominated by the slow \(O(n)\) insertion step.

- **bisect.insort_right**(a, x, \(lo=0\), \(hi=len(a)\))

- **bisect.insort**(a, x, \(lo=0\), \(hi=len(a)\))

  Similar to **bisect.insort_left()**, but inserting \(x\) in \(a\) after any existing entries of \(x\).

---

2 The disk balancing algorithms which are current, nowadays, are more annoying than clever, and this is a consequence of the seeking capabilities of the disks. On devices which cannot seek, like big tape drives, the story was quite different, and one had to be very clever to ensure (far in advance) that each tape movement will be the most effective possible (that is, will best participate at “progressing” the merge). Some tapes were even able to read backwards, and this was also used to avoid the rewinding time. Believe me, real good tape sorts were quite spectacular to watch! From all times, sorting has always been a Great Art! :-)

---

8.6. **bisect** — Array bisection algorithm

185
See Also:

SortedCollection recipe that uses bisect to build a full-featured collection class with straight-forward search methods and support for a key-function. The keys are precomputed to save unnecessary calls to the key function during searches.

8.6.1 Searching Sorted Lists

The above bisect() functions are useful for finding insertion points but can be tricky or awkward to use for common searching tasks. The following five functions show how to transform them into the standard lookups for sorted lists:

```python
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

def find_ge(a, x):
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
```

8.6.2 Other Examples

The bisect() function can be useful for numeric table lookups. This example uses bisect() to look up a letter grade for an exam score (say) based on a set of ordered numeric breakpoints: 90 and up is an ‘A’, 80 to 89 is a ‘B’, and so on:

```python
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
    ...     i = bisect(breakpoints, score)
    ...     return grades[i]
... ```
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']

Unlike the `sorted()` function, it does not make sense for the `bisect()` functions to have `key` or `reversed` arguments because that would lead to an inefficient design (successive calls to `bisect` functions would not “remember” all of the previous key lookups).

Instead, it is better to search a list of precomputed keys to find the index of the record in question:

```python
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])
>>> keys = [r[1] for r in data]  # precomputed list of keys
>>> data[bisect_left(keys, 0)]  # 'black', 0
>>> data[bisect_left(keys, 1)]  # 'blue', 1
>>> data[bisect_left(keys, 5)]  # 'red', 5
>>> data[bisect_left(keys, 8)]  # 'yellow', 8
```

## 8.7 array — Efficient arrays of numeric values

This module defines an object type which can compactly represent an array of basic values: characters, integers, floating point numbers. Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a `type code`, which is a single character. The following type codes are defined:

<table>
<thead>
<tr>
<th>Type code</th>
<th>C Type</th>
<th>Python Type</th>
<th>Minimum size in bytes</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>'b'</td>
<td>signed char</td>
<td>int</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>'B'</td>
<td>unsigned char</td>
<td>int</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>'u'</td>
<td>Py_UNICODE</td>
<td>Unicode character</td>
<td>2</td>
<td>(1)</td>
</tr>
<tr>
<td>'h'</td>
<td>signed short</td>
<td>int</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>'H'</td>
<td>unsigned short</td>
<td>int</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>'i'</td>
<td>signed int</td>
<td>int</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>'I'</td>
<td>unsigned int</td>
<td>int</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>'l'</td>
<td>signed long</td>
<td>int</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>'L'</td>
<td>unsigned long</td>
<td>int</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>'q'</td>
<td>signed long long</td>
<td>int</td>
<td>8</td>
<td>(2)</td>
</tr>
<tr>
<td>'Q'</td>
<td>unsigned long long</td>
<td>int</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>'f'</td>
<td>float</td>
<td>float</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>'d'</td>
<td>double</td>
<td>float</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. The ‘u’ type code corresponds to Python’s obsolete unicode character (Py_UNICODE which is wchar_t). Depending on the platform, it can be 16 bits or 32 bits.

   ‘u’ will be removed together with the rest of the Py_UNICODE API. Deprecated since version 3.3, will be removed in version 4.0.

2. The ‘q’ and ‘Q’ type codes are available only if the platform C compiler used to build Python supports C
   long long, or, on Windows, __int64. New in version 3.3.

The actual representation of values is determined by the machine architecture (strictly speaking, by the C implementation). The actual size can be accessed through the `itemsize` attribute.

The module defines the following type:

```python
class array.array (typecode[, initializer])
```

A new array whose items are restricted by `typecode`, and initialized from the optional `initializer` value, which must be a list, a `bytes-like object`, or iterable over elements of the appropriate type.
If given a list or string, the initializer is passed to the new array’s `fromlist()`, `frombytes()`, or `fromunicode()` method (see below) to add initial items to the array. Otherwise, the iterable initializer is passed to the `extend()` method.

`array.typecodes`
A string with all available type codes.

Array objects support the ordinary sequence operations of indexing, slicing, concatenation, and multiplication. When using slice assignment, the assigned value must be an array object with the same type code; in all other cases, `TypeError` is raised. Array objects also implement the buffer interface, and may be used wherever `bytes-like objects` are supported.

The following data items and methods are also supported:

`array.typecode`
The typecode character used to create the array.

`array.itemsize`
The length in bytes of one array item in the internal representation.

`array.append(x)`
Append a new item with value `x` to the end of the array.

`array.buffer_info()`
Return a tuple `(address, length)` giving the current memory address and the length in elements of the buffer used to hold array’s contents. The size of the memory buffer in bytes can be computed as `array.buffer_info()[1] * array.itemsize`. This is occasionally useful when working with low-level (and inherently unsafe) I/O interfaces that require memory addresses, such as certain `ioctl()` operations. The returned numbers are valid as long as the array exists and no length-changing operations are applied to it.

**Note:** When using array objects from code written in C or C++ (the only way to effectively make use of this information), it makes more sense to use the buffer interface supported by array objects. This method is maintained for backward compatibility and should be avoided in new code. The buffer interface is documented in `bufferobjects`.

`array.byteswap()`
“Byteswap” all items of the array. This is only supported for values which are 1, 2, 4, or 8 bytes in size; for other types of values, `RuntimeError` is raised. It is useful when reading data from a file written on a machine with a different byte order.

`array.count(x)`
Return the number of occurrences of `x` in the array.

`array.extend(iterable)`
Append items from `iterable` to the end of the array. If `iterable` is another array, it must have exactly the same type code; if not, `TypeError` will be raised. If `iterable` is not an array, it must be iterable and its elements must be the right type to be appended to the array.

`array.frombytes(s)`
Appends items from the string, interpreting the string as an array of machine values (as if it had been read from a file using the `fromfile()` method). New in version 3.2: `fromstring()` is renamed to `frombytes()` for clarity.

`array.fromfile(f, n)`
Read `n` items (as machine values) from the file object `f` and append them to the end of the array. If less than `n` items are available, `EOFError` is raised, but the items that were available are still inserted into the array. `f` must be a real built-in file object; something else with a `read()` method won’t do.

`array.fromlist(list)`
Append items from the list. This is equivalent to `for x in list: a.append(x)` except that if there is a type error, the array is unchanged.
array.\texttt{fromstring}()  
Depreciated alias for \texttt{frombytes()}.

array.\texttt{fromunicode}(s)  
Extends this array with data from the given unicode string. The array must be a type ‘u’ array; otherwise a ValueError is raised. Use array.\texttt{frombytes(unicodestring.encode(enc))} to append Unicode data to an array of some other type.

array.\texttt{index}(x)  
Return the smallest $i$ such that $i$ is the index of the first occurrence of $x$ in the array.

array.\texttt{insert}(i, x)  
Insert a new item with value $x$ in the array before position $i$. Negative values are treated as being relative to the end of the array.

array.\texttt{pop}([i])  
Removes the item with the index $i$ from the array and returns it. The optional argument defaults to -1, so that by default the last item is removed and returned.

array.\texttt{remove}(x)  
Remove the first occurrence of $x$ from the array.

array.\texttt{reverse}()  
Reverse the order of the items in the array.

array.\texttt{tobytes}()  
Convert the array to an array of machine values and return the bytes representation (the same sequence of bytes that would be written to a file by the tofile() method.) New in version 3.2: tostring() is renamed to tobytes() for clarity.

array.\texttt{tofile}(f)  
Write all items (as machine values) to the file object $f$.

array.\texttt{tolist}()  
Convert the array to an ordinary list with the same items.

array.\texttt{tostring}()  
Deprecated alias for tobytes().

array.\texttt{tounicode}()  
Convert the array to a unicode string. The array must be a type ‘u’ array; otherwise a ValueError is raised. Use array.tobytes().decode(enc) to obtain a unicode string from an array of some other type.

When an array object is printed or converted to a string, it is represented as array(typecode, initializer). The initializer is omitted if the array is empty, otherwise it is a string if the typecode is ‘u’, otherwise it is a list of numbers. The string is guaranteed to be able to be converted back to an array with the same type and value using eval(), so long as the array() function has been imported using from array import array. Examples:

array(‘l’)  
array(‘u’, ‘hello \u2641’)  
array(‘l’, [1, 2, 3, 4, 5])  
array(‘d’, [1.0, 2.0, 3.14])

See Also:

Module \texttt{struct}  Packing and unpacking of heterogeneous binary data.

Module \texttt{xdrlib}  Packing and unpacking of External Data Representation (XDR) data as used in some remote procedure call systems.

The Numerical Python Documentation  The Numeric Python extension (NumPy) defines another array type; see http://www.numpy.org/ for further information about Numerical Python.

8.7. array — Efficient arrays of numeric values  189
The `weakref` module allows the Python programmer to create weak references to objects.

In the following, the term `referent` means the object which is referred to by a weak reference.

A weak reference to an object is not enough to keep the object alive: when the only remaining references to a referent are weak references, garbage collection is free to destroy the referent and reuse its memory for something else. However, until the object is actually destroyed the weak reference may return the object even if there are no strong references to it.

A primary use for weak references is to implement caches or mappings holding large objects, where it’s desired that a large object not be kept alive solely because it appears in a cache or mapping.

For example, if you have a number of large binary image objects, you may wish to associate a name with each. If you used a Python dictionary to map names to images, or images to names, the image objects would remain alive just because they appeared as values or keys in the dictionaries. The `WeakKeyDictionary` and `WeakValueDictionary` classes supplied by the `weakref` module are an alternative, using weak references to construct mappings that don’t keep objects alive solely because they appear in the mapping objects. If, for example, an image object is a value in a `WeakValueDictionary`, then when the last remaining references to that image object are the weak references held by weak mappings, garbage collection can reclaim the object, and its corresponding entries in weak mappings are simply deleted.

`WeakKeyDictionary` and `WeakValueDictionary` use weak references in their implementation, setting up callback functions on the weak references that notify the weak dictionaries when a key or value has been reclaimed by garbage collection. `WeakSet` implements the `set` interface, but keeps weak references to its elements, just like a `WeakKeyDictionary` does.

Most programs should find that using one of these weak container types is all they need – it’s not usually necessary to create your own weak references directly. The low-level machinery used by the weak dictionary implementations is exposed by the `weakref` module for the benefit of advanced uses.

Not all objects can be weakly referenced; those objects which can include class instances, functions written in Python (but not in C), instance methods, sets, frozensets, some `file objects`, `generators`, type objects, sockets, arrays, deques, regular expression pattern objects, and code objects. Changed in version 3.2: Added support for `thread.lock`, `threading.Lock`, and code objects. Several built-in types such as `list` and `dict` do not directly support weak references but can add support through subclassing:

```python
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)  # this object is weak referenceable
```

Other built-in types such as `tuple` and `int` do not support weak references even when subclassed (This is an implementation detail and may be different across various Python implementations.).

Extension types can easily be made to support weak references; see `weakref-support`.

```python
class weakref.ref(object[, callback])
```

Return a weak reference to `object`. The original object can be retrieved by calling the reference object if the referent is still alive; if the referent is no longer alive, calling the reference object will cause `None` to be returned. If `callback` is provided and not `None`, and the returned weakref object is still alive, the callback will be called when the object is about to be finalized; the weak reference object will be passed as the only parameter to the callback; the referent will no longer be available.

It is allowable for many weak references to be constructed for the same object. Callbacks registered for each weak reference will be called from the most recently registered callback to the oldest registered callback.

Exceptions raised by the callback will be noted on the standard error output, but cannot be propagated; they are handled in exactly the same way as exceptions raised from an object’s `__del__()` method.
Weak references are hashable if the object is hashable. They will maintain their hash value even after the object was deleted. If hash() is called the first time only after the object was deleted, the call will raise TypeError.

Weak references support tests for equality, but not ordering. If the referents are still alive, two references have the same equality relationship as their referents (regardless of the callback). If either referent has been deleted, the references are equal only if the reference objects are the same object.

This is a subclassable type rather than a factory function.

```python
weakref.proxy(object[, callback])
```

Return a proxy to object which uses a weak reference. This supports use of the proxy in most contexts instead of requiring the explicit dereferencing used with weak reference objects. The returned object will have a type of either ProxyType or CallableProxyType, depending on whether object is callable. Proxy objects are not hashable regardless of the referent; this avoids a number of problems related to their fundamentally mutable nature, and prevent their use as dictionary keys. callback is the same as the parameter of the same name to the ref() function.

```python
weakref.getweakrefcount(object)
```

Return the number of weak references and proxies which refer to object.

```python
weakref.getweakrefs(object)
```

Return a list of all weak reference and proxy objects which refer to object.

```python
class weakref.WeakKeyDictionary(dict)
```

Mapping class that references keys weakly. Entries in the dictionary will be discarded when there is no longer a strong reference to the key. This can be used to associate additional data with an object owned by other parts of an application without adding attributes to those objects. This can be especially useful with objects that override attribute accesses.

**Note:** Caution: Because a WeakKeyDictionary is built on top of a Python dictionary, it must not change size when iterating over it. This can be difficult to ensure for a WeakKeyDictionary because actions performed by the program during iteration may cause items in the dictionary to vanish “by magic” (as a side effect of garbage collection).

WeakKeyDictionary objects have the following additional methods. These expose the internal references directly. The references are not guaranteed to be “live” at the time they are used, so the result of calling the references needs to be checked before being used. This can be used to avoid creating references that will cause the garbage collector to keep the keys around longer than needed.

```python
WeakKeyDictionary.keyrefs()
```

Return an iterable of the weak references to the keys.

```python
class weakref.WeakValueDictionary(dict)
```

Mapping class that references values weakly. Entries in the dictionary will be discarded when no strong reference to the value exists any more.

**Note:** Caution: Because a WeakValueDictionary is built on top of a Python dictionary, it must not change size when iterating over it. This can be difficult to ensure for a WeakValueDictionary because actions performed by the program during iteration may cause items in the dictionary to vanish “by magic” (as a side effect of garbage collection).

WeakValueDictionary objects have the following additional methods. These method have the same issues as the and keyrefs() method of WeakKeyDictionary objects.

```python
WeakValueDictionary.valuerefs()
```

Return an iterable of the weak references to the values.

```python
class weakref.WeakSet(elements)
```

Set class that keeps weak references to its elements. An element will be discarded when no strong reference to it exists any more.

8.8. weakref — Weak references

---

191
weakref.**ReferenceType**
The type object for weak references objects.

weakref.**ProxyType**
The type object for proxies of objects which are not callable.

weakref.**CallableProxyType**
The type object for proxies of callable objects.

weakref.**ProxyTypes**
Sequence containing all the type objects for proxies. This can make it simpler to test if an object is a proxy without being dependent on naming both proxy types.

**exception weakref.**ReferenceError
Exception raised when a proxy object is used but the underlying object has been collected. This is the same as the standard ReferenceError exception.

See Also:

PEP 0205 - Weak References The proposal and rationale for this feature, including links to earlier implementations and information about similar features in other languages.

### 8.8.1 Weak Reference Objects

Weak reference objects have no attributes or methods, but do allow the referent to be obtained, if it still exists, by calling it:

```python
>>> import weakref
>>> class Object:
...     pass
...>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

If the referent no longer exists, calling the reference object returns None:

```python
>>> del o, o2
>>> print(r())
None
```

Testing that a weak reference object is still live should be done using the expression `ref() is not None`. Normally, application code that needs to use a reference object should follow this pattern:

```python
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print("Object has been deallocated; can’t frobnicate.")
else:
    print("Object is still live!")
o.do_something_useful()
```

Using a separate test for “liveness” creates race conditions in threaded applications; another thread can cause a weak reference to become invalidated before the weak reference is called; the idiom shown above is safe in threaded applications as well as single-threaded applications.

Specialized versions of `ref` objects can be created through subclassing. This is used in the implementation of the WeakValueDictionary to reduce the memory overhead for each entry in the mapping. This may be most useful to associate additional information with a reference, but could also be used to insert additional processing on calls to retrieve the referent.
This example shows how a subclass of \texttt{ref} can be used to store additional information about an object and affect the value that’s returned when the referent is accessed:

```python
import weakref

class ExtendedRef(weakref.ref):
    def \_init\_(self, ob, callback=\texttt{None}, **annotations):
        super(ExtendedRef, self).\_init\_(ob, callback)
        self.\_counter = 0
        for k, v in annotations.items():
            setattr(self, k, v)

    def \_call\_(self):
        """Return a pair containing the referent and the number of times the reference has been called."
        ob = super(ExtendedRef, self).\_call\_()
        if ob is not \texttt{None}:
            self.\_counter += 1
            ob = (ob, self.\_counter)
        return ob
```

\subsection{8.8.2 Example}

This simple example shows how an application can use objects IDs to retrieve objects that it has seen before. The IDs of the objects can then be used in other data structures without forcing the objects to remain alive, but the objects can still be retrieved by ID if they do.

```python
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```

\section{8.9 \texttt{types} — Dynamic type creation and names for built-in types}

\textbf{Source code:} Lib/types.py

This module defines utility function to assist in dynamic creation of new types.

It also defines names for some object types that are used by the standard Python interpreter, but not exposed as builtins like \texttt{int} or \texttt{str} are.

\subsection{8.9.1 Dynamic Type Creation}

\texttt{types.new\_class} \((name, bases=(), kwds=\texttt{None}, exec\_body=\texttt{None})\)

Creates a class object dynamically using the appropriate metaclass.

The first three arguments are the components that make up a class definition header: the class name, the base classes (in order), the keyword arguments (such as \texttt{metaclass}).
The `exec_body` argument is a callback that is used to populate the freshly created class namespace. It should accept the class namespace as its sole argument and update the namespace directly with the class contents. If no callback is provided, it has the same effect as passing in `lambda ns: ns`. New in version 3.3.

```
prepare_class(name, bases=(), kwds=None)
```

Calculates the appropriate metaclass and creates the class namespace.

The arguments are the components that make up a class definition header: the class name, the base classes (in order) and the keyword arguments (such as `metaclass`).

The return value is a 3-tuple: `metaclass, namespace, kwds`

`metaclass` is the appropriate metaclass, `namespace` is the prepared class namespace and `kwds` is an updated copy of the passed in `kwds` argument with any `‘metaclass’` entry removed. If no `kwds` argument is passed in, this will be an empty dict. New in version 3.3.

See Also:

```
metaclasses  Full details of the class creation process supported by these functions
```

```
PEP 3115 - Metaclasses in Python 3000  Introduced the __prepare__ namespace hook
```

### 8.9.2 Standard Interpreter Types

This module provides names for many of the types that are required to implement a Python interpreter. It deliberately avoids including some of the types that arise only incidentally during processing such as the `listiterator` type.

Typical use of these names is for `isinstance()` or `issubclass()` checks.

Standard names are defined for the following types:

```
FunctionType
```

The type of user-defined functions and functions created by `lambda` expressions.

```
LambdaType
```

The type of `generator`-iterator objects, produced by calling a generator function.

```
CodeType
```

The type for code objects such as returned by `compile()`.

```
MethodType
```

The type of methods of user-defined class instances.

```
BuiltinFunctionType
```

The type of built-in functions like `len()` or `sys.exit()`, and methods of built-in classes. (Here, the term “built-in” means “written in C”.)

```
ModuleType
```

The type of modules.

```
TracebackType
```

The type of traceback objects such as found in `sys.exc_info()`[2].

```
FrameType
```

The type of frame objects such as found in `tb.tb_frame` if `tb` is a traceback object.

```
GetSetDescriptorType
```

The type of objects defined in extension modules with `PyGetSetDef`, such as `FrameType.f_locals` or `array.array.typecode`. This type is used as descriptor for object attributes; it has the same purpose as the `property` type, but for classes defined in extension modules.

```
MemberDescriptorType
```

The type of objects defined in extension modules with `PyMemberDef`, such as `datetime.timedelta.days`. This type is used as descriptor for simple C data members which
use standard conversion functions; it has the same purpose as the property type, but for classes defined in extension modules.

CPython implementation detail: In other implementations of Python, this type may be identical to GetSetDescriptorType.

class types.MappingProxyType(mapping)
Read-only proxy of a mapping. It provides a dynamic view on the mapping’s entries, which means that when the mapping changes, the view reflects these changes. New in version 3.3.

key in proxy
Return True if the underlying mapping has a key key, else False.

proxy[key]
Return the item of the underlying mapping with key key. Raises a KeyError if key is not in the underlying mapping.

iter(proxy)
Return an iterator over the keys of the underlying mapping. This is a shortcut for iter(proxy.keys()).

len(proxy)
Return the number of items in the underlying mapping.

copy()
Return a shallow copy of the underlying mapping.

get(key[, default])
Return the value for key if key is in the underlying mapping, else default. If default is not given, it defaults to None, so that this method never raises a KeyError.

items()
Return a new view of the underlying mapping’s items ((key, value) pairs).

keys()
Return a new view of the underlying mapping’s keys.

values()
Return a new view of the underlying mapping’s values.

class types.SimpleNamespace
A simple object subclass that provides attribute access to its namespace, as well as a meaningful repr.

Unlike object, with SimpleNamespace you can add and remove attributes. If a SimpleNamespace object is initialized with keyword arguments, those are directly added to the underlying namespace.

The type is roughly equivalent to the following code:

class SimpleNamespace:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)
    def __repr__(self):
        keys = sorted(self.__dict__)
        items = "{}={!r}".format(k, self.__dict__[k]) for k in keys
        return "{}({})".format(type(self).__name__, ", ".join(items))

SimpleNamespace may be useful as a replacement for class NS: pass. However, for a structured record type use namedtuple() instead. New in version 3.3.

8.10 copy — Shallow and deep copy operations

Assignment statements in Python do not copy objects, they create bindings between a target and an object. For collections that are mutable or contain mutable items, a copy is sometimes needed so one can change one copy
without changing the other. This module provides generic shallow and deep copy operations (explained below).

Interface summary:

```python
copy.copy(x)
    Return a shallow copy of x.

copy.deepcopy(x)
    Return a deep copy of x.
```

```python
exception copy.error
    Raised for module specific errors.
```

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A **shallow copy** constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
- A **deep copy** constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

Two problems often exist with deep copy operations that don’t exist with shallow copy operations:

- Recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) may cause a recursive loop.
- Because deep copy copies everything it may copy too much, e.g., administrative data structures that should be shared even between copies.

The `deepcopy()` function avoids these problems by:

- keeping a “memo” dictionary of objects already copied during the current copying pass; and
- letting user-defined classes override the copying operation or the set of components copied.

This module does not copy types like module, method, stack trace, stack frame, file, socket, window, array, or any similar types. It does “copy” functions and classes (shallow and deeply), by returning the original object unchanged; this is compatible with the way these are treated by the `pickle` module.

Shallow copies of dictionaries can be made using `dict.copy()`, and of lists by assigning a slice of the entire list, for example, `copied_list = original_list[:]`.

Classes can use the same interfaces to control copying that they use to control pickling. See the description of module `pickle` for information on these methods. In fact, `copy` module uses the registered pickle functions from `copyreg` module.

In order for a class to define its own copy implementation, it can define special methods `__copy__()` and `__deepcopy__()`. The former is called to implement the shallow copy operation; no additional arguments are passed. The latter is called to implement the deep copy operation; it is passed one argument, the memo dictionary. If the `__deepcopy__()` implementation needs to make a deep copy of a component, it should call the `deepcopy()` function with the component as first argument and the memo dictionary as second argument.

See Also:

Module `pickle` Discussion of the special methods used to support object state retrieval and restoration.

## 8.11 pprint — Data pretty printer

**Source code:** Lib/pprint.py

The `pprint` module provides a capability to “pretty-print” arbitrary Python data structures in a form which can be used as input to the interpreter. If the formatted structures include objects which are not fundamental Python types, the representation may not be loadable. This may be the case if objects such as files, sockets, classes, or instances are included, as well as many other built-in objects which are not representable as Python constants.
The formatted representation keeps objects on a single line if it can, and breaks them onto multiple lines if they don’t fit within the allowed width. Construct `PrettyPrinter` objects explicitly if you need to adjust the width constraint.

Dictionaries are sorted by key before the display is computed.

The `pprint` module defines one class:

```python
class pprint.PrettyPrinter (indent=1, width=80, depth=None, stream=None)
```

Construct a `PrettyPrinter` instance. This constructor understands several keyword parameters. An output stream may be set using the `stream` keyword; the only method used on the stream object is the file protocol’s `write()` method. If not specified, the `PrettyPrinter` adopts `sys.stdout`. Three additional parameters may be used to control the formatted representation. The keywords are `indent`, `depth`, and `width`. The amount of indentation added for each recursive level is specified by `indent`; the default is one. Other values can cause output to look a little odd, but can make nesting easier to spot. The number of levels which may be printed is controlled by `depth`; if the data structure being printed is too deep, the next contained level is replaced by `...`. By default, there is no constraint on the depth of the objects being formatted. The desired output width is constrained using the `width` parameter; the default is 80 characters. If a structure cannot be formatted within the constrained width, a best effort will be made.

```python
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[ ['spam', 'eggs', 'lumberjack', 'knights', 'ni'],
  'spam',
  'eggs',
  'lumberjack',
  'knights',
  'ni']
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead', ...
  ('parrot', ('fresh fruit',)))))))
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead', (...))))))
```

The `PrettyPrinter` class supports several derivative functions:

- `pprint.pformat(object, indent=1, width=80, depth=None)`
  Return the formatted representation of `object` as a string. `indent`, `width` and `depth` will be passed to the `PrettyPrinter` constructor as formatting parameters.

- `pprint.pprint(object, stream=None, indent=1, width=80, depth=None)`
  Prints the formatted representation of `object` on `stream`, followed by a newline. If `stream` is `None`, `sys.stdout` is used. This may be used in the interactive interpreter instead of the `print()` function for inspecting values (you can even reassign `print = pprint.pprint` for use within a scope). `indent`, `width` and `depth` will be passed to the `PrettyPrinter` constructor as formatting parameters.

```python
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
<Recursion on list with id=...>,
  'spam',
  'eggs',
  'lumberjack',
  'knights',
  'ni']
```

8.11. `pprint` — Data pretty printer 197
pprint\.isreadable\(\text{object}\)
Determine if the formatted representation of \text{object} is “readable,” or can be used to reconstruct the value using \text{eval()}. This always returns \text{False} for recursive objects.

```python
>>> pprint.isreadable(stuff)
False
```

pprint\.isrecursive\(\text{object}\)
Determine if \text{object} requires a recursive representation.

One more support function is also defined:

pprint\.saferepr\(\text{object}\)
Return a string representation of \text{object}, protected against recursive data structures. If the representation of \text{object} exposes a recursive entry, the recursive reference will be represented as \text{<Recursion on typename with id=number>}. The representation is not otherwise formatted.

```python
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs', 'lumberjack', 'knights', 'ni']"
```

### 8.11.1 PrettyPrinter Objects

PrettyPrinter instances have the following methods:

PrettyPrinter\.pformat\(\text{object}\)
Return the formatted representation of \text{object}. This takes into account the options passed to the PrettyPrinter constructor.

PrettyPrinter\.pprint\(\text{object}\)
Print the formatted representation of \text{object} on the configured stream, followed by a newline.

The following methods provide the implementations for the corresponding functions of the same names. Using these methods on an instance is slightly more efficient since new PrettyPrinter objects don’t need to be created.

PrettyPrinter\.isreadable\(\text{object}\)
Determine if the formatted representation of the object is “readable,” or can be used to reconstruct the value using \text{eval()}. Note that this returns \text{False} for recursive objects. If the \text{depth} parameter of the PrettyPrinter is set and the object is deeper than allowed, this returns \text{False}.

PrettyPrinter\.isrecursive\(\text{object}\)
Determine if the object requires a recursive representation.

This method is provided as a hook to allow subclasses to modify the way objects are converted to strings. The default implementation uses the internals of the saferepr() implementation.

PrettyPrinter\.format\(\text{object}, \text{context}, \text{maxlevels}, \text{level}\)
Returns three values: the formatted version of \text{object} as a string, a flag indicating whether the result is readable, and a flag indicating whether recursion was detected. The first argument is the object to be presented. The second is a dictionary which contains the \text{id() }of objects that are part of the current presentation context (direct and indirect containers for \text{object} that are affecting the presentation) as the keys; if an object needs to be presented which is already represented in \text{context}, the third return value should be \text{True}. Recursive calls to the format() method should add additional entries for containers to this dictionary. The third argument, \text{maxlevels}, gives the requested limit to recursion; this will be \text{0} if there is no requested limit. This argument should be passed unmodified to recursive calls. The fourth argument, \text{level}, gives the current level; recursive calls should be passed a value less than that of the current call.

### 8.11.2 Example

To demonstrate several uses of the pprint() function and its parameters, let’s fetch information about a project from PyPI:
>>> import json
>>> import pprint
>>> from urllib.request import urlopen
>>> with urlopen('http://pypi.python.org/pypi/configparser/json') as url:
...     http_info = url.info()
...     raw_data = url.read().decode(http_info.get_content_charset())
>>> project_info = json.loads(raw_data)
>>> result = {'headers': http_info.items(), 'body': project_info}

In its basic form, pprint() shows the whole object:

>>> pprint.pprint(result)
{'body': {'info': {'_pypi_hidden': False,
'  _pypi_ordering': 12,
'  classifiers': ['Development Status :: 4 - Beta',
'    Intended Audience :: Developers',
'    License :: OSI Approved :: MIT License',
'    Natural Language :: English',
'    Operating System :: OS Independent',
'    Programming Language :: Python',
'    Programming Language :: Python :: 2',
'    Programming Language :: Python :: 2.6',
'    Programming Language :: Python :: 2.7',
'    Topic :: Software Development :: Libraries',
'    Topic :: Software Development :: Libraries :: Python Modules'],
'    download_url': 'UNKNOWN',
'    home_page': 'http://docs.python.org/py3k/library/configparser.html',
'    keywords': 'configparser ini parsing conf cfg configuration file',
'    license': 'MIT',
'    name': 'configparser',
'    package_url': 'http://pypi.python.org/pypi/configparser',
'    platform': 'any',
'    release_url': 'http://pypi.python.org/pypi/configparser/3.2.0r3',
'    requires_python': None,
'    stable_version': None,
'    summary': 'This library brings the updated configparser from Python
     version: 3.2.0r3'},
'  urls': [{'comment_text': '',
    'downloads': 47,
    'filename': 'configparser-3.2.0r3.tar.gz',
    'has_sig': False,
    'md5_digest': '8500fd87c61ac0de328fc996fceb69b96',
    'packagetype': 'sdist',
    'python_version': 'source',
    'size': 32281,
    'upload_time': '2011-05-10T16:28:50',
    'url': 'http://pypi.python.org/packages/source/c/configparser/configparser-3.2.0r3.tar.gz'}],
'headers': [('Date', 'Sat, 14 May 2011 12:48:52 GMT'),
    ('Server', 'Apache/2.2.16 (Debian)'),
    ('Content-Disposition', 'inline'),
    ('Connection', 'close'),
    ('Transfer-Encoding', 'chunked'),
    ('Content-Type', 'application/json; charset=UTF-8')]}

The result can be limited to a certain depth (ellipsis is used for deeper contents):

>>> pprint.pprint(result, depth=3)
{'body': {'info': {'_pypi_hidden': False,
'  _pypi_ordering': 12,
'  classifiers': [...],
'    download_url': 'UNKNOWN',
'    home_page': 'http://docs.python.org/py3k/library/configparser.html',
    'keywords': 'configparser ini parsing conf cfg configuration file',
'    license': 'MIT',
'    name': 'configparser',
'    package_url': 'http://pypi.python.org/pypi/configparser',
'    platform': 'any',
'    release_url': 'http://pypi.python.org/pypi/configparser/3.2.0r3',
'    requires_python': None,
'    stable_version': None,
'    summary': 'This library brings the updated configparser from Python
     version: 3.2.0r3'},
'  urls': [{'comment_text': '',
    'downloads': 47,
    'filename': 'configparser-3.2.0r3.tar.gz',
    'has_sig': False,
    'md5_digest': '8500fd87c61ac0de328fc996fceb69b96',
    'packagetype': 'sdist',
    'python_version': 'source',
    'size': 32281,
    'upload_time': '2011-05-10T16:28:50',
    'url': 'http://pypi.python.org/packages/source/c/configparser/configparser-3.2.0r3.tar.gz'}],
'headers': [('Date', 'Sat, 14 May 2011 12:48:52 GMT'),
    ('Server', 'Apache/2.2.16 (Debian)'),
    ('Content-Disposition', 'inline'),
    ('Connection', 'close'),
    ('Transfer-Encoding', 'chunked'),
    ('Content-Type', 'application/json; charset=UTF-8')]}}
Additionally, maximum width can be suggested. If a long object cannot be split, the specified width will be exceeded:

```python
>>> pprint.pprint(result['headers'], width=30)
[('Date', 'Sat, 14 May 2011 12:48:52 GMT'),
 ('Server', 'Apache/2.2.16 (Debian)'),
 ('Content-Disposition', 'inline'),
 ('Connection', 'close'),
 ('Transfer-Encoding', 'chunked'),
 ('Content-Type', 'application/json; charset=UTF-8')]}
```

**8.12 reprlib — Alternate repr() implementation**

*Source code: Lib/reprlib.py*

The `reprlib` module provides a means for producing object representations with limits on the size of the resulting strings. This is used in the Python debugger and may be useful in other contexts as well.

This module provides a class, an instance, and a function:

```python
class reprlib.Repr
    Class which provides formatting services useful in implementing functions similar to the built-in repr(); size limits for different object types are added to avoid the generation of representations which are excessively long.
reprlib.aRepr
    This is an instance of Repr which is used to provide the repr() function described below. Changing the attributes of this object will affect the size limits used by repr() and the Python debugger.
reprlib.repr(obj)
    This is the repr() method of aRepr. It returns a string similar to that returned by the built-in function of the same name, but with limits on most sizes.
```
In addition to size-limiting tools, the module also provides a decorator for detecting recursive calls to \_\_repr\_\_() and substituting a placeholder string instead.

@\_\_reprlib\_.\_\_recursive\_repr\_(fillvalue="...")
Decorator for \_\_repr\_\_() methods to detect recursive calls within the same thread. If a recursive call is made, the fillvalue is returned, otherwise, the usual \_\_repr\_\_() call is made. For example:

```python
>>> class MyList(list):
...     @\_\_recursive\_repr()
...     def \_\_repr\_\_(self):
...         return '<' + '|'.join(map(repr, self)) + '>
... >>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

New in version 3.2.

### 8.12.1 Repr Objects

\_\_repr\_\_ instances provide several attributes which can be used to provide size limits for the representations of different object types, and methods which format specific object types.

**\_\_repr\_\_.\_\_maxlevel**
Depth limit on the creation of recursive representations. The default is 6.

**\_\_repr\_\_.\_\_maxdict**
**\_\_repr\_\_.\_\_maxlist**
**\_\_repr\_\_.\_\_maxtuple**
**\_\_repr\_\_.\_\_maxset**
**\_\_repr\_\_.\_\_maxfrozenset**
**\_\_repr\_\_.\_\_maxdeque**
**\_\_repr\_\_.\_\_maxarray**
Limits on the number of entries represented for the named object type. The default is 4 for maxdict, 5 for maxarray, and 6 for the others.

**\_\_repr\_\_.\_\_maxlong**
Maximum number of characters in the representation for an integer. Digits are dropped from the middle. The default is 40.

**\_\_repr\_\_.\_\_maxstring**
Limit on the number of characters in the representation of the string. Note that the “normal” representation of the string is used as the character source: if escape sequences are needed in the representation, these may be mangled when the representation is shortened. The default is 30.

**\_\_repr\_\_.\_\_maxother**
This limit is used to control the size of object types for which no specific formatting method is available on the \_\_repr\_\_ object. It is applied in a similar manner as maxstring. The default is 20.

**\_\_repr\_\_.\_\_repr\_(obj)**
The equivalent to the built-in \_\_repr\_\_() that uses the formatting imposed by the instance.

**\_\_repr\_\_.\_\_repr\_1\_(obj, level)**
Recursive implementation used by \_\_repr\_\_. The this uses the type of obj to determine which formatting method to call, passing it obj and level. The type-specific methods should call \_\_repr\_1\_() to perform recursive formatting, with level - 1 for the value of level in the recursive call.

**\_\_repr\_\_.\_\_repr\_\_TYPE\_\_(obj, level)**
Formatting methods for specific types are implemented as methods with a name based on the type name. In the method name, TYPE is replaced by string.join(string.split(type(obj).__name__, 8.12.\_

---

8.12. reprlib — Alternate \_\_repr\_\_() implementation
Dispatch to these methods is handled by `repr1()`. Type-specific methods which need to recursively format a value should call `self.repr1(subobj, level - 1)`.

### 8.12.2 Subclassing Repr Objects

The use of dynamic dispatching by `Repr.repr1()` allows subclasses of `Repr` to add support for additional built-in object types or to modify the handling of types already supported. This example shows how special support for file objects could be added:

```python
import reprlib
import sys

class MyRepr(reprlib.Repr):
    def repr_file(self, obj, level):
        if obj.name in ['<stdin>', '<stdout>', '<stderr>']:
            return obj.name
        else:
            return repr(obj)

aRepr = MyRepr()
print(aRepr.repr(sys.stdin))  # prints '<stdin>'
```

202 Chapter 8. Data Types
The modules described in this chapter provide numeric and math-related functions and data types. The numbers module defines an abstract hierarchy of numeric types. The math and cmath modules contain various mathematical functions for floating-point and complex numbers. The decimal module supports exact representations of decimal numbers, using arbitrary precision arithmetic.

The following modules are documented in this chapter:

### 9.1 numbers — Numeric abstract base classes

The numbers module (PEP 3141) defines a hierarchy of numeric abstract base classes which progressively define more operations. None of the types defined in this module can be instantiated.

#### class numbers.Number

The root of the numeric hierarchy. If you just want to check if an argument \( x \) is a number, without caring what kind, use `isinstance(x, Number)`.

#### 9.1.1 The numeric tower

**class numbers.Complex**

Subclasses of this type describe complex numbers and include the operations that work on the built-in complex type. These are: conversions to `complex` and `bool`, `real`, `imag`, `+`, `-`, `*`, `/`, `abs()`, `conjugate()`, `==`, and `!=`. All except `-` and `!=` are abstract.

- **real**
  Abstract. Retrieves the real component of this number.

- **imag**
  Abstract. Retrieves the imaginary component of this number.

- **conjugate()**
  Abstract. Returns the complex conjugate. For example, \((1+3j).conjugate() == (1-3j)\).

**class numbers.Real**

To `Complex`, `Real` adds the operations that work on real numbers.

In short, those are: a conversion to `float`, `math.trunc()`, `round()`, `math.floor()`, `math.ceil()`, `divmod()`, `//`, `%`, `<`, `<=`, `>`, `>=`.

Real also provides defaults for `complex()`, `real`, `imag`, and `conjugate()`.

**class numbers.Rational**

Subtypes `Real` and adds `numerator` and `denominator` properties, which should be in lowest terms. With these, it provides a default for `float()`.
numerator
Abstract.
denominator
Abstract.

class numbers.Integral
Subtypes Rational and adds a conversion to int. Provides defaults for float(), numerator, and denominator. Adds abstract methods for ** and bit-string operations: <<, >>, &, ^, |, ~.

9.1.2 Notes for type implementors

Implementors should be careful to make equal numbers equal and hash them to the same values. This may be subtle if there are two different extensions of the real numbers. For example, fractions.Fraction implements hash() as follows:

```python
def __hash__(self):
    if self.denominator == 1:
        # Get integers right.
        return hash(self.numerator)
    # Expensive check, but definitely correct.
    if self == float(self):
        return hash(float(self))
    else:
        # Use tuple’s hash to avoid a high collision rate on
        # simple fractions.
        return hash((self.numerator, self.denominator))
```

Adding More Numeric ABCs

There are, of course, more possible ABCs for numbers, and this would be a poor hierarchy if it precluded the possibility of adding those. You can add MyFoo between Complex and Real with:

```python
class MyFoo(Complex): ...
MyFoo.register(Real)
```

Implementing the arithmetic operations

We want to implement the arithmetic operations so that mixed-mode operations either call an implementation whose author knew about the types of both arguments, or convert both to the nearest built in type and do the operation there. For subtypes of Integral, this means that __add__() and __radd__() should be defined as:

```python
class MyIntegral(Integral):
    def __add__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(self, other)
        else:
            return NotImplemented

def __radd__(self, other):
    if isinstance(other, MyIntegral):
        return do_my_adding_stuff(other, self)
    elif isinstance(other, OtherTypeIKnowAbout):
        return do_my_other_adding_stuff(other, self)
```

204 Chapter 9. Numeric and Mathematical Modules
elif isinstance(other, Integral):
    return int(other) + int(self)
elif isinstance(other, Real):
    return float(other) + float(self)
elif isinstance(other, Complex):
    return complex(other) + complex(self)
else:
    return NotImplemented

There are 5 different cases for a mixed-type operation on subclasses of `Complex`. I’ll refer to all of the above code that doesn’t refer to `MyIntegral` and `OtherTypeIKnowAbout` as “boilerplate”. `a` will be an instance of `A`, which is a subtype of `Complex(a : A <: Complex)`, and `b : B <: Complex`. I’ll consider `a + b`:

1. If `A` defines an `__add__()` which accepts `b`, all is well.
2. If `A` falls back to the boilerplate code, and it were to return a value from `__add__()`, we’d miss the possibility that `B` defines a more intelligent `__radd__()` so the boilerplate should return `NotImplemented` from `__add__()`. (Or `A` may not implement `__add__()` at all.)
3. Then `B`’s `__radd__()` gets a chance. If it accepts `a`, all is well.
4. If it falls back to the boilerplate, there are no more possible methods to try, so this is where the default implementation should live.
5. If `B <: A`, Python tries `B.__radd__` before `A.__add__`. This is ok, because it was implemented with knowledge of `A`, so it can handle those instances before delegating to `Complex`.

If `A <: Complex` and `B <: Real` without sharing any other knowledge, then the appropriate shared operation is the one involving the built in `complex`, and both `__radd__()` s land there, so `a+b == b+a`.

Because most of the operations on any given type will be very similar, it can be useful to define a helper function which generates the forward and reverse instances of any given operator. For example, `fractions.Fraction` uses:

```python
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
        else:
            return NotImplemented
    forward.__name__ = '__' + fallback_operator.__name__ + '__'
    forward.__doc__ = monomorphic_operator.__doc__

    def reverse(b, a):
        if isinstance(a, Rational):
            # Includes ints.
            return monomorphic_operator(a, b)
        elif isinstance(a, numbers.Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, numbers.Complex):
            return fallback_operator(complex(a), complex(b))
        else:
            return NotImplemented
    reverse.__name__ = '__r' + fallback_operator.__name__ + '__'
    reverse.__doc__ = monomorphic_operator.__doc__

    return forward, reverse
```

`9.1. numbers — Numeric abstract base classes`
```python
def _add(a, b):
    """a + b""
    return Fraction(a.numerator * b.denominator +
                   b.numerator * a.denominator,
                   a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(_add, operator.add)
```

# ...

## 9.2 math — Mathematical functions

This module is always available. It provides access to the mathematical functions defined by the C standard.

These functions cannot be used with complex numbers; use the functions of the same name from the `cmath` module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don’t is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.

### 9.2.1 Number-theoretic and representation functions

- **math.ceil(x)**
  Return the ceiling of `x`, the smallest integer greater than or equal to `x`. If `x` is not a float, delegates to `x.__ceil__()`, which should return an `Integral` value.

- **math.copysign(x, y)**
  Return `x` with the sign of `y`. On a platform that supports signed zeros, `copysign(1.0, -0.0)` returns `-1.0`.

- **math.fabs(x)**
  Return the absolute value of `x`.

- **math.factorial(x)**
  Return `x` factorial. Raises `ValueError` if `x` is not integral or is negative.

- **math.floor(x)**
  Return the floor of `x`, the largest integer less than or equal to `x`. If `x` is not a float, delegates to `x.__floor__()`, which should return an `Integral` value.

- **math.fmod(x, y)**
  Return `fmod(x, y)`, as defined by the platform C library. Note that the Python expression `x % y` may not return the same result. The intent of the C standard is that `fmod(x, y)` be exactly (mathematically; to infinite precision) equal to `x - n*y` for some integer `n` such that the result has the same sign as `x` and magnitude less than `abs(y)`. Python’s `x % y` returns a result with the sign of `y` instead, and may not be exactly computable for float arguments. For example, `fmod(-1e-100, 1e100)` is `-1e-100`, but the result of Python’s `-1e-100 % 1e100` is `1e100-1e-100`, which cannot be represented exactly as a float, and rounds to the surprising `1e100`. For this reason, function `fmod()` is generally preferred when working with floats, while Python’s `x % y` is preferred when working with integers.

- **math.frexp(x)**
  Return the mantissa and exponent of `x` as the pair `(m, e)`. `m` is a float and `e` is an integer such that `x == m * 2**e` exactly. If `x` is zero, returns `(0.0, 0)`, otherwise `0.5 <= abs(m) < 1`. This is used to “pick apart” the internal representation of a float in a portable way.
The Python Library Reference, Release 3.3.3

math.\texttt{fsum}(\texttt{iterable})

Return an accurate floating point sum of values in the iterable. Avoids loss of precision by tracking multiple intermediate partial sums:

\begin{verbatim}
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>> fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
1.0
\end{verbatim}

The algorithm’s accuracy depends on IEEE-754 arithmetic guarantees and the typical case where the rounding mode is half-even. On some non-Windows builds, the underlying C library uses extended precision addition and may occasionally double-round an intermediate sum causing it to be off in its least significant bit.

For further discussion and two alternative approaches, see the ASPN cookbook recipes for accurate floating point summation.

math.\texttt{isfinite}(x)

Return True if \( x \) is neither an infinity nor a NaN, and False otherwise. (Note that 0.0 is considered finite.) New in version 3.2.

math.\texttt{isinf}(x)

Return True if \( x \) is a positive or negative infinity, and False otherwise.

math.\texttt{isnan}(x)

Return True if \( x \) is a NaN (not a number), and False otherwise.

math.\texttt{ldexp}(x, i)

Return \( x \times 2^i \). This is essentially the inverse of function \texttt{frexp()}. Note that \texttt{frexp()} and \texttt{modf()} have a different call/return pattern than their C equivalents: they take a single argument and return a pair of values, rather than returning their second return value through an ‘output parameter’ (there is no such thing in Python).

For the \texttt{ceil()}, \texttt{floor()}, and \texttt{modf()} functions, note that all floating-point numbers of sufficiently large magnitude are exact integers. Python floats typically carry no more than 53 bits of precision (the same as the platform C double type), in which case any float \( x \) with \( \text{abs}(x) \geq 2^{52} \) necessarily has no fractional bits.

9.2.2 Power and logarithmic functions

math.\texttt{exp}(x)

Return \( e \times x \).

math.\texttt{expm1}(x)

Return \( e^{x} - 1 \). For small \( x \), the subtraction in \( \text{exp}(x) - 1 \) can result in a significant loss of precision; the \texttt{expm1()} function provides a way to compute this quantity to full precision:

\begin{verbatim}
>>> from math import exp, expm1
>>> exp(1e-5) - 1 # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5) # result accurate to full precision
1.0000050000166668e-05
\end{verbatim}

New in version 3.2.
The Python Library Reference, Release 3.3.3

math.log(x[, base])

With one argument, return the natural logarithm of x (to base e).

With two arguments, return the logarithm of x to the given base, calculated as \( \log(x) / \log(\text{base}) \).

math.log1p(x)

Return the natural logarithm of \( 1+x \) (base e). The result is calculated in a way which is accurate for x near zero.

math.log2(x)

Return the base-2 logarithm of x. This is usually more accurate than \( \log(x, 2) \). New in version 3.3.

See Also:

int.bit_length() returns the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros.

math.log10(x)

Return the base-10 logarithm of x. This is usually more accurate than \( \log(x, 10) \).

math.pow(x, y)

Return \( x \) raised to the power \( y \). Exceptional cases follow Annex ‘F’ of the C99 standard as far as possible. In particular, \( \text{pow}(1.0, x) \) and \( \text{pow}(x, 0.0) \) always return 1.0, even when \( x \) is a zero or a NaN. If both \( x \) and \( y \) are finite, \( x \) is negative, and \( y \) is not an integer then \( \text{pow}(x, y) \) is undefined, and raises ValueError.

Unlike the built-in ** operator, math.pow() converts both its arguments to type float. Use ** or the built-in pow() function for computing exact integer powers.

math.sqrt(x)

Return the square root of \( x \).

9.2.3 Trigonometric functions

math.acos(x)

Return the arc cosine of \( x \), in radians.

math.asin(x)

Return the arc sine of \( x \), in radians.

math.atan(x)

Return the arc tangent of \( x \), in radians.

math.atan2(y, x)

Return the \( \arctan(y / x) \), in radians. The result is between \( -\pi \) and \( \pi \). The vector in the plane from the origin to point \((x, y)\) makes this angle with the positive X axis. The point of atan2() is that the signs of both inputs are known to it, so it can compute the correct quadrant for the angle. For example, \( \text{atan}(1) \) and \( \text{atan2}(1, 1) \) are both \( \pi/4 \), but \( \text{atan2}(-1, -1) \) is \( -3*\pi/4 \).

math.cos(x)

Return the cosine of \( x \) radians.

math.hypot(x, y)

Return the Euclidean norm, \( \sqrt{x^2 + y^2} \). This is the length of the vector from the origin to point \((x, y)\).

math.sin(x)

Return the sine of \( x \) radians.

math.tan(x)

Return the tangent of \( x \) radians.
9.2.4 Angular conversion

math.degrees(x)
Converts angle x from radians to degrees.

math.radians(x)
Converts angle x from degrees to radians.

9.2.5 Hyperbolic functions

Hyperbolic functions are analogs of trigonometric functions that are based on hyperbolas instead of circles.

math.acosh(x)
Return the inverse hyperbolic cosine of x.

math.asinh(x)
Return the inverse hyperbolic sine of x.

math.atanh(x)
Return the inverse hyperbolic tangent of x.

math.cosh(x)
Return the hyperbolic cosine of x.

math.sinh(x)
Return the hyperbolic sine of x.

math.tanh(x)
Return the hyperbolic tangent of x.

9.2.6 Special functions

math.erf(x)
Return the error function at x.

The erf() function can be used to compute traditional statistical functions such as the cumulative standard normal distribution:

```python
def phi(x):
    'Cumulative distribution function for the standard normal distribution'
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

New in version 3.2.

math.erfc(x)
Return the complementary error function at x. The complementary error function is defined as \(1.0 - \text{erf}(x)\). It is used for large values of x where a subtraction from one would cause a loss of significance. New in version 3.2.

math.gamma(x)
Return the Gamma function at x. New in version 3.2.

math.lgamma(x)
Return the natural logarithm of the absolute value of the Gamma function at x. New in version 3.2.

9.2.7 Constants

math.pi
The mathematical constant \(\pi = 3.141592...\), to available precision.
math.e

The mathematical constant e = 2.718281..., to available precision.

**CPython implementation detail:** The `math` module consists mostly of thin wrappers around the platform C math library functions. Behavior in exceptional cases follows Annex F of the C99 standard where appropriate. The current implementation will raise `ValueError` for invalid operations like `sqrt(-1.0)` or `log(0.0)` (where C99 Annex F recommends signaling invalid operation or divide-by-zero), and `OverflowError` for results that overflow (for example, `exp(1000.0)`). A `NaN` will not be returned from any of the functions above unless one or more of the input arguments was a `NaN`; in that case, most functions will return a `NaN`, but (again following C99 Annex F) there are some exceptions to this rule, for example `pow(float('nan'), 0.0)` or `hypot(float('nan'), float('inf'))`.

Note that Python makes no effort to distinguish signaling NaNs from quiet NaNs, and behavior for signaling NaNs remains unspecified. Typical behavior is to treat all NaNs as though they were quiet.

**See Also:**

Module `cmath` Complex number versions of many of these functions.

### 9.3 `cmath` — Mathematical functions for complex numbers

This module is always available. It provides access to mathematical functions for complex numbers. The functions in this module accept integers, floating-point numbers or complex numbers as arguments. They will also accept any Python object that has either a `__complex__()` or a `__float__()` method: these methods are used to convert the object to a complex or floating-point number, respectively, and the function is then applied to the result of the conversion.

**Note:** On platforms with hardware and system-level support for signed zeros, functions involving branch cuts are continuous on both sides of the branch cut: the sign of the zero distinguishes one side of the branch cut from the other. On platforms that do not support signed zeros, the continuity is as specified below.

#### 9.3.1 Conversions to and from polar coordinates

A Python complex number `z` is stored internally using rectangular or Cartesian coordinates. It is completely determined by its **real part** `z.real` and its **imaginary part** `z.imag`. In other words:

```
z == z.real + z.imag*1j
```

**Polar coordinates** give an alternative way to represent a complex number. In polar coordinates, a complex number `z` is defined by the **modulus** `r` and the **phase angle** `phi`. The modulus `r` is the distance from `z` to the origin, while the phase `phi` is the counterclockwise angle, measured in radians, from the positive x-axis to the line segment that joins the origin to `z`.

The following functions can be used to convert from the native rectangular coordinates to polar coordinates and back.

`cmath.phase(x)`

Return the phase of `x` (also known as the **argument** of `x`), as a float. `phase(x)` is equivalent to `math.atan2(x.imag, x.real)`. The result lies in the range \([-\pi, \pi]\), and the branch cut for this operation lies along the negative real axis, continuous from above. On systems with support for signed zeros (which includes most systems in current use), this means that the sign of the result is the same as the sign of `x.imag`, even when `x.imag` is zero:

```
>>> phase(complex(-1.0, 0.0))
3.141592653589793
>>> phase(complex(-1.0, -0.0))
-3.141592653589793
```
Note: The modulus (absolute value) of a complex number \( x \) can be computed using the built-in \( \text{abs()} \) function. There is no separate \( \text{cmath} \) module function for this operation.

\[
\text{cmath.polar}(x)
\]
Return the representation of \( x \) in polar coordinates. Returns a pair \((r, \phi)\) where \( r \) is the modulus of \( x \) and \( \phi \) is the phase of \( x \). \( \text{polar}(x) \) is equivalent to \((\text{abs}(x), \text{phase}(x))\).

\[
\text{cmath.rect}(r, \phi)
\]
Return the complex number \( x \) with polar coordinates \( r \) and \( \phi \). Equivalent to \( r * (\text{math.cos}(\phi) + \text{math.sin}(\phi)*1j) \).

### 9.3.2 Power and logarithmic functions

\[
\text{cmath.exp}(x)
\]
Return the exponential value \( e^x \).

\[
\text{cmath.log}(x[, base])
\]
Returns the logarithm of \( x \) to the given \( base \). If the \( base \) is not specified, returns the natural logarithm of \( x \). There is one branch cut, from 0 along the negative real axis to \(-\infty\), continuous from above.

\[
\text{cmath.log10}(x)
\]
Return the base-10 logarithm of \( x \). This has the same branch cut as \( \log() \).

\[
\text{cmath.sqrt}(x)
\]
Return the square root of \( x \). This has the same branch cut as \( \log() \).

### 9.3.3 Trigonometric functions

\[
\text{cmath.acos}(x)
\]
Return the arc cosine of \( x \). There are two branch cuts: One extends right from 1 along the real axis to \( \infty \), continuous from below. The other extends left from \(-1\) along the real axis to \(-\infty\), continuous from above.

\[
\text{cmath.asin}(x)
\]
Return the arc sine of \( x \). This has the same branch cuts as \( \text{acos()} \).

\[
\text{cmath.atan}(x)
\]
Return the arc tangent of \( x \). There are two branch cuts: One extends from \( 1j \) along the imaginary axis to \( \infty j \), continuous from the right. The other extends from \(-1j \) along the imaginary axis to \(-\infty j \), continuous from the left.

\[
\text{cmath.cos}(x)
\]
Return the cosine of \( x \).

\[
\text{cmath.sin}(x)
\]
Return the sine of \( x \).

\[
\text{cmath.tan}(x)
\]
Return the tangent of \( x \).

### 9.3.4 Hyperbolic functions

\[
\text{cmath.acosh}(x)
\]
Return the hyperbolic arc cosine of \( x \). There is one branch cut, extending left from \( 1 \) along the real axis to \(-\infty\), continuous from above.

\[
\text{cmath.asinh}(x)
\]
Return the hyperbolic arc sine of \( x \). There are two branch cuts: One extends from \( 1j \) along the imaginary axis to \( \infty j \), continuous from the right. The other extends from \(-1j \) along the imaginary axis to \(-\infty j \), continuous from the left.
```plaintext
cmath.atanh(x)
Return the hyperbolic arc tangent of x. There are two branch cuts: One extends from 1 along the real axis to ∞, continuous from below. The other extends from -1 along the real axis to -∞, continuous from above.

cmath.cosh(x)
Return the hyperbolic cosine of x.

cmath.sinh(x)
Return the hyperbolic sine of x.

cmath.tanh(x)
Return the hyperbolic tangent of x.

9.3.5 Classification functions

cmath.isfinite(x)
Return True if both the real and imaginary parts of x are finite, and False otherwise. New in version 3.2.

cmath.isinf(x)
Return True if either the real or the imaginary part of x is an infinity, and False otherwise.

cmath.isnan(x)
Return True if either the real or the imaginary part of x is a NaN, and False otherwise.

9.3.6 Constants

cmath.pi
The mathematical constant π, as a float.

cmath.e
The mathematical constant e, as a float.

Note that the selection of functions is similar, but not identical, to that in module math. The reason for having two modules is that some users aren’t interested in complex numbers, and perhaps don’t even know what they are. They would rather have math.sqrt(-1) raise an exception than return a complex number. Also note that the functions defined in cmath always return a complex number, even if the answer can be expressed as a real number (in which case the complex number has an imaginary part of zero).

A note on branch cuts: They are curves along which the given function fails to be continuous. They are a necessary feature of many complex functions. It is assumed that if you need to compute with complex functions, you will understand about branch cuts. Consult almost any (not too elementary) book on complex variables for enlightenment. For information of the proper choice of branch cuts for numerical purposes, a good reference should be the following:

See Also:

9.4 decimal — Decimal fixed point and floating point arithmetic

The decimal module provides support for fast correctly-rounded decimal floating point arithmetic. It offers several advantages over the float datatype:

- Decimal “is based on a floating-point model which was designed with people in mind, and necessarily has a paramount guiding principle – computers must provide an arithmetic that works in the same way as the arithmetic that people learn at school.” – excerpt from the decimal arithmetic specification.

- Decimal numbers can be represented exactly. In contrast, numbers like 1.1 and 2.2 do not have exact representations in binary floating point. End users typically would not expect 1.1 + 2.2 to display as 3.3000000000000003 as it does with binary floating point.
```
The exactness carries over into arithmetic. In decimal floating point, $0.1 + 0.1 + 0.1 - 0.3$ is exactly equal to zero. In binary floating point, the result is $5.5511151231257827e-017$. While near to zero, the differences prevent reliable equality testing and differences can accumulate. For this reason, decimal is preferred in accounting applications which have strict equality invariants.

The decimal module incorporates a notion of significant places so that $1.30 + 1.20$ is 2.50. The trailing zero is kept to indicate significance. This is the customary presentation for monetary applications. For multiplication, the “schoolbook” approach uses all the figures in the multiplicands. For instance, $1.3 * 1.2$ gives 1.56 while $1.30 * 1.20$ gives 1.5600.

Unlike hardware based binary floating point, the decimal module has a user alterable precision (defaulting to 28 places) which can be as large as needed for a given problem:

```python
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

Both binary and decimal floating point are implemented in terms of published standards. While the built-in float type exposes only a modest portion of its capabilities, the decimal module exposes all required parts of the standard. When needed, the programmer has full control over rounding and signal handling. This includes an option to enforce exact arithmetic by using exceptions to block any inexact operations.

The decimal module was designed to support “without prejudice, both exact unrounded decimal arithmetic (sometimes called fixed-point arithmetic) and rounded floating-point arithmetic.” – excerpt from the decimal arithmetic specification.

The module design is centered around three concepts: the decimal number, the context for arithmetic, and signals.

A decimal number is immutable. It has a sign, coefficient digits, and an exponent. To preserve significance, the coefficient digits do not truncate trailing zeros. Decimals also include special values such as Infinity, -Infinity, and NaN. The standard also differentiates -0 from +0.

The context for arithmetic is an environment specifying precision, rounding rules, limits on exponents, flags indicating the results of operations, and trap enablers which determine whether signals are treated as exceptions. Rounding options include ROUND_CEILING, ROUND_DOWN, ROUND_FLOOR, ROUND_HALF_DOWN, ROUND_HALF_EVEN, ROUND_HALF_UP, ROUND_UP, and ROUND_05UP.

Signals are groups of exceptional conditions arising during the course of computation. Depending on the needs of the application, signals may be ignored, considered as informational, or treated as exceptions. The signals in the decimal module are: Clamped, InvalidOperation, DivisionByZero, Inexact, Rounded, Subnormal, Overflow, Underflow and FloatOperation.

For each signal there is a flag and a trap enabler. When a signal is encountered, its flag is set to one, then, if the trap enabler is set to one, an exception is raised. Flags are sticky, so the user needs to reset them before monitoring a calculation.

See Also:


### 9.4.1 Quick-start Tutorial

The usual start to using decimals is importing the module, viewing the current context with `getcontext()` and, if necessary, setting new values for precision, rounding, or enabled traps:

```python
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
```
import capitals=1, clamp=0, flags=[], traps=[Overflow, DivisionByZero, InvalidOperation])

```python
>>> getcontext().prec = 7  # Set a new precision
```

Decimal instances can be constructed from integers, strings, floats, or tuples. Construction from an integer or a float performs an exact conversion of the value of that integer or float. Decimal numbers include special values such as NaN which stands for “Not a number”, positive and negative Infinity, and -0:

```python
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.14')
>>> Decimal('3.140000000000000012434497875801753252744674682617875')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.4142135623730951')
```

If the FloatOperation signal is trapped, accidental mixing of decimals and floats in constructors or ordering comparisons raises an exception:

```python
>>> c = getcontext()
>>> c.traps[FloatOperation] = True
>>> Decimal(3.14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') < 3.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') == 3.5
True
```

New in version 3.3. The significance of a new Decimal is determined solely by the number of digits input. Context precision and rounding only come into play during arithmetic operations.

```python
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

If the internal limits of the C version are exceeded, constructing a decimal raises InvalidOperation:

```python
>>> Decimal("1e99999999999999999")
Traceback (most recent call last):
```

214 Chapter 9. Numeric and Mathematical Modules
Decimals interact well with much of the rest of Python. Here is a small decimal floating point flying circus:

```python
>>> data = list(map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split()))

>>> max(data)
Decimal('9.25')

>>> min(data)
Decimal('0.03')

>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'), Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]

>>> sum(data)
Decimal('19.29')

>>> a, b, c = data[:3]

>>> str(a)
'1.34'

>>> float(a)
1.34

>>> round(a, 1)
Decimal('1.3')

>>> int(a)
1

>>> a * 5
Decimal('6.70')

>>> a * b
Decimal('2.5058')

>>> c % a
Decimal('0.77')
```

And some mathematical functions are also available to Decimal:

```python
>>> getcontext().prec = 28

>>> Decimal(2).sqrt()
Decimal('1.414213562373095048801688724')

>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')

>>> Decimal('10').ln()
Decimal('2.302585092994045684017991455')

>>> Decimal('10').log10()
Decimal('1')
```

The `quantize()` method rounds a number to a fixed exponent. This method is useful for monetary applications that often round results to a fixed number of places:

```python
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')

>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

As shown above, the `getcontext()` function accesses the current context and allows the settings to be changed. This approach meets the needs of most applications.

For more advanced work, it may be useful to create alternate contexts using the `Context()` constructor. To make an alternate active, use the `setcontext()` function.

In accordance with the standard, the `Decimal` module provides two ready to use standard contexts, `BasicContext` and `ExtendedContext`. The former is especially useful for debugging because many of the traps are enabled.
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571428571428571428571428571428571428571')

ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
capitals=1, clamp=0, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')

ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
capitals=1, clamp=0, flags=[], traps=[])
>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0

Individual traps are set using the dictionary in the traps field of a context:

Individual traps are set using the dictionary in the traps field of a context:

Individual traps are set using the dictionary in the traps field of a context:

Most programs adjust the current context only once, at the beginning of the program. And, in many applications, data is converted to Decimal with a single cast inside a loop. With context set and decimals created, the bulk of the program manipulates the data no differently than with other Python numeric types.

### 9.4.2 Decimal objects

**class decimal.Decimal(value="0", context=None)**

Construct a new Decimal object based from value.

value can be an integer, string, tuple, float, or another Decimal object. If no value is given, returns Decimal('0'). If value is a string, it should conform to the decimal numeric string syntax after leading and trailing whitespace characters are removed:
Other Unicode decimal digits are also permitted where digit appears above. These include decimal digits from various other alphabets (for example, Arabic-Indic and Devanagari digits) along with the fullwidth digits ‘０’ through ‘９’.

If value is a tuple, it should have three components, a sign (0 for positive or 1 for negative), a tuple of digits, and an integer exponent. For example, Decimal((0, (1, 4, 1, 4), -3)) returns Decimal(‘1.414’).

If value is a float, the binary floating point value is losslessly converted to its exact decimal equivalent. This conversion can often require 53 or more digits of precision. For example, Decimal(float(‘1.1’)) converts to Decimal(‘1.100000000000000088817841970012523233890533447265625’).

The context precision does not affect how many digits are stored. That is determined exclusively by the number of digits in value. For example, Decimal(‘3.00000’) records all five zeros even if the context precision is only three.

The purpose of the context argument is determining what to do if value is a malformed string. If the context traps InvalidOperation, an exception is raised; otherwise, the constructor returns a new Decimal with the value of NaN.

Once constructed, Decimal objects are immutable. Changed in version 3.2: The argument to the constructor is now permitted to be a float instance. Changed in version 3.3: float arguments raise an exception if the FloatOperation trap is set. By default the trap is off. Decimal floating point objects share many properties with the other built-in numeric types such as float and int. All of the usual math operations and special methods apply. Likewise, decimal objects can be copied, pickled, printed, used as dictionary keys, used as set elements, compared, sorted, and coerced to another type (such as float or int).

There are some small differences between arithmetic on Decimal objects and arithmetic on integers and floats. When the remainder operator % is applied to Decimal objects, the sign of the result is the sign of the dividend rather than the sign of the divisor:

>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal(‘-3’)

The integer division operator // behaves analogously, returning the integer part of the true quotient (truncating towards zero) rather than its floor, so as to preserve the usual identity \( x \equiv (x // y) \times y + x \mod y \):

>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal(‘-1’)

The % and // operators implement the remainder and divide-integer operations (respectively) as described in the specification.

Decimal objects cannot generally be combined with floats or instances of fractions.Fraction in arithmetic operations: an attempt to add a Decimal to a float, for example, will raise a TypeError.

9.4. decimal — Decimal fixed point and floating point arithmetic
However, it is possible to use Python’s comparison operators to compare a `Decimal` instance `x` with another number `y`. This avoids confusing results when doing equality comparisons between numbers of different types. Changed in version 3.2: Mixed-type comparisons between `Decimal` instances and other numeric types are now fully supported. In addition to the standard numeric properties, decimal floating point objects also have a number of specialized methods:

adjusted()  
Return the adjusted exponent after shifting out the coefficient’s rightmost digits until only the lead digit remains: `Decimal('321e+5').adjusted()` returns seven. Used for determining the position of the most significant digit with respect to the decimal point.

as_tuple()  
Return a named tuple representation of the number: `DecimalTuple(sign, digits, exponent)`.

canonical()  
Return the canonical encoding of the argument. Currently, the encoding of a `Decimal` instance is always canonical, so this operation returns its argument unchanged.

countopare(other, context=None)  
Compare the values of two Decimal instances. `compare()` returns a Decimal instance, and if either operand is a NaN then the result is a NaN:

```
   a or b is a NaN ==> Decimal('NaN')
   a < b            ==> Decimal('-1')
   a == b           ==> Decimal('0')
   a > b            ==> Decimal('1')
```

countopare_attemptal(other, context=None)  
This operation is identical to the `compare()` method, except that all NaNs signal. That is, if neither operand is a signaling NaN then any quiet NaN operand is treated as though it were a signaling NaN.

countopare_total(other, context=None)  
Compare two operands using their abstract representation rather than their numerical value. Similar to the `compare()` method, but the result gives a total ordering on `Decimal` instances. Two `Decimal` instances with the same numeric value but different representations compare unequal in this ordering:

```
   >>> Decimal('12.0').compare_total(Decimal('12'))
   Decimal('-1')
```

Quiet and signaling NaNs are also included in the total ordering. The result of this function is `Decimal('0')` if both operands have the same representation, `Decimal('-1')` if the first operand is lower in the total order than the second, and `Decimal('1')` if the first operand is higher in the total order than the second operand. See the specification for details of the total order.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

countopare_total_mag(other, context=None)  
Compare two operands using their abstract representation rather than their value as in `compare_total()`, but ignoring the sign of each operand. `x.compare_total_mag(y)` is equivalent to `x.copy_abs().compare_total(y.copy_abs())`.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

countuplate()  
Just returns self, this method is only to comply with the Decimal Specification.
copy_abs()  
Return the absolute value of the argument. This operation is unaffected by the context and is quiet: no 
flags are changed and no rounding is performed.

copy_negate()  
Return the negation of the argument. This operation is unaffected by the context and is quiet: no flags 
are changed and no rounding is performed.

copy_sign(other, context=None)  
Return a copy of the first operand with the sign set to be the same as the sign of the second operand. 
For example:

```python
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

This operation is unaffected by context and is quiet: no flags are changed and no rounding is per-
formed. As an exception, the C version may raise InvalidOperation if the second operand cannot be 
converted exactly.

exp(context=None)  
Return the value of the (natural) exponential function e**x at the given number. The result is correctly 
rounded using the ROUND_HALF_EVEN rounding mode.

```python
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

from_float()  
Classmethod that converts a float to a decimal number, exactly. 
Note: From Python 3.2 onwards, a Decimal instance can also be constructed directly from a 
float.

```python
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

New in version 3.1.

fma(other, third, context=None)  
Fused multiply-add. Return self*other+third with no rounding of the intermediate product self*other.

```python
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

is_canonical()  
Return True if the argument is canonical and False otherwise. Currently, a Decimal instance is 
always canonical, so this operation always returns True.

9.4. decimal — Decimal fixed point and floating point arithmetic 219
is_finite()  
Return True if the argument is a finite number, and False if the argument is an infinity or a NaN.

is_infinite()  
Return True if the argument is either positive or negative infinity and False otherwise.

is_nan()  
Return True if the argument is a (quiet or signaling) NaN and False otherwise.

is_normal (context=None)  
Return True if the argument is a normal finite number. Return False if the argument is zero, subnormal, infinite or a NaN.

is_qnan()  
Return True if the argument is a quiet NaN, and False otherwise.

is_signed()  
Return True if the argument has a negative sign and False otherwise. Note that zeros and NaNs can both carry signs.

is_snan()  
Return True if the argument is a signaling NaN and False otherwise.

is_subnormal (context=None)  
Return True if the argument is subnormal, and False otherwise.

is_zero()  
Return True if the argument is a (positive or negative) zero and False otherwise.

ln (context=None)  
Return the natural (base e) logarithm of the operand. The result is correctly rounded using the ROUND_HALF_EVEN rounding mode.

log10 (context=None)  
Return the base ten logarithm of the operand. The result is correctly rounded using the ROUND_HALF_EVEN rounding mode.

logb (context=None)  
For a nonzero number, return the adjusted exponent of its operand as a Decimal instance. If the operand is a zero then Decimal(‘-Infinity’) is returned and the DivisionByZero flag is raised. If the operand is an infinity then Decimal(‘Infinity’) is returned.

logical_and (other, context=None)  
logical_and() is a logical operation which takes two logical operands (see Logical operands). The result is the digit-wise and of the two operands.

logical_invert (context=None)  
logical_invert() is a logical operation. The result is the digit-wise inversion of the operand.

logical_or (other, context=None)  
logical_or() is a logical operation which takes two logical operands (see Logical operands). The result is the digit-wise or of the two operands.

logical_xor (other, context=None)  
logical_xor() is a logical operation which takes two logical operands (see Logical operands). The result is the digit-wise exclusive or of the two operands.

max (other, context=None)  
Like max(self, other) except that the context rounding rule is applied before returning and that NaN values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

max_mag (other, context=None)  
Similar to the max() method, but the comparison is done using the absolute values of the operands.

min (other, context=None)  
Like min(self, other) except that the context rounding rule is applied before returning and that
NaN values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

**min_mag**(other, context=None)
Similar to the **min()** method, but the comparison is done using the absolute values of the operands.

**next_minus**(context=None)
Return the largest number representable in the given context (or in the current thread’s context if no context is given) that is smaller than the given operand.

**next_plus**(context=None)
Return the smallest number representable in the given context (or in the current thread’s context if no context is given) that is larger than the given operand.

**next_toward**(other, context=None)
If the two operands are unequal, return the number closest to the first operand in the direction of the second operand. If both operands are numerically equal, return a copy of the first operand with the sign set to be the same as the sign of the second operand.

**normalize**(context=None)
Normalize the number by stripping the rightmost trailing zeros and converting any result equal to Decimal('0') to Decimal('0e0'). Used for producing canonical values for attributes of an equivalence class. For example, Decimal('32.100') and Decimal('0.321000e+2') both normalize to the equivalent value Decimal('32.1').

**number_class**(context=None)
Return a string describing the class of the operand. The returned value is one of the following ten strings.

- **"-Infinity"**, indicating that the operand is negative infinity.
- **"-Normal"**, indicating that the operand is a negative normal number.
- **"-Subnormal"**, indicating that the operand is negative and subnormal.
- **"-Zero"**, indicating that the operand is a negative zero.
- **"+Zero"**, indicating that the operand is a positive zero.
- **"+Subnormal"**, indicating that the operand is positive and subnormal.
- **"+Normal"**, indicating that the operand is a positive normal number.
- **"+Infinity"**, indicating that the operand is positive infinity.
- **"NaN"**, indicating that the operand is a quiet NaN (Not a Number).
- **"sNaN"**, indicating that the operand is a signaling NaN.

**quantize**(exp, rounding=None, context=None, watchexp=True)
Return a value equal to the first operand after rounding and having the exponent of the second operand.

```python
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

Unlike other operations, if the length of the coefficient after the quantize operation would be greater than precision, then an **InvalidOperation** is signaled. This guarantees that, unless there is an error condition, the quantized exponent is always equal to that of the right-hand operand.

Also unlike other operations, quantize never signals Underflow, even if the result is subnormal and inexact.

If the exponent of the second operand is larger than that of the first then rounding may be necessary. In this case, the rounding mode is determined by the **rounding** argument if given, else by the given **context** argument; if neither argument is given the rounding mode of the current thread’s context is used.
If `watchexp` is set (default), then an error is returned whenever the resulting exponent is greater than `Emax` or less than `Etiny`. Deprecated since version 3.3: `watchexp` is an implementation detail from the pure Python version and is not present in the C version. It will be removed in version 3.4, where it defaults to True.

**radix()**

Return `Decimal(10)`, the radix (base) in which the `Decimal` class does all its arithmetic. Included for compatibility with the specification.

**remainder_near (other, context=None)**

Return the remainder from dividing `self` by `other`. This differs from `self % other` in that the sign of the remainder is chosen so as to minimize its absolute value. More precisely, the return value is `self - n * other` where `n` is the integer nearest to the exact value of `self / other`, and if two integers are equally near then the even one is chosen.

If the result is zero then its sign will be the sign of `self`.

```python
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

**rotate (other, context=None)**

Return the result of rotating the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to rotate. If the second operand is positive then rotation is to the left; otherwise rotation is to the right. The coefficient of the first operand is padded on the left with zeros to length precision if necessary. The sign and exponent of the first operand are unchanged.

**same_quantum (other, context=None)**

Test whether `self` and `other` have the same exponent or whether both are NaN.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

**scaleb (other, context=None)**

Return the first operand with exponent adjusted by the second. Equivalently, return the first operand multiplied by `10**other`. The second operand must be an integer.

**shift (other, context=None)**

Return the result of shifting the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to shift. If the second operand is positive then the shift is to the left; otherwise the shift is to the right. Digits shifted into the coefficient are zeros. The sign and exponent of the first operand are unchanged.

**sqrt (context=None)**

Return the square root of the argument to full precision.

**to_eng_string (context=None)**

Convert to an engineering-type string.

Engineering notation has an exponent which is a multiple of 3, so there are up to 3 digits left of the decimal place. For example, converts `Decimal('123E+1')` to `Decimal('1.23E+3')`

**to_integral (rounding=None, context=None)**

Identical to the `to_integral_value()` method. The `to_integral` name has been kept for compatibility with older versions.
to_integral_exact (rounding=None, context=None)  
Round to the nearest integer, signaling Inexact or Rounded as appropriate if rounding occurs. The rounding mode is determined by the rounding parameter if given, else by the given context. If neither parameter is given then the rounding mode of the current context is used.

to_integral_value (rounding=None, context=None)  
Round to the nearest integer without signaling Inexact or Rounded. If given, applies rounding; otherwise, uses the rounding method in either the supplied context or the current context.

Logical operands

The logical_and(), logical_invert(), logical_or(), and logical_xor() methods expect their arguments to be logical operands. A logical operand is a Decimal instance whose exponent and sign are both zero, and whose digits are all either 0 or 1.

9.4.3 Context objects

Contexts are environments for arithmetic operations. They govern precision, set rules for rounding, determine which signals are treated as exceptions, and limit the range for exponents.

Each thread has its own current context which is accessed or changed using the getcontext() and setcontext() functions:

decimal.getcontext ()  
Return the current context for the active thread.

decimal.setcontext (c)  
Set the current context for the active thread to c.

You can also use the with statement and the localcontext() function to temporarily change the active context.

decimal.localcontext (ctx=None)  
Return a context manager that will set the current context for the active thread to a copy of ctx on entry to the with-statement and restore the previous context when exiting the with-statement. If no context is specified, a copy of the current context is used.

For example, the following code sets the current decimal precision to 42 places, performs a calculation, and then automatically restores the previous context:

```python
from decimal import localcontext
with localcontext () as ctx:
    ctx.prec = 42  # Perform a high precision calculation
    s = calculate_something ()
    s = +s  # Round the final result back to the default precision
```

New contexts can also be created using the Context constructor described below. In addition, the module provides three pre-made contexts:

class decimal.BasicContext  
This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to ROUND_HALF_UP. All flags are cleared. All traps are enabled (treated as exceptions) except Inexact, Rounded, and Subnormal.

Because many of the traps are enabled, this context is useful for debugging.

class decimal.ExtendedContext  
This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to ROUND_HALF_EVEN. All flags are cleared. No traps are enabled (so that exceptions are not raised during computations).
Because the traps are disabled, this context is useful for applications that prefer to have result value of NaN or Infinity instead of raising exceptions. This allows an application to complete a run in the presence of conditions that would otherwise halt the program.

```python
class decimal.DefaultContext
This context is used by the Context constructor as a prototype for new contexts. Changing a field (such a precision) has the effect of changing the default for new contexts created by the Context constructor.

This context is most useful in multi-threaded environments. Changing one of the fields before threads are started has the effect of setting system-wide defaults. Changing the fields after threads have started is not recommended as it would require thread synchronization to prevent race conditions.

In single threaded environments, it is preferable to not use this context at all. Instead, simply create contexts explicitly as described below.

The default values are prec=28, rounding=ROUND_HALF_EVEN, and enabled traps for Overflow, InvalidOperation, and DivisionByZero.
```

In addition to the three supplied contexts, new contexts can be created with the Context constructor.

```python
class decimal.Context (prec=None, rounding=None, Emin=None, Emax=None, capitals=None, clamp=None, flags=None, traps=None)
```

Creates a new context. If a field is not specified or is None, the default values are copied from the DefaultContext. If the flags field is not specified or is None, all flags are cleared.

prec is an integer in the range [1, MAX_PREC] that sets the precision for arithmetic operations in the context.

The rounding option is one of the constants listed in the section Rounding Modes.

The traps and flags fields list any signals to be set. Generally, new contexts should only set traps and leave the flags clear.

The Emin and Emax fields are integers specifying the outer limits allowable for exponents. Emin must be in the range [MIN_EMIN, 0]. Emax in the range [0, MAX_EMAX].

The capitals field is either 0 or 1 (the default). If set to 1, exponents are printed with a capital E; otherwise, a lowercase e is used: Decimal(‘6.02e+23’).

The clamp field is either 0 (the default) or 1. If set to 1, the exponent e of a Decimal instance representable in this context is strictly limited to the range Emin - prec + 1 <= e <= Emax - prec + 1. If clamp is 0 then a weaker condition holds: the adjusted exponent of the Decimal instance is at most Emax. When clamp is 1, a large normal number will, where possible, have its exponent reduced and a corresponding number of zeros added to its coefficient, in order to fit the exponent constraints; this preserves the value of the number but loses information about significant trailing zeros. For example:

```python
>>> Context(prec=6, Emax=999, clamp=1).create_decimal(‘1.23e999’)
Decimal(‘1.23000E+999’)  # Reduced exponent
```

A clamp value of 1 allows compatibility with the fixed-width decimal interchange formats specified in IEEE 754.

The Context class defines several general purpose methods as well as a large number of methods for doing arithmetic directly in a given context. In addition, for each of the Decimal methods described above (with the exception of the adjusted() and as_tuple() methods) there is a corresponding Context method. For example, for a Context instance C and Decimal instance x, C.exp(x) is equivalent to x.exp(context=C). Each Context method accepts a Python integer (an instance of int) anywhere that a Decimal instance is accepted.

```python
clear_flags ()
Resets all of the flags to 0.

clear_traps ()
Resets all of the traps to 0. New in version 3.3.

copy ()
Return a duplicate of the context.
```
copy_decimal(num)
Return a copy of the Decimal instance num.

create_decimal(num)
Creates a new Decimal instance from num but using self as context. Unlike the Decimal constructor, the context precision, rounding method, flags, and traps are applied to the conversion.

This is useful because constants are often given to a greater precision than is needed by the application. Another benefit is that rounding immediately eliminates unintended effects from digits beyond the current precision. In the following example, using unrounded inputs means that adding zero to a sum can change the result:

```python
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

This method implements the to-number operation of the IBM specification. If the argument is a string, no leading or trailing whitespace is permitted.

create_decimal_from_float(f)
Creates a new Decimal instance from a float f but rounding using self as the context. Unlike the Decimal.from_float() class method, the context precision, rounding method, flags, and traps are applied to the conversion.

```python
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(math.pi)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create_decimal_from_float(math.pi)
Traceback (most recent call last):
  ...
decimal.Inexact: None
```

New in version 3.1.

Etiny()
Returns a value equal to Emin - prec + 1 which is the minimum exponent value for subnormal results. When underflow occurs, the exponent is set to Etiny.

Etop()
Returns a value equal to Emax - prec + 1.

The usual approach to working with decimals is to create Decimal instances and then apply arithmetic operations which take place within the current context for the active thread. An alternative approach is to use context methods for calculating within a specific context. The methods are similar to those for the Decimal class and are only briefly recounted here.

abs(x)
Returns the absolute value of x.

add(x, y)
Return the sum of x and y.

canonical(x)
Returns the same Decimal object x.

compare(x, y)
Compares x and y numerically.

compare_signal(x, y)
Compares the values of the two operands numerically.
The Python Library Reference, Release 3.3.3

**compare_total** *(x, y)*

Compares two operands using their abstract representation.

**compare_total_mag** *(x, y)*

Compares two operands using their abstract representation, ignoring sign.

**copy_abs** *(x)*

Returns a copy of *x* with the sign set to 0.

**copy_negate** *(x)*

Returns a copy of *x* with the sign inverted.

**copy_sign** *(x, y)*

Copies the sign from *y* to *x*.

**divide** *(x, y)*

Return *x* divided by *y*.

**divide_int** *(x, y)*

Return *x* divided by *y*, truncated to an integer.

**divmod** *(x, y)*

Divides two numbers and returns the integer part of the result.

**exp** *(x)*

Returns *e** x*.

**fma** *(x, y, z)*

Returns *x* multiplied by *y*, plus *z*.

**is_canonical** *(x)*

Returns True if *x* is canonical; otherwise returns False.

**is_finite** *(x)*

Returns True if *x* is finite; otherwise returns False.

**is_infinite** *(x)*

Returns True if *x* is infinite; otherwise returns False.

**is_nan** *(x)*

Returns True if *x* is a qNaN or sNaN; otherwise returns False.

**is_normal** *(x)*

Returns True if *x* is a normal number; otherwise returns False.

**is_qnan** *(x)*

Returns True if *x* is a quiet NaN; otherwise returns False.

**is_signed** *(x)*

Returns True if *x* is negative; otherwise returns False.

**is_snan** *(x)*

Returns True if *x* is a signaling NaN; otherwise returns False.

**is_subnormal** *(x)*

Returns True if *x* is subnormal; otherwise returns False.

**is_zero** *(x)*

Returns True if *x* is a zero; otherwise returns False.

**ln** *(x)*

Returns the natural (base e) logarithm of *x*.

**log10** *(x)*

Returns the base 10 logarithm of *x*.

**logb** *(x)*

Returns the exponent of the magnitude of the operand’s MSD.
logical_and(x, y)
Applies the logical operation and between each operand's digits.

logical_invert(x)
Invert all the digits in x.

logical_or(x, y)
Applies the logical operation or between each operand's digits.

logical_xor(x, y)
Applies the logical operation xor between each operand's digits.

max(x, y)
Compares two values numerically and returns the maximum.

max_mag(x, y)
Compares the values numerically with their sign ignored.

min(x, y)
Compares two values numerically and returns the minimum.

min_mag(x, y)
Compares the values numerically with their sign ignored.

minus(x)
Minus corresponds to the unary prefix minus operator in Python.

multiply(x, y)
Return the product of x and y.

next_minus(x)
Returns the largest representable number smaller than x.

next_plus(x)
Returns the smallest representable number larger than x.

next_toward(x, y)
Returns the number closest to x, in direction towards y.

normalize(x)
Reduces x to its simplest form.

number_class(x)
Returns an indication of the class of x.

plus(x)
Plus corresponds to the unary prefix plus operator in Python. This operation applies the context precision and rounding, so it is not an identity operation.

power(x, y, modulo=None)
Return x to the power of y, reduced modulo modulo if given.

With two arguments, compute x**y. If x is negative then y must be integral. The result will be inexact unless y is integral and the result is finite and can be expressed exactly in ‘precision’ digits. The rounding mode of the context is used. Results are always correctly-rounded in the Python version. Changed in version 3.3: The C module computes power() in terms of the correctly-rounded exp() and ln() functions. The result is well-defined but only “almost always correctly-rounded”.

With three arguments, compute (x**y) % modulo. For the three argument form, the following restrictions on the arguments hold:

* all three arguments must be integral
* y must be nonnegative
* at least one of x or y must be nonzero
* modulo must be nonzero and have at most ‘precision’ digits
The value resulting from `Context.power(x, y, modulo)` is equal to the value that would be obtained by computing \((x**y) \mod \text{modulo}\) with unbounded precision, but is computed more efficiently. The exponent of the result is zero, regardless of the exponents of \(x\), \(y\) and \(\text{modulo}\). The result is always exact.

**quantize(x, y)**

Returns a value equal to \(x\) (rounded), having the exponent of \(y\).

**radix()**

Just returns 10, as this is Decimal, :)  

**remainder(x, y)**

Returns the remainder from integer division.  

The sign of the result, if non-zero, is the same as that of the original dividend.

**remainder_near(x, y)**

Returns \(x - y \times n\), where \(n\) is the integer nearest the exact value of \(x / y\) (if the result is 0 then its sign will be the sign of \(x\)).

**rotate(x, y)**

Returns a rotated copy of \(x\), \(y\) times.

**same_quantum(x, y)**

Returns True if the two operands have the same exponent.

**scaleb(x, y)**

Returns the first operand after adding the second value its exp.

**shift(x, y)**

Returns a shifted copy of \(x\), \(y\) times.

**sqrt(x)**

Square root of a non-negative number to context precision.

**subtract(x, y)**

Return the difference between \(x\) and \(y\).

**to_eng_string(x)**

Converts a number to a string, using scientific notation.

**to_integral_exact(x)**

Rounds to an integer.

**to_sci_string(x)**

Converts a number to a string using scientific notation.

### 9.4.4 Constants

The constants in this section are only relevant for the C module. They are also included in the pure Python version for compatibility.

<table>
<thead>
<tr>
<th></th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal.MAX_PREC</td>
<td>4250000000</td>
<td>999999999999999999</td>
</tr>
<tr>
<td>decimal.MAX_EMAX</td>
<td>4250000000</td>
<td>999999999999999999</td>
</tr>
<tr>
<td>decimal.MIN_EMIN</td>
<td>-4250000000</td>
<td>-999999999999999999</td>
</tr>
<tr>
<td>decimal.MIN_ETINY</td>
<td>-849999999</td>
<td>-1999999999999999997</td>
</tr>
</tbody>
</table>
decimal.HAVE_THREADS

The default value is True. If Python is compiled without threads, the C version automatically disables the expensive thread local context machinery. In this case, the value is False.

9.4.5 Rounding modes

decimal.ROUND_CEILING
Round towards Infinity.
decimal.ROUND_DOWN
Round towards zero.
decimal.ROUND_FLOOR
Round towards -Infinity.
decimal.ROUND_HALF_DOWN
Round to nearest with ties going towards zero.
decimal.ROUND_HALF_EVEN
Round to nearest with ties going to nearest even integer.
decimal.ROUND_HALF_UP
Round to nearest with ties going away from zero.
decimal.ROUND_UP
Round away from zero.
decimal.ROUND_05UP
Round away from zero if last digit after rounding towards zero would have been 0 or 5; otherwise round towards zero.

9.4.6 Signals

Signals represent conditions that arise during computation. Each corresponds to one context flag and one context trap enabler.

The context flag is set whenever the condition is encountered. After the computation, flags may be checked for informational purposes (for instance, to determine whether a computation was exact). After checking the flags, be sure to clear all flags before starting the next computation.

If the context’s trap enabler is set for the signal, then the condition causes a Python exception to be raised. For example, if the DivisionByZero trap is set, then a DivisionByZero exception is raised upon encountering the condition.

class decimal.Clamped
   Altered an exponent to fit representation constraints.
   Typically, clamping occurs when an exponent falls outside the context’s Emin and Emax limits. If possible, the exponent is reduced to fit by adding zeros to the coefficient.

class decimal.DecimalException
   Base class for other signals and a subclass of ArithmeticError.

class decimal.DivisionByZero
   Signals the division of a non-infinite number by zero.
   Can occur with division, modulo division, or when raising a number to a negative power. If this signal is not trapped, returns Infinity or -Infinity with the sign determined by the inputs to the calculation.

class decimal.Inexact
   Indicates that rounding occurred and the result is not exact.
   Signals when non-zero digits were discarded during rounding. The rounded result is returned. The signal flag or trap is used to detect when results are inexact.
class `decimal.InvalidOperation`

An invalid operation was performed.

Indicates that an operation was requested that does not make sense. If not trapped, returns NaN. Possible causes include:

- `Infinity - Infinity`
- `0 * Infinity`
- `Infinity / Infinity`
- `x % 0`
- `Infinity % x`
- `sqrt(-x) and x > 0`
- `0 ** 0`
- `x ** (non-integer)`
- `x ** Infinity`

**class `decimal.Overflow`**

Numerical overflow.

Indicates the exponent is larger than Emax after rounding has occurred. If not trapped, the result depends on the rounding mode, either pulling inward to the largest representable finite number or rounding outward to Infinity. In either case, `Inexact` and `Rounded` are also signaled.

**class `decimal.Rounded`**

Rounding occurred though possibly no information was lost.

Signaled whenever rounding discards digits; even if those digits are zero (such as rounding 5.00 to 5.0). If not trapped, returns the result unchanged. This signal is used to detect loss of significant digits.

**class `decimal.Subnormal`**

Exponent was lower than Emin prior to rounding.

Occurs when an operation result is subnormal (the exponent is too small). If not trapped, returns the result unchanged.

**class `decimal.Underflow`**

Numerical underflow with result rounded to zero.

Occurs when a subnormal result is pushed to zero by rounding. `Inexact` and `Subnormal` are also signaled.

**class `decimal.FloatOperation`**

Enable stricter semantics for mixing floats and Decimals.

If the signal is not trapped (default), mixing floats and Decimals is permitted in the `Decimal` constructor, `create_decimal()` and all comparison operators. Both conversion and comparisons are exact. Any occurrence of a mixed operation is silently recorded by setting `FloatOperation` in the context flags. Explicit conversions with `from_float()` or `create_decimal_from_float()` do not set the flag.

Otherwise (the signal is trapped), only equality comparisons and explicit conversions are silent. All other mixed operations raise `FloatOperation`.

The following table summarizes the hierarchy of signals:

<table>
<thead>
<tr>
<th>exceptions.ArithmeticError (exceptions.Exception)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DecimalException</td>
</tr>
<tr>
<td>Clamped</td>
</tr>
<tr>
<td>DivisionByZero (DecimalException, exceptions.ZeroDivisionError)</td>
</tr>
<tr>
<td>Inexact</td>
</tr>
<tr>
<td>Overflow (Inexact, Rounded)</td>
</tr>
<tr>
<td>Underflow (Inexact, Rounded, Subnormal)</td>
</tr>
<tr>
<td>InvalidOperation</td>
</tr>
<tr>
<td>Rounded</td>
</tr>
<tr>
<td>Subnormal</td>
</tr>
<tr>
<td>FloatOperation (DecimalException, exceptions.TypeError)</td>
</tr>
</tbody>
</table>
9.4.7 Floating Point Notes

Mitigating round-off error with increased precision

The use of decimal floating point eliminates decimal representation error (making it possible to represent \(0.1\) exactly); however, some operations can still incur round-off error when non-zero digits exceed the fixed precision.

The effects of round-off error can be amplified by the addition or subtraction of nearly offsetting quantities resulting in loss of significance. Knuth provides two instructive examples where rounded floating point arithmetic with insufficient precision causes the breakdown of the associative and distributive properties of addition:

```
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('10')
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.00600000')
>>> u * (v+w)
Decimal('0.00600000')
```

The `decimal` module makes it possible to restore the identities by expanding the precision sufficiently to avoid loss of significance:

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.00600000')
>>> u * (v+w)
Decimal('0.00600000')
```

Special values

The number system for the `decimal` module provides special values including NaN, sNaN, -Infinity, Infinity, and two zeros, +0 and -0.

Infinities can be constructed directly with `Decimal('Infinity')`. Also, they can arise from dividing by zero when the `DivisionByZero` signal is not trapped. Likewise, when the `Overflow` signal is not trapped, infinity can result from rounding beyond the limits of the largest representable number.

The infinities are signed (affine) and can be used in arithmetic operations where they get treated as very large, indeterminate numbers. For instance, adding a constant to infinity gives another infinite result.

Some operations are indeterminate and return NaN, or if the `InvalidOperation` signal is trapped, raise an exception. For example, \(0/0\) returns NaN which means “not a number”. This variety of NaN is quiet and, once created, will flow through other computations always resulting in another NaN. This behavior can be useful for a series of computations that occasionally have missing inputs — it allows the calculation to proceed while flagging specific results as invalid.
A variant is sNaN which signals rather than remaining quiet after every operation. This is a useful return value when an invalid result needs to interrupt a calculation for special handling.

The behavior of Python’s comparison operators can be a little surprising where a NaN is involved. A test for equality where one of the operands is a quiet or signaling NaN always returns False (even when doing Decimal(‘NaN’)==Decimal(‘NaN’)), while a test for inequality always returns True. An attempt to compare two Decimals using any of the <, <=, > or >= operators will raise the InvalidOperation signal if either operand is a NaN, and return False if this signal is not trapped. Note that the General Decimal Arithmetic specification does not specify the behavior of direct comparisons; these rules for comparisons involving a NaN were taken from the IEEE 854 standard (see Table 3 in section 5.7). To ensure strict standards-compliance, use the compare() and compare-signal() methods instead.

The signed zeros can result from calculations that underflow. They keep the sign that would have resulted if the calculation had been carried out to greater precision. Since their magnitude is zero, both positive and negative zeros are treated as equal and their sign is informational.

In addition to the two signed zeros which are distinct yet equal, there are various representations of zero with differing precisions yet equivalent in value. This takes a bit of getting used to. For an eye accustomed to normalized floating point representations, it is not immediately obvious that the following calculation returns a value equal to zero:

```python
>>> 1 / Decimal('Infinity')
Decimal('0E-1000026')
```

### 9.4.8 Working with threads

The `getcontext()` function accesses a different `Context` object for each thread. Having separate thread contexts means that threads may make changes (such as `getcontext().prec=10`) without interfering with other threads.

Likewise, the `setcontext()` function automatically assigns its target to the current thread.

If `setcontext()` has not been called before `getcontext()`, then `getcontext()` will automatically create a new context for use in the current thread.

The new context is copied from a prototype context called `DefaultContext`. To control the defaults so that each thread will use the same values throughout the application, directly modify the `DefaultContext` object. This should be done before any threads are started so that there won’t be a race condition between threads calling `getcontext()`. For example:

```python
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
```

```python
t1.start()
t2.start()
t3.start()

. . .
```

### 9.4.9 Recipes

Here are a few recipes that serve as utility functions and that demonstrate ways to work with the `Decimal` class:

```python
def moneyfmt(value, places=2, curr='', sep='', dp='.', pos='', neg='-', trailneg=''):  
    """Convert Decimal to a money formatted string."""
```
places: required number of places after the decimal point
curr: optional currency symbol before the sign (may be blank)
sep: optional grouping separator (comma, period, space, or blank)
dp: decimal point indicator (comma or period)
  only specify as blank when places is zero
pos: optional sign for positive numbers: ‘+’, space or blank
neg: optional sign for negative numbers: ‘−’, ‘(’, space or blank
trailneg: optional trailing minus indicator: ‘−’, ‘)’, space or blank

```python
>>> d = Decimal(‘-1234567.8901’)
>>> moneyfmt(d, curr=’$’)
’-$1,234,567.89’
>>> moneyfmt(d, places=0, sep=‘.’, dp=‘’, neg=‘’, trailneg=‘−’)
’1.234.568−’
>>> moneyfmt(d, curr=’$’, neg=‘(’, trailneg=‘)’)
’($1,234,567.89)’
>>> moneyfmt(Decimal(123456789), sep=’ ’)
’123 456 789.00’
>>> moneyfmt(Decimal(‘−0.02’), neg=‘<’, trailneg=’>’)
’<0.02>’
```

```python
q = Decimal(10) ** -places  # 2 places --> ’0.01’
sign, digits, exp = value.quantize(q).as_tuple()
result = []
digits = list(map(str, digits))
build, next = result.append, digits.pop
if sign:
    build(trailneg)
for i in range(places):
    build(next() if digits else ‘0’)
if places:
    build(dp)
if not digits:
    build(‘0’)
i = 0
while digits:
    build(next())
i += 1
    if i == 3 and digits:
        i = 0
        build(sep)
build(curr)
build(neg if sign else pos)
return ‘’.join(reversed(result))
```

```python
def pi():
    """Compute Pi to the current precision."
    >>> print(pi())
    3.141592653589793238462643383
```

```python
getcontext().prec += 2  # extra digits for intermediate steps
three = Decimal(3)  # substitute "three=3.0" for regular floats
lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
while s != lasts:
    lasts = s
```
n, na = n+na, na+8
d, da = d+da, da+32
t = (t * n) / d
s += t
getcontext().prec -= 2
return +s  # unary plus applies the new precision

def exp(x):
    """Return e raised to the power of x. Result type matches input type."

    >>> print(exp(Decimal('0.5')))
    0.8775825618903727161162815826
    >>> print(exp(0.5))
    0.87758256189
    >>> print(exp(0.5+0j))
    (0.87758256189+0j)
    """
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 1
        fact *= i
        num *= x
        s += num / fact
    getcontext().prec -= 2
    return +s

def cos(x):
    """Return the cosine of x as measured in radians."
    The Taylor series approximation works best for a small value of x. For larger values, first compute x = x % (2 * pi).

    >>> print(cos(Decimal('0.5')))
    0.8775825618903727161162815826
    >>> print(cos(0.5))
    0.87758256189
    >>> print(cos(0.5+0j))
    (0.87758256189+0j)
    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

def sin(x):
Return the sine of x as measured in radians.

The Taylor series approximation works best for a small value of x. For larger values, first compute \( x = x \% (2 \times \pi) \).

```python
>>> print(sin(Decimal('0.5'))) 0.4794255386042030002732879352
>>> print(sin(0.5)) 0.479425538604
>>> print(sin(0.5+0j)) (0.479425538604+0j)
```

```python
getcontext().prec += 2
i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
while s != lasts:
    lasts = s
    i += 2
    fact *= i * (i-1)
    num *= x * x
    sign *= -1
    s += num / fact * sign
getcontext().prec -= 2
return +s
```

9.4.10 Decimal FAQ

Q. It is cumbersome to type `decimal.Decimal('1234.5')`. Is there a way to minimize typing when using the interactive interpreter?

A. Some users abbreviate the constructor to just a single letter:

```python
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

Q. In a fixed-point application with two decimal places, some inputs have many places and need to be rounded. Others are not supposed to have excess digits and need to be validated. What methods should be used?

A. The `quantize()` method rounds to a fixed number of decimal places. If the `Inexact` trap is set, it is also useful for validation:

```python
>>> TWOPLACES = Decimal(10) ** -2  # same as Decimal('0.01')
>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')
>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')
>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))  # Traceback (most recent call last):
...  
Inexact: None
```

Q. Once I have valid two place inputs, how do I maintain that invariant throughout an application?

A. Some operations like addition, subtraction, and multiplication by an integer will automatically preserve fixed point. Others operations, like division and non-integer multiplication, will change the number of decimal places and need to be followed-up with a `quantize()` step:
In developing fixed-point applications, it is convenient to define functions to handle the `quantize()` step:

```python
def mul(x, y, fp=TWOPLACES):
    return (x * y).quantize(fp)
def div(x, y, fp=TWOPLACES):
    return (x / y).quantize(fp)
```

```python
>>> mul(a, b)
Decimal('325.62')
```

```python
>>> div(b, a)
Decimal('0.03')
```

Q. There are many ways to express the same value. The numbers 200, 200.000, 2E2, and 02E+4 all have the same value at various precisions. Is there a way to transform them to a single recognizable canonical value?

A. The `normalize()` method maps all equivalent values to a single representative:

```python
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

Q. Some decimal values always print with exponential notation. Is there a way to get a non-exponential representation?

A. For some values, exponential notation is the only way to express the number of significant places in the coefficient. For example, expressing 5.0E+3 as 5000 keeps the value constant but cannot show the original’s two-place significance.

If an application does not care about tracking significance, it is easy to remove the exponent and trailing zeroes, losing significance, but keeping the value unchanged:

```python
def remove_exponent(d):
    return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()
```

```python
>>> remove_exponent(Decimal('5E+3'))
Decimal('5000')
```

Q. Is there a way to convert a regular float to a `Decimal`?

A. Yes, any binary floating point number can be exactly expressed as a `Decimal` though an exact conversion may take more precision than intuition would suggest:

```python
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

Q. Within a complex calculation, how can I make sure that I haven’t gotten a spurious result because of insufficient precision or rounding anomalies?

A. The decimal module makes it easy to test results. A best practice is to re-run calculations using greater precision and with various rounding modes. Widely differing results indicate insufficient precision, rounding mode issues, ill-conditioned inputs, or a numerically unstable algorithm.

Q. I noticed that context precision is applied to the results of operations but not to the inputs. Is there anything to watch out for when mixing values of different precisions?
A. Yes. The principle is that all values are considered to be exact and so is the arithmetic on those values. Only the results are rounded. The advantage for inputs is that “what you type is what you get”. A disadvantage is that the results can look odd if you forget that the inputs haven’t been rounded:

```python
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
```

```python
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

The solution is either to increase precision or to force rounding of inputs using the unary plus operation:

```python
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')  # unary plus triggers rounding
Decimal('1.23')
```

Alternatively, inputs can be rounded upon creation using the `Context.create_decimal()` method:

```python
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

## 9.5 fractions — Rational numbers

**Source code:** Lib/fractions.py

The `fractions` module provides support for rational number arithmetic.

A Fraction instance can be constructed from a pair of integers, from another rational number, or from a string.

```python
class fractions.Fraction(numerator=0, denominator=1)
class fractions.Fraction(other_fraction)
class fractions.Fraction(float)
class fractions.Fraction(decimal)
class fractions.Fraction(string)
```

The first version requires that `numerator` and `denominator` are instances of `numbers.Rational` and returns a new `Fraction` instance with value `numerator/denominator`. If `denominator` is 0, it raises a `ZeroDivisionError`. The second version requires that `other_fraction` is an instance of `numbers.Rational` and returns a `Fraction` instance with the same value. The next two versions accept either a `float` or a `decimal.Decimal` instance, and return a `Fraction` instance with exactly the same value. Note that due to the usual issues with binary floating-point (see tut-fp-issues), the argument to `Fraction(1.1)` is not exactly equal to 11/10, and so `Fraction(1.1)` does not return `Fraction(11, 10)` as one might expect. (But see the documentation for the `limit_denominator()` method below.) The last version of the constructor expects a string or unicode instance. The usual form for this instance is:

```
[sign] numerator ["/" denominator]
```

where the optional `sign` may be either ‘+’ or ‘-’ and `numerator` and `denominator` (if present) are strings of decimal digits. In addition, any string that represents a finite value and is accepted by the `float` constructor is also accepted by the `Fraction` constructor. In either form the input string may also have leading and/or trailing whitespace. Here are some examples:

```python
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
```
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction('-3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \
1000000')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)

The `Fraction` class inherits from the abstract base class `numbers.Rational`, and implements all of the methods and operations from that class. `Fraction` instances are hashable, and should be treated as immutable. In addition, `Fraction` has the following properties and methods:

**Changed in version 3.2:** The `Fraction` constructor now accepts `float` and `decimal.Decimal` instances.

### `numerator`
Numerator of the Fraction in lowest term.

### `denominator`
Denominator of the Fraction in lowest term.

#### `from_float` (`flt`)
This class method constructs a `Fraction` representing the exact value of `flt`, which must be a `float`. Beware that `Fraction.from_float(0.3)` is not the same value as `Fraction(3, 10)`.

**Note:** From Python 3.2 onwards, you can also construct a `Fraction` instance directly from a `float`.

#### `from_decimal` (`dec`)
This class method constructs a `Fraction` representing the exact value of `dec`, which must be a `decimal.Decimal` instance.

**Note:** From Python 3.2 onwards, you can also construct a `Fraction` instance directly from a `decimal.Decimal` instance.

#### `limit_denominator` (`max_denominator=1000000`)  
Finds and returns the closest `Fraction` to `self` that has denominator at most `max_denominator`. This method is useful for finding rational approximations to a given floating-point number:

```python
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

or for recovering a rational number that’s represented as a float:

```python
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
Fraction(4503599627370497, 9007199254740992)
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
```
Fraction(1.1).limit_denominator()
Fraction(11, 10)

__floor__()
Returns the greatest \( \text{int} \leq \text{self} \). This method can also be accessed through the \texttt{math.floor()} function:

```python
>>> from math import floor
>>> floor(Fraction(355, 113))
3
```

__ceil__()
Returns the least \( \text{int} \geq \text{self} \). This method can also be accessed through the \texttt{math.ceil()} function.

__round__()
__round__ (\texttt{ndigits})
The first version returns the nearest \text{int} to \text{self}, rounding half to even. The second version rounds \text{self} to the nearest multiple of \texttt{Fraction(1, 10**\texttt{ndigits})} (logically, if \texttt{ndigits} is negative), again rounding half toward even. This method can also be accessed through the \texttt{round()} function.

fractions.gcd(a, b)
Return the greatest common divisor of the integers \( a \) and \( b \). If either \( a \) or \( b \) is nonzero, then the absolute value of \( \text{gcd}(a, b) \) is the largest integer that divides both \( a \) and \( b \). \( \text{gcd}(a, b) \) has the same sign as \( b \) if \( b \) is nonzero; otherwise it takes the sign of \( a \). \( \text{gcd}(0, 0) \) returns 0.

See Also:

Module \texttt{numbers} The abstract base classes making up the numeric tower.

## 9.6 random — Generate pseudo-random numbers

Source code: Lib/random.py

This module implements pseudo-random number generators for various distributions.

For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.

On the real line, there are functions to compute uniform, normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. For generating distributions of angles, the von Mises distribution is available.

Almost all module functions depend on the basic function \texttt{random()}, which generates a random float uniformly in the semi-open range \([0.0, 1.0)\). Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of \(2^{19937}-1\). The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

The functions supplied by this module are actually bound methods of a hidden instance of the \texttt{random.Random} class. You can instantiate your own instances of \texttt{Random} to get generators that don’t share state.

Class \texttt{Random} can also be subclassed if you want to use a different basic generator of your own devising: in that case, override the \texttt{random()}, \texttt{seed()}, \texttt{getstate()}, and \texttt{setstate()} methods. Optionally, a new generator can supply a \texttt{getrandbits()} method — this allows \texttt{randrange()} to produce selections over an arbitrarily large range.
The `random` module also provides the `SystemRandom` class which uses the system function `os.urandom()` to generate random numbers from sources provided by the operating system.

**Warning:** The pseudo-random generators of this module should not be used for security purposes. Use `os.urandom()` or `SystemRandom` if you require a cryptographically secure pseudo-random number generator.

### Bookkeeping functions:

- **`random.seed(a=None, version=2)`**
  - Initialize the random number generator.
  - If `a` is omitted or `None`, the current system time is used. If randomness sources are provided by the operating system, they are used instead of the system time (see the `os.urandom()` function for details on availability).
  - If `a` is an int, it is used directly.
  - With version 2 (the default), a `str`, `bytes`, or `bytearray` object gets converted to an `int` and all of its bits are used. With version 1, the `hash()` of `a` is used instead. Changed in version 3.2: Moved to the version 2 scheme which uses all of the bits in a string seed.

- **`random.getstate()`**
  - Return an object capturing the current internal state of the generator. This object can be passed to `setstate()` to restore the state.

- **`random.setstate(state)`**
  - `state` should have been obtained from a previous call to `getstate()`, and `setstate()` restores the internal state of the generator to what it was at the time `getstate()` was called.

- **`random.getrandbits(k)`**
  - Returns a Python integer with `k` random bits. This method is supplied with the MersenneTwister generator and some other generators may also provide it as an optional part of the API. When available, `getrandbits()` enables `randrange()` to handle arbitrarily large ranges.

### Functions for integers:

- **`random.randrange(stop)`**
- **`random.randrange(start, stop[, step])`**
  - Return a randomly selected element from `range(start, stop, step)`. This is equivalent to `choice(range(start, stop, step))`, but doesn’t actually build a range object.
  - The positional argument pattern matches that of `range()`. Keyword arguments should not be used because the function may use them in unexpected ways. Changed in version 3.2: `randrange()` is more sophisticated about producing equally distributed values. Formerly it used a style like `int(random() * n)` which could produce slightly uneven distributions.

- **`random.randint(a, b)`**
  - Return a random integer `N` such that `a <= N <= b`. Alias for `randrange(a, b+1)`.

### Functions for sequences:

- **`random.choice(seq)`**
  - Return a random element from the non-empty sequence `seq`. If `seq` is empty, raises `IndexError`.

- **`random.shuffle(x[, random])`**
  - Shuffle the sequence `x` in place. The optional argument `random` is a 0-argument function returning a random float in [0.0, 1.0); by default, this is the function `random()`.
  - Note that for even rather small `len(x)`, the total number of permutations of `x` is larger than the period of most random number generators; this implies that most permutations of a long sequence can never be generated.

- **`random.sample(population, k)`**
  - Return a `k` length list of unique elements chosen from the population sequence or set. Used for random sampling without replacement.
Returns a new list containing elements from the population while leaving the original population unchanged. The resulting list is in selection order so that all sub-slices will also be valid random samples. This allows raffle winners (the sample) to be partitioned into grand prize and second place winners (the subslices).

Members of the population need not be `hashable` or unique. If the population contains repeats, then each occurrence is a possible selection in the sample.

To choose a sample from a range of integers, use an `range()` object as an argument. This is especially fast and space efficient for sampling from a large population: `sample(range(10000000), 60)`.

If the sample size is larger than the population size, a `ValueError` is raised.

The following functions generate specific real-valued distributions. Function parameters are named after the corresponding variables in the distribution’s equation, as used in common mathematical practice; most of these equations can be found in any statistics text.

- `random.random()`
  Return the next random floating point number in the range [0.0, 1.0).

- `random.uniform(a, b)`
  Return a random floating point number \( N \) such that \( a \leq N \leq b \) for \( a \leq b \) and \( b \leq N \leq a \) for \( b < a \).

  The end-point value \( b \) may or may not be included in the range depending on floating-point rounding in the equation \( a + (b-a) \times \text{random}() \).

- `random.triangular(low, high, mode)`
  Return a random floating point number \( N \) such that \( low \leq N \leq high \) and with the specified `mode` between those bounds. The `low` and `high` bounds default to zero and one. The `mode` argument defaults to the midpoint between the bounds, giving a symmetric distribution.

- `random.betavariate(alpha, beta)`
  Beta distribution. Conditions on the parameters are \( \alpha > 0 \) and \( \beta > 0 \). Returned values range between 0 and 1.

- `random.expoavariate(lambd)`
  Exponential distribution. \( \lambda \) is 1.0 divided by the desired mean. It should be nonzero. (The parameter would be called “lambda”, but that is a reserved word in Python.) Returned values range from 0 to positive infinity if \( \lambda \) is positive, and from negative infinity to 0 if \( \lambda \) is negative.

- `random.gammavariate(alpha, beta)`
  Gamma distribution. (Not the gamma function!) Conditions on the parameters are \( \alpha > 0 \) and \( \beta > 0 \).

  The probability distribution function is:

  \[
  \text{pdf}(x) = \frac{x^{(\alpha - 1)} \times \exp(-x/\beta)}{\Gamma(\alpha) \times \beta^{\alpha}}
  \]

  where \( \Gamma(\alpha) \) is the gamma function.

- `random.gauss(mu, sigma)`
  Gaussian distribution. \( \mu \) is the mean, and \( \sigma \) is the standard deviation. This is slightly faster than the `normalvariate()` function defined below.

- `random.lognornormavariate(mu, sigma)`
  Log normal distribution. If you take the natural logarithm of this distribution, you’ll get a normal distribution with mean \( \mu \) and standard deviation \( \sigma \). \( \mu \) can have any value, and \( \sigma \) must be greater than zero.

- `random.normalvariate(mu, sigma)`
  Normal distribution. \( \mu \) is the mean, and \( \sigma \) is the standard deviation.

- `random.vonmisesvariate(mu, kappa)`
  \( \mu \) is the mean angle, expressed in radians between 0 and \( 2\pi \), and \( \kappa \) is the concentration parameter, which must be greater than or equal to zero. If \( \kappa \) is equal to zero, this distribution reduces to a uniform random angle over the range 0 to \( 2\pi \).
**random.paretovariate**(alpha)

Pareto distribution. alpha is the shape parameter.

**random.weibullvariate**(alpha, beta)

Weibull distribution. alpha is the scale parameter and beta is the shape parameter.

Alternative Generator:

```python
class random.SystemRandom([seed])
```

Class that uses the os.urandom() function for generating random numbers from sources provided by the operating system. Not available on all systems. Does not rely on software state, and sequences are not reproducible. Accordingly, the seed() method has no effect and is ignored. The getstate() and setstate() methods raise NotImplementedError if called.

**See Also:**


Complementary-Multiply-with-Carry recipe for a compatible alternative random number generator with a long period and comparatively simple update operations.

### 9.6.1 Notes on Reproducibility

Sometimes it is useful to be able to reproduce the sequences given by a pseudo random number generator. By re-using a seed value, the same sequence should be reproducible from run to run as long as multiple threads are not running.

Most of the random module’s algorithms and seeding functions are subject to change across Python versions, but two aspects are guaranteed not to change:

- If a new seeding method is added, then a backward compatible seeder will be offered.
- The generator’s random() method will continue to produce the same sequence when the compatible seeder is given the same seed.

### 9.6.2 Examples and Recipes

Basic usage:

```python
>>> random.random() # Random float x, 0.0 <= x < 1.0
0.37444887175646646

>>> random.uniform(1, 10) # Random float x, 1.0 <= x < 10.0
1.1800146073117523

>>> random.randrange(10) # Integer from 0 to 9
7

>>> random.randrange(0, 101, 2) # Even integer from 0 to 100
26

>>> random.choice('abcdefghij') # Single random element
'c'

>>> items = [1, 2, 3, 4, 5, 6, 7]
>>> random.shuffle(items)
>>> items
[7, 3, 2, 5, 6, 4, 1]
```
A common task is to make a `random.choice()` with weighted probabilities. If the weights are small integer ratios, a simple technique is to build a sample population with repeats:

```python
>>> weighted_choices = [('Red', 3), ('Blue', 2), ('Yellow', 1), ('Green', 4)]
>>> population = [val for val, cnt in weighted_choices for i in range(cnt)]
>>> random.choice(population)
'Green'
```

A more general approach is to arrange the weights in a cumulative distribution with `itertools.accumulate()`, and then locate the random value with `bisect.bisect()`:

```python
>>> choices, weights = zip(*weighted_choices)
>>> cumdist = list(itertools.accumulate(weights))
>>> x = random.random() * cumdist[-1]
>>> choices[bisect.bisect(cumdist, x)]
'Blue'
```
FUNCTIONAL PROGRAMMING
MODULES

The modules described in this chapter provide functions and classes that support a functional programming style, and general operations on callables.

The following modules are documented in this chapter:

10.1 *itertools* — Functions creating iterators for efficient looping

This module implements a number of *iterator* building blocks inspired by constructs from APL, Haskell, and SML. Each has been recast in a form suitable for Python.

The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an “iterator algebra” making it possible to construct specialized tools succinctly and efficiently in pure Python.

For instance, SML provides a tabulation tool: `tabulate(f)` which produces a sequence `f(0), f(1), ...` The same effect can be achieved in Python by combining `map()` and `count()` to form `map(f, count())`.

These tools and their built-in counterparts also work well with the high-speed functions in the `operator` module. For example, the multiplication operator can be mapped across two vectors to form an efficient dot-product: `sum(map(operator.mul, vector1, vector2))`.

### Infinite Iterators:

<table>
<thead>
<tr>
<th>Iterator</th>
<th>Arguments</th>
<th>Results</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count()</code></td>
<td><code>start, [step] p</code></td>
<td><code>start, start+step, start+2*step, ...</code></td>
<td><code>count(10) --&gt; 10 11 12 13 14 ...</code></td>
</tr>
<tr>
<td><code>cycle()</code></td>
<td><code>elem[,n]</code></td>
<td><code>elem, elem, elem, ... endlessly or up to n times</code></td>
<td><code>cycle('ABCD') --&gt; A B C D A B C D ...</code></td>
</tr>
<tr>
<td><code>repeat()</code></td>
<td></td>
<td></td>
<td><code>repeat(10, 3) --&gt; 10 10 10</code></td>
</tr>
</tbody>
</table>

Iterators terminating on the shortest input sequence:
## Itertool functions

The following module functions all construct and return iterators. Some provide streams of infinite length, so they should only be accessed by functions or loops that truncate the stream.

**itertools.accumulate(iterable[, func])**

Make an iterator that returns accumulated sums. Elements may be any addable type including `Decimal` or `Fraction`. If the optional `func` argument is supplied, it should be a function of two arguments and it will be used instead of addition.

Equivalent to:

```python
def accumulate(iterable, func=operator.add):
    'Return running totals'
```

### Combinatoric generators:

<table>
<thead>
<tr>
<th>Iterator</th>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>product()</strong></td>
<td>p, q, ... [repeat=1]</td>
<td>cartesian product, equivalent to a nested for-loop</td>
</tr>
<tr>
<td><strong>permutations()</strong></td>
<td>p, r</td>
<td>r-length tuples, all possible orderings, no repeated elements</td>
</tr>
<tr>
<td><strong>combinations()</strong></td>
<td>p, r</td>
<td>r-length tuples, in sorted order, no repeated elements</td>
</tr>
<tr>
<td><strong>combinations_with_replacement()</strong></td>
<td>p, r</td>
<td>r-length tuples, in sorted order, with repeated elements</td>
</tr>
<tr>
<td><strong>product('ABCD', repeat=2)</strong></td>
<td>AA AB AC AD AB BB BC BD CA CB CC CD DA DB DC DD</td>
<td></td>
</tr>
<tr>
<td><strong>permutations('ABCD', 2)</strong></td>
<td>AB AC AD BA BC BD CA CB CD DA DB DC</td>
<td></td>
</tr>
<tr>
<td><strong>combinations('ABCD', 2)</strong></td>
<td>AB AC AD BC BD</td>
<td></td>
</tr>
<tr>
<td><strong>combinations_with_replacement('ABCD', 2)</strong></td>
<td>AA AB AC AD BB BC BD CC CD DD</td>
<td></td>
</tr>
</tbody>
</table>

## 10.1.1 Itertool functions

The following module functions all construct and return iterators. Some provide streams of infinite length, so they should only be accessed by functions or loops that truncate the stream.

**itertools.accumulate(iterable[, func])**

Make an iterator that returns accumulated sums. Elements may be any addable type including `Decimal` or `Fraction`. If the optional `func` argument is supplied, it should be a function of two arguments and it will be used instead of addition.

Equivalent to:

```python
def accumulate(iterable, func=operator.add):
    'Return running totals'
```
The Python Library Reference, Release 3.3.3

# accumulate([1,2,3,4,5]) --> 1 3 6 10 15
# accumulate([1,2,3,4,5], operator.mul) --> 1 2 6 24 120
it = iter(iterable)
total = next(it)
yield total
for element in it:
    total = func(total, element)
yield total

There are a number of uses for the func argument. It can be set to min() for a running minimum, max() for a running maximum, or operator.mul() for a running product. Amortization tables can be built by accumulating interest and applying payments. First-order recurrence relations can be modeled by supplying the initial value in the iterable and using only the accumulated total in func argument:

```python
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, operator.mul))  # running product
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]
>>> list(accumulate(data, max))  # running maximum
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]
# Amortize a 5% loan of 1000 with 4 annual payments of 90
>>> cashflows = [1000, -90, -90, -90, -90]
>>> list(accumulate(cashflows, lambda bal, pmt: bal*1.05 + pmt))
[1000, 960.0, 918.0, 873.9000000000001, 827.5950000000001]
# Chaotic recurrence relation http://en.wikipedia.org/wiki/Logistic_map
>>> logistic_map = lambda x, _: r * x * (1 - x)
>>> r = 3.8
>>> x0 = 0.4
>>> [format(x, '.2f') for x in accumulate(inputs, logistic_map)]
['0.40', '0.91', '0.30', '0.81', '0.60', '0.92', '0.29', '0.79', '0.63',
 '0.88', '0.39', '0.90', '0.33', '0.84', '0.52', '0.95', '0.18', '0.57',
 '0.93', '0.25', '0.71', '0.79', '0.63', '0.88', '0.39', '0.91', '0.32',
 '0.83', '0.54', '0.95', '0.20', '0.60', '0.91', '0.30', '0.80', '0.60']
```

See functools.reduce() for a similar function that returns only the final accumulated value. New in version 3.2. Changed in version 3.3: Added the optional func parameter.

**itertools.chain(*iterables)**

Make an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. Used for treating consecutive sequences as a single sequence. Equivalent to:

```python
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

**classmethod chain.from_iterable(iterable)**

Alternate constructor for chain(). Gets chained inputs from a single iterable argument that is evaluated lazily. Roughly equivalent to:

```python
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

10.1. itertools — Functions creating iterators for efficient looping
itertools.combinations(iterable, r)
    Return r length subsequences of elements from the input iterable.

    Combinations are emitted in lexicographic sort order. So, if the input iterable is sorted, the combination tuples will be produced in sorted order.

    Elements are treated as unique based on their position, not on their value. So if the input elements are unique, there will be no repeat values in each combination.

    Equivalent to:

def combinations(iterable, r):
    # combinations(‘ABCD’, 2) --> AB AC AD BC BD CD
    # combinations(range(4), 3) --> 012 013 023 123
    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = list(range(r))
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)

    The code for combinations() can be also expressed as a subsequence of permutations() after filtering entries where the elements are not in sorted order (according to their position in the input pool):

def combinations(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in permutations(range(n), r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)

    The number of items returned is n! / r! / (n-r)! when 0 <= r <= n or zero when r > n.

itertools.combinations_with_replacement(iterable, r)
    Return r length subsequences of elements from the input iterable allowing individual elements to be re-

    Combinations are emitted in lexicographic sort order. So, if the input iterable is sorted, the combination tuples will be produced in sorted order.

    Elements are treated as unique based on their position, not on their value. So if the input elements are unique, the generated combinations will also be unique.

    Equivalent to:

def combinations_with_replacement(iterable, r):
    # combinations_with_replacement(‘ABC’, 2) --> AA AB AC BB BC CC
    pool = tuple(iterable)
    n = len(pool)
    if not n and r:
        return
    indices = [0] * r
The code for `combinations_with_replacement()` can be also expressed as a subsequence of `product()` after filtering entries where the elements are not in sorted order (according to their position in the input pool):

```python
def combinations_with_replacement(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in product(range(n), repeat=r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

The number of items returned is \((n+r-1)! / r! / (n-1)!\) when \(n > 0\). New in version 3.1.

`itertools.compress(data, selectors)`
Make an iterator that filters elements from `data` returning only those that have a corresponding element in `selectors` that evaluates to `True`. Stops when either the `data` or `selectors` iterables has been exhausted. Equivalent to:

```python
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
    return (d for d, s in zip(data, selectors) if s)
```

New in version 3.1.

`itertools.count(start=0, step=1)`
Make an iterator that returns evenly spaced values starting with number `start`. Often used as an argument to `map()` to generate consecutive data points. Also, used with `zip()` to add sequence numbers. Equivalent to:

```python
def count(start=0, step=1):
    # count(10) --> 10 11 12 13 14 ...
    # count(2.5, 0.5) --> 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
```

When counting with floating point numbers, better accuracy can sometimes be achieved by substituting multiplicative code such as: `(start + step * i for i in count())`. Changed in version 3.1: Added `step` argument and allowed non-integer arguments.

`itertools.cycle(iterable)`
Make an iterator returning elements from the iterable and saving a copy of each. When the iterable is exhausted, return elements from the saved copy. Repeats indefinitely. Equivalent to:

```python
def cycle(iterable):
    # cycle('ABCD') --> A B C D A B C D A B C D ...
    saved = []
    for element in iterable:
        yield element
```
saved.append(element)

while saved:
    for element in saved:
        yield element

Note, this member of the toolkit may require significant auxiliary storage (depending on the length of the iterable).

**itertools.dropwhile**(predicate, iterable)

Make an iterator that drops elements from the iterable as long as the predicate is true; afterwards, returns every element. Note, the iterator does not produce any output until the predicate first becomes false, so it may have a lengthy start-up time. Equivalent to:

```python
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
        break
    for x in iterable:
        yield x
```

**itertools.filterfalse**(predicate, iterable)

Make an iterator that filters elements from iterable returning only those for which the predicate is False. If predicate is None, return the items that are false. Equivalent to:

```python
def filterfalse(predicate, iterable):
    # filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
    if predicate is None:
        predicate = bool
    for x in iterable:
        if not predicate(x):
            yield x
```

**itertools.groupby**(iterable, key=None)

Make an iterator that returns consecutive keys and groups from the iterable. The key is a function computing a key value for each element. If not specified or is None, key defaults to an identity function and returns the element unchanged. Generally, the iterable needs to already be sorted on the same key function.

The operation of **groupby**() is similar to the uniq filter in Unix. It generates a break or new group every time the value of the key function changes (which is why it is usually necessary to have sorted the data using the same key function). That behavior differs from SQL's GROUP BY which aggregates common elements regardless of their input order.

The returned group is itself an iterator that shares the underlying iterable with **groupby**(). Because the source is shared, when the **groupby**() object is advanced, the previous group is no longer visible. So, if that data is needed later, it should be stored as a list:

```python
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))  # Store group iterator as a list
    uniquekeys.append(k)
```

**groupby**() is equivalent to:

```python
class groupby:
    # [k for k, g in groupby(‘AAAABBBCCDAABBB’)] --> A B C D A B
```
# [list(g) for k, g in groupby('AAAAABBBCCD')] --> AAAA BBB CC D

def __init__(self, iterable, key=None):
    if key is None:
        key = lambda x: x
    self.keyfunc = key
    self.it = iter(iterable)
    self.tgtkey = self.currkey = self.currvalue = object()

def __iter__(self):
    return self

def __next__(self):
    while self.currkey == self.tgtkey:
        self.currvalue = next(self.it)  # Exit on StopIteration
        self.currkey = self.keyfunc(self.currvalue)
    self.tgtkey = self.currkey
    return (self.currkey, self._grouper(self.tgtkey))

def _grouper(self, tgtkey):
    while self.currkey == tgtkey:
        yield self.currvalue
        self.currvalue = next(self.it)  # Exit on StopIteration
        self.currkey = self.keyfunc(self.currvalue)

itertools.islice(iterable, start, stop, step)

Make an iterator that returns selected elements from the iterable. If start is non-zero, then elements from the iterable are skipped until start is reached. Afterward, elements are returned consecutively unless step is set higher than one which results in items being skipped. If stop is None, then iteration continues until the iterator is exhausted, if at all; otherwise, it stops at the specified position. Unlike regular slicing, islice() does not support negative values for start, stop, or step. Can be used to extract related fields from data where the internal structure has been flattened (for example, a multi-line report may list a name field on every third line). Equivalent to:

def islice(iterable, *args):
    # islice('ABCDEFG', 2) --> A B
    # islice('ABCDEFG', 2, 4) --> C D
    # islice('ABCDEFG', 2, None) --> C D E F G
    # islice('ABCDEFG', 0, None, 2) --> A C E G
    s = slice(*args)
    it = iter(range(s.start or 0, s.stop or sys.maxsize, s.step or 1))
    nexti = next(it)
    for i, element in enumerate(iterable):
        if i == nexti:
            yield element
        nexti = next(it)

    If start is None, then iteration starts at zero. If step is None, then the step defaults to one.

itertools.permutations(iterable, r=None)

Return successive r length permutations of elements in the iterable.

If r is not specified or is None, then r defaults to the length of the iterable and all possible full-length permutations are generated.

Permutations are emitted in lexicographic sort order. So, if the input iterable is sorted, the permutation tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. So if the input elements are unique, there will be no repeat values in each permutation.

Equivalent to:
The code for `permutations()` can be also expressed as a subsequence of `product()`, filtered to exclude entries with repeated elements (those from the same position in the input pool):

```python
def permutations(iterable, r=None):
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return
    indices = list(range(n))
    cycles = list(range(n, n-r, -1))
    yield tuple(pool[i] for i in indices[:r])
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
```

The number of items returned is $n! / (n-r)!$ when $0 \leq r \leq n$ or zero when $r > n$.
result = [x+[y] for x in result for y in pool]
for prod in result:
    yield tuple(prod)

```
import itertools

result = [x+[y] for x in result for y in pool]
for prod in result:
    yield tuple(prod)
```

**itertools.repeat**(object[, times])

Make an iterator that returns object over and over again. Runs indefinitely unless the times argument is specified. Used as argument to map() for invariant parameters to the called function. Also used with zip() to create an invariant part of a tuple record. Equivalent to:

```python
def repeat(object, times=None):
    # repeat(10, 3) --> 10 10 10
    if times is None:
        while True:
            yield object
    else:
        for i in range(times):
            yield object
```

A common use for repeat is to supply a stream of constant values to map or zip:

```python
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

**itertools.starmap**(function, iterable)

Make an iterator that computes the function using arguments obtained from the iterable. Used instead of map() when argument parameters are already grouped in tuples from a single iterable (the data has been “pre-zipped”). The difference between map() and starmap() parallels the distinction between function(a,b) and function(*c). Equivalent to:

```python
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
    for args in iterable:
        yield function(*args)
```

**itertools.takewhile**(predicate, iterable)

Make an iterator that returns elements from the iterable as long as the predicate is true. Equivalent to:

```python
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

**itertools.tee**(iterable, n=2)

Return n independent iterators from a single iterable. Equivalent to:

```python
def tee(iterable, n=2):
    it = iter(iterable)
    deques = [collections.deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:
                # when the local deque is empty
                newval = next(it)
                # fetch a new value and
                for d in deques:
                    # load it to all the deques
                    d.append(newval)
            yield d.popleft()
    for i in range(n):
        yield gen(deques[i])
```

## 10.1. itertools — Functions creating iterators for efficient looping

253
yield mydeque.popleft()
return tuple(gen(d) for d in deques)

Once tee() has made a split, the original iterable should not be used anywhere else; otherwise, the iterable could get advanced without the tee objects being informed.

This itertool may require significant auxiliary storage (depending on how much temporary data needs to be stored). In general, if one iterator uses most or all of the data before another iterator starts, it is faster to use list() instead of tee().

**itertools.zip_longest(*iterables, fillvalue=None)**

Make an iterator that aggregates elements from each of the iterables. If the iterables are of uneven length, missing values are filled-in with fillvalue. Iteration continues until the longest iterable is exhausted. Equivalent to:

```python
class ZipExhausted(Exception):
    pass

def zip_longest(*args, **kwds):
    fillvalue = kwds.get('fillvalue')
    counter = len(args) - 1
    def sentinel():
        nonlocal counter
        if not counter:
            raise ZipExhausted
        counter -= 1
    fillers = repeat(fillvalue)
    iterators = [chain(it, sentinel(), fillers) for it in args]
    try:
        while iterators:
            yield tuple(map(next, iterators))
    except ZipExhausted:
        pass
```

If one of the iterables is potentially infinite, then the `zip_longest()` function should be wrapped with something that limits the number of calls (for example `islice()` or `takewhile()`). If not specified, `fillvalue` defaults to `None`.

### 10.1.2 Itertools Recipes

This section shows recipes for creating an extended toolset using the existing itertools as building blocks.

The extended tools offer the same high performance as the underlying toolset. The superior memory performance is kept by processing elements one at a time rather than bringing the whole iterable into memory all at once. Code volume is kept small by linking the tools together in a functional style which helps eliminate temporary variables. High speed is retained by preferring “vectorized” building blocks over the use of for-loops and generators which incur interpreter overhead.

```python
def take(n, iterable):
    "Return first n items of the iterable as a list"
    return list(islice(iterable, n))

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return map(function, count(start))

def consume(iterator, n):
    ...
"Advance the iterator n-steps ahead. If n is none, consume entirely."
# Use functions that consume iterators at C speed.

```python
if n is None:
    # feed the entire iterator into a zero-length deque
collections.deque(iterator, maxlen=0)
else:
    # advance to the empty slice starting at position n
next(islice(iterator, n, n), None)
```

```python
def nth(iterable, n, default=None):
    "Returns the nth item or a default value"
    return next(islice(iterable, n, None), default)

def quantify(iterable, pred=bool):
    "Count how many times the predicate is true"
    return sum(map(pred, iterable))

def padnone(iterable):
    """Returns the sequence elements and then returns None indefinitely.
    Useful for emulating the behavior of the built-in map() function.
    """
    return chain(iterable, repeat(None))

def ncycles(iterable, n):
    "Returns the sequence elements n times"
    return chain.from_iterable(repeat(tuple(iterable), n))

def dotproduct(vec1, vec2):
    return sum(map(operator.mul, vec1, vec2))

def flatten(listOfLists):
    "Flatten one level of nesting"
    return chain.from_iterable(listOfLists)

def repeatfunc(func, times=None, *args):
    """Repeat calls to func with specified arguments.
    Example: repeatfunc(random.random)
    """
    if times is None:
        return starmap(func, repeat(args))
    return starmap(func, repeat(args, times))

def pairwise(iterable):
    """s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    next(b, None)
    return zip(a, b)

def grouper(iterable, n, fillvalue=None):
    """Collect data into fixed-length chunks or blocks"
    # grouper(’ABCDEF’, 3, ’x’) --> ABC DEF Gxx"
    args = [iter(iterable)] * n
    return zip_longest(*args, fillvalue=fillvalue)

def roundrobin(*iterables):
    "roundrobin(’ABC’, ’D’, ’EF’) --> A D E B F C"
```

10.1. `itertools` — Functions creating iterators for efficient looping

The Python Library Reference, Release 3.3.3
# Recipe credited to George Sakkis

```python
pending = len(iterables)
nexts = cycle((next for it in iterables))
while pending:
    try:
        for next in nexts:
            yield next()
    except StopIteration:
        pending -= 1
        nexts = cycle(islice(nexts, pending))
```

```python
def partition(pred, iterable):
    # Use a predicate to partition entries into false entries and true entries
    # partition(is_odd, range(10)) --> 0 2 4 6 8 and 1 3 5 7 9
    t1, t2 = tee(iterable)
    return filterfalse(pred, t1), filter(pred, t2)
```

```python
def powerset(iterable):
    "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))
```

```python
def unique_everseen(iterable, key=None):
    """List unique elements, preserving order. Remember all elements ever seen."""
    # unique_everseen('AAAABBBCCDAABBB') --> A B C D
    return map(next, map(itemgetter(1), groupby(iterable, key)))
```

```python
def iter_except(func, exception, first=None):
    """Call a function repeatedly until an exception is raised.
    Like builtins.iter(func, sentinel) but uses an exception instead of a sentinel to end the loop.
    
    Examples:
    iter_except(functools.partial(heappop, h), IndexError) # priority queue iterator
    iter_except(d.popitem, KeyError) # non-blocking dict iterator
    iter_except(d.popleft, IndexError) # non-blocking deque iterator
    iter_except(q.get_nowait, Queue.Empty) # loop over a producer
    iter_except(s.pop, KeyException) # non-blocking set iterator
    """
    ```
```python
try:
    if first is not None:
        yield first()
    while 1:
        yield func()
except exception:
    pass

def random_product(*args, repeat=1):
    """Random selection from itertools.product(*args, **kwds)""
    pools = [tuple(pool) for pool in args] * repeat
    return tuple(random.choice(pool) for pool in pools)

def random_permutation(iterable, r=None):
    """Random selection from itertools.permutations(iterable, r)""
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    """Random selection from itertools.combinations(iterable, r)""
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with_replacement(iterable, r):
    """Random selection from itertools.combinations_with_replacement(iterable, r)""
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.randrange(n) for i in range(r))
    return tuple(pool[i] for i in indices)
```

Note, many of the above recipes can be optimized by replacing global lookups with local variables defined as default values. For example, the `dotproduct` recipe can be written as:

```python
def dotproduct(vec1, vec2, sum=sum, map=map, mul=operator.mul):
    return sum(map(mul, vec1, vec2))
```

## 10.2 `functools` — Higher-order functions and operations on callable objects

### Source code: `Lib/functools.py`

The `functools` module is for higher-order functions: functions that act on or return other functions. In general, any callable object can be treated as a function for the purposes of this module.

The `functools` module defines the following functions:

- `functools.cmp_to_key(func)`: Transform an old-style comparison function to a key function. Used with tools that accept key functions (such as `sorted()`, `min()`, `max()`, `heapq.nlargest()`, `heapq.nsmallest()`, `itertools.groupby()`). This function is primarily used as a transition tool for programs being converted from Python 2 which supported the use of comparison functions.
A comparison function is any callable that accept two arguments, compares them, and returns a negative number for less-than, zero for equality, or a positive number for greater-than. A key function is a callable that accepts one argument and returns another value indicating the position in the desired collation sequence.

Example:

```python
class cmp_to_key:
    def __init__(self, func):
        self.func = func

    def __call__(self, x):
        return self.func(x)

    def __eq__(self, other):
        return isinstance(other, self.__class__) and self.func == other.func

    def __lt__(self, other):
        return (self.func, id(self)) < (other.func, id(other))

    def __le__(self, other):
        return self == other or self < other

    def __gt__(self, other):
        return (self.func, id(self)) > (other.func, id(other))

    def __ge__(self, other):
        return self == other or self > other

def cmp_to_key(func):
    return cmp_to_key(func)
```

New in version 3.2.

```python
@functools.lru_cache(maxsize=128, typed=False)
```

Decorator to wrap a function with a memoizing callable that saves up to the `maxsize` most recent calls. It can save time when an expensive or I/O bound function is periodically called with the same arguments.

Since a dictionary is used to cache results, the positional and keyword arguments to the function must be hashable.

If `maxsize` is set to None, the LRU feature is disabled and the cache can grow without bound. The LRU feature performs best when `maxsize` is a power-of-two.

If `typed` is set to True, function arguments of different types will be cached separately. For example, `f(3)` and `f(3.0)` will be treated as distinct calls with distinct results.

To help measure the effectiveness of the cache and tune the `maxsize` parameter, the wrapped function is instrumented with a `cache_info()` function that returns a named tuple showing `hits`, `misses`, `maxsize` and `currsize`. In a multi-threaded environment, the hits and misses are approximate.

The decorator also provides a `cache_clear()` function for clearing or invalidating the cache.

The original underlying function is accessible through the `__wrapped__` attribute. This is useful for introspection, for bypassing the cache, or for rewrapping the function with a different cache.

An LRU (least recently used) cache works best when the most recent calls are the best predictors of upcoming calls (for example, the most popular articles on a news server tend to change each day). The cache’s size limit assures that the cache does not grow without bound on long-running processes such as web servers.

Example of an LRU cache for static web content:

```python
@lru_cache(maxsize=32)
def get_pep(num):
    resource = 'http://www.python.org/dev/peps/pep-%04d/' % num
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
...    pep = get_pep(n)
...    print(n, len(pep))

>>> get_pep.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)
```

Example of efficiently computing Fibonacci numbers using a cache to implement a dynamic programming technique:

```python
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```
>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)

New in version 3.2. Changed in version 3.3: Added the typed option.

@functools.total_ordering
Given a class defining one or more rich comparison ordering methods, this class decorator supplies the rest.
This simplifies the effort involved in specifying all of the possible rich comparison operations:
The class must define one of __lt__, __le__, __gt__, or __ge__. In addition, the class
should supply an __eq__ method.

For example:

class Student:
    def __eq__(self, other):
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))

New in version 3.2.

functools.partial(func, *args, **keywords)
Return a new partial object which when called will behave like func called with the positional arguments
args and keyword arguments keywords. If more arguments are supplied to the call, they are appended to
args. If additional keyword arguments are supplied, they extend and override keywords. Roughly equivalent
to:

def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*(args + fargs), **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc

The partial() is used for partial function application which “freezes” some portion of a function’s
arguments and/or keywords resulting in a new object with a simplified signature. For example, partial() can be
used to create a callable that behaves like the int() function where the base argument defaults to
two:

>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18

functools.reduce(function, iterable[, initializer])
Apply function of two arguments cumulatively to the items of sequence, from left to right, so as to reduce
the sequence to a single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates (((1+2)+3)+4)+5). The left argument, x, is the accumulated value and the right argument,
y, is the update value from the sequence. If the optional initializer is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty. If initializer is not given and sequence contains only one item, the first item is returned.

Equivalent to:

```python
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value
```

`functools.update_wrapper(wrapper, wrapped, assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATES)`

Update a wrapper function to look like the wrapped function. The optional arguments are tuples to specify which attributes of the original function are assigned directly to the matching attributes on the wrapper function and which attributes of the wrapper function are updated with the corresponding attributes from the original function. The default values for these arguments are the module level constants WRAPPER_ASSIGNMENTS (which assigns to the wrapper function’s __name__, __module__, __annotations__ and __doc__, the documentation string) and WRAPPER_UPDATES (which updates the wrapper function’s __dict__, i.e. the instance dictionary).

To allow access to the original function for introspection and other purposes (e.g. bypassing a caching decorator such as `lru_cache()`), this function automatically adds a __wrapped__ attribute to the wrapper that refers to the original function.

The main intended use for this function is in decorator functions which wrap the decorated function and return the wrapper. If the wrapper function is not updated, the metadata of the returned function will reflect the wrapper definition rather than the original function definition, which is typically less than helpful.

`update_wrapper()` may be used with callables other than functions. Any attributes named in assigned or updated that are missing from the object being wrapped are ignored (i.e. this function will not attempt to set them on the wrapper function). AttributeError is still raised if the wrapper function itself is missing any attributes named in updated. New in version 3.2: Automatic addition of the __wrapped__ attribute. New in version 3.2: Copying of the __annotations__ attribute by default. Changed in version 3.2: Missing attributes no longer trigger an AttributeError.

`@functools.wraps(wrapped, assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATES)`

This is a convenience function for invoking `partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)` as a function decorator when defining a wrapper function. For example:

```python
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print('Calling decorated function')
...         return f(*args, **kwds)
...     return wrapper
...     
>>> @my_decorator
... def example():
...     """Docstring""
...     print('Called example function')
...     
>>> example()
```
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'

Without the use of this decorator factory, the name of the example function would have been 'wrapper', and the docstring of the original example() would have been lost.

10.2.1 partial Objects

partial objects are callable objects created by partial(). They have three read-only attributes:

partial.func
A callable object or function. Calls to the partial object will be forwarded to func with new arguments and keywords.

partial.args
The leftmost positional arguments that will be prepended to the positional arguments provided to a partial object call.

partial.keywords
The keyword arguments that will be supplied when the partial object is called.

partial objects are like function objects in that they are callable, weak referencable, and can have attributes. There are some important differences. For instance, the __name__ and __doc__ attributes are not created automatically. Also, partial objects defined in classes behave like static methods and do not transform into bound methods during instance attribute look-up.

10.3 operator — Standard operators as functions

The operator module exports a set of efficient functions corresponding to the intrinsic operators of Python. For example, operator.add(x, y) is equivalent to the expression x+y. The function names are those used for special class methods; variants without leading and trailing __ are also provided for convenience.

The functions fall into categories that perform object comparisons, logical operations, mathematical operations and sequence operations.

The object comparison functions are useful for all objects, and are named after the rich comparison operators they support:

operator.lt(a, b)
operator.le(a, b)
operator.eq(a, b)
operator.ne(a, b)
operator.ge(a, b)
operator.gt(a, b)
operator.__lt__(a, b)
operator.__le__(a, b)
operator.__eq__(a, b)
operator.__ne__(a, b)
operator.__ge__(a, b)
operator.__gt__(a, b)

Perform “rich comparisons” between a and b. Specifically, it(a, b) is equivalent to a < b, le(a, b) is equivalent to a <= b, eq(a, b) is equivalent to a == b, ne(a, b) is equivalent to a != b, gt(a, b) is equivalent to a > b and ge(a, b) is equivalent to a >= b. Note that these functions can return any value, which may or may not be interpretable as a Boolean value. See comparisons for more information about rich comparisons.
The logical operations are also generally applicable to all objects, and support truth tests, identity tests, and boolean operations:

```python
operator.not_(obj)
operator.__not__(obj)
    Return the outcome of not obj. (Note that there is no __not__() method for object instances; only the interpreter core defines this operation. The result is affected by the __bool__() and __len__() methods.)

operator.truth(obj)
    Return True if obj is true, and False otherwise. This is equivalent to using the bool constructor.

operator.is_(a, b)
    Return a is b. Tests object identity.

operator.is_not(a, b)
    Return a is not b. Tests object identity.
```

The mathematical and bitwise operations are the most numerous:

```python
operator.abs(obj)
operator.__abs__(obj)
    Return the absolute value of obj.

operator.add(a, b)
operator.__add__(a, b)
    Return a + b, for a and b numbers.

operator.and_(a, b)
operator.__and__(a, b)
    Return the bitwise and of a and b.

operator.floordiv(a, b)
operator.__floordiv__(a, b)
    Return a // b.

operator.index(a)
operator.__index__(a)
    Return a converted to an integer. Equivalent to a.__index__().

operator.inv(obj)
operator.invert(obj)
operator.__inv__(obj)
operator.__invert__(obj)
    Return the bitwise inverse of the number obj. This is equivalent to ~obj.

operator.lshift(a, b)
operator.__lshift__(a, b)
    Return a shifted left by b.

operator.mod(a, b)
operator.__mod__(a, b)
    Return a % b.

operator.mul(a, b)
operator.__mul__(a, b)
    Return a * b, for a and b numbers.

operator.neg(obj)
operator.__neg__(obj)
    Return obj negated (~obj).

operator.or_(a, b)
operator.__or__(a, b)
    Return the bitwise or of a and b.

operator.pos(obj)
```
The Python Library Reference, Release 3.3.3

operator.__pos__(obj)
    Return obj positive (+obj).

operator.pow(a, b)
operator.__pow__(a, b)
    Return a ** b, for a and b numbers.

operator.rshift(a, b)
operator.__rshift__(a, b)
    Return a shifted right by b.

operator.sub(a, b)
operator.__sub__(a, b)
    Return a - b.

operator.truediv(a, b)
operator.__truediv__(a, b)
    Return a / b where 2/3 is .66 rather than 0. This is also known as “true” division.

operator.xor(a, b)
operator.__xor__(a, b)
    Return the bitwise exclusive or of a and b.

Operations which work with sequences (some of them with mappings too) include:

operator.concat(a, b)
operator.__concat__(a, b)
    Return a + b for a and b sequences.

operator.contains(a, b)
operator.__contains__(a, b)
    Return the outcome of the test b in a. Note the reversed operands.

operator.countOf(a, b)
operator.__countOf__(a, b)
    Return the number of occurrences of b in a.

operator.delitem(a, b)
operator.__delitem__(a, b)
    Remove the value of a at index b.

operatorgetitem(a, b)
operator.__getitem__(a, b)
    Return the value of a at index b.

operator.indexOf(a, b)
operator.__indexOf__(a, b)
    Return the index of the first of occurrence of b in a.

operator.setItem(a, b, c)
operator.__setItem__(a, b, c)
    Set the value of a at index b to c.

Example: Build a dictionary that maps the ordinals from 0 to 255 to their character equivalents.

```python
def build_dictionary():
    d = {}
    keys = range(256)
    vals = map(chr, keys)
    map(operator.setItem, [d] * len(keys), keys, vals)
    return d
```

The operator module also defines tools for generalized attribute and item lookups. These are useful for making fast field extractors as arguments for `map()`, `sorted()`, `itertools.groupby()`, or other functions that expect a function argument.

operator.attrgetter(attr)
operator.attrgetter(*attrs)
    Return a callable object that fetches attr from its operand. If more than one attribute is requested, returns a tuple of attributes. The attribute names can also contain dots. For example:
• After \( f = \text{attrgetter}('\text{name}') \), the call \( f(b) \) returns \( b.\text{name} \).

• After \( f = \text{attrgetter}('\text{name}', '\text{date}') \), the call \( f(b) \) returns \( (b.\text{name}, b.\text{date}) \).

• After \( f = \text{attrgetter}('\text{name.first}', '\text{name.last}') \), the call \( f(b) \) returns \( (r.\text{name.first}, r.\text{name.last}) \).

Equivalent to:

```python
def attrgetter(*items):
    if any(not isinstance(item, str) for item in items):
        raise TypeError('attribute name must be a string')
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
        def g(obj):
            return tuple(resolve_attr(obj, attr) for attr in items)
    return g

def resolve_attr(obj, attr):
    for name in attr.split('.'):  
        obj = getattr(obj, name)
    return obj
```

operator.itemgetter(item)
operator.itemgetter(*items)

Return a callable object that fetches `item` from its operand using the operand’s `__getitem__()` method. If multiple items are specified, returns a tuple of lookup values. For example:

• After \( f = \text{itemgetter}(2) \), the call \( f(r) \) returns \( r[2] \).

• After \( g = \text{itemgetter}(2, 5, 3) \), the call \( g(r) \) returns \( (r[2], r[5], r[3]) \).

Equivalent to:

```python
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

The items can be any type accepted by the operand’s `__getitem__()` method. Dictionaries accept any hashable value. Lists, tuples, and strings accept an index or a slice:

```text
>>> itemgetter(1)('ABCDEFG')
'B'
>>> itemgetter(1,3,5)('ABCDEFG')
('B', 'D', 'F')
>>> itemgetter(slice(2,None))('ABCDEFG')
'CDEFG'
```

Example of using `itemgetter()` to retrieve specific fields from a tuple record:

```text
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
```
```python
>>> list(map(getcount, inventory))
[3, 2, 5, 1]
```
```python
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

**operator**

Return a callable object that calls the method `name` on its operand. If additional arguments and/or keyword arguments are given, they will be given to the method as well. For example:

*After `f = methodcaller('name')`, the call `f(b)` returns `b.name()`.*

*After `f = methodcaller('name', 'foo', bar=1)`, the call `f(b)` returns `b.name('foo', bar=1)`.*

Equivalent to:

```python
def methodcaller(name, *args, **kwargs):
    def caller(obj):
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

10.3.1 Mapping Operators to Functions

This table shows how abstract operations correspond to operator symbols in the Python syntax and the functions in the `operator` module.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td><code>a + b</code></td>
<td><code>add(a, b)</code></td>
</tr>
<tr>
<td>Concatenation</td>
<td><code>seq1 + seq2</code></td>
<td><code>concat(seq1, seq2)</code></td>
</tr>
<tr>
<td>Containment Test</td>
<td><code>obj in seq</code></td>
<td><code>contains(seq, obj)</code></td>
</tr>
<tr>
<td>Division</td>
<td><code>a / b</code></td>
<td><code>truediv(a, b)</code></td>
</tr>
<tr>
<td>Division</td>
<td><code>a // b</code></td>
<td><code>floordiv(a, b)</code></td>
</tr>
<tr>
<td>Bitwise And</td>
<td><code>a &amp; b</code></td>
<td><code>and_(a, b)</code></td>
</tr>
<tr>
<td>Bitwise Exclusive Or</td>
<td><code>a ^ b</code></td>
<td><code>xor(a, b)</code></td>
</tr>
<tr>
<td>Bitwise Inversion</td>
<td><code>~ a</code></td>
<td><code>invert(a)</code></td>
</tr>
<tr>
<td>Bitwise Or</td>
<td>`a</td>
<td>b`</td>
</tr>
<tr>
<td>Exponentiation</td>
<td><code>a ** b</code></td>
<td><code>pow(a, b)</code></td>
</tr>
<tr>
<td>Identity</td>
<td><code>a is b</code></td>
<td><code>is_(a, b)</code></td>
</tr>
<tr>
<td>Identity</td>
<td><code>a is not b</code></td>
<td><code>is_not(a, b)</code></td>
</tr>
<tr>
<td>Indexed Assignment</td>
<td><code>obj[k] = v</code></td>
<td><code>setitem(obj, k, v)</code></td>
</tr>
<tr>
<td>Indexed Deletion</td>
<td><code>del obj[k]</code></td>
<td><code>delitem(obj, k)</code></td>
</tr>
<tr>
<td>Indexing</td>
<td><code>obj[k]</code></td>
<td><code>getitem(obj, k)</code></td>
</tr>
<tr>
<td>Left Shift</td>
<td><code>a &lt;&lt; b</code></td>
<td><code>lshift(a, b)</code></td>
</tr>
<tr>
<td>Modulo</td>
<td><code>a % b</code></td>
<td><code>mod(a, b)</code></td>
</tr>
<tr>
<td>Multiplication</td>
<td><code>a * b</code></td>
<td><code>mul(a, b)</code></td>
</tr>
<tr>
<td>Negation (Arithmetic)</td>
<td><code>- a</code></td>
<td><code>neg(a)</code></td>
</tr>
<tr>
<td>Negation (Logical)</td>
<td><code>not a</code></td>
<td><code>not_(a)</code></td>
</tr>
<tr>
<td>Positive</td>
<td><code>+ a</code></td>
<td><code>pos(a)</code></td>
</tr>
<tr>
<td>Right Shift</td>
<td><code>a &gt;&gt; b</code></td>
<td><code>rshift(a, b)</code></td>
</tr>
<tr>
<td>Slice Assignment</td>
<td><code>seq[i:j] = values</code></td>
<td><code>setitem(seq, slice(i, j), values)</code></td>
</tr>
<tr>
<td>Slice Deletion</td>
<td><code>del seq[i:j]</code></td>
<td><code>delitem(seq, slice(i, j))</code></td>
</tr>
<tr>
<td>Slicing</td>
<td><code>seq[i:j]</code></td>
<td><code>getitem(seq, slice(i, j))</code></td>
</tr>
<tr>
<td>String Formatting</td>
<td><code>s % obj</code></td>
<td><code>mod(s, obj)</code></td>
</tr>
<tr>
<td>Subtraction</td>
<td><code>a - b</code></td>
<td><code>sub(a, b)</code></td>
</tr>
<tr>
<td>Truth Test</td>
<td><code>obj</code></td>
<td><code>truth(obj)</code></td>
</tr>
<tr>
<td>Ordering</td>
<td><code>a &lt; b</code></td>
<td><code>lt(a, b)</code></td>
</tr>
<tr>
<td>Ordering</td>
<td><code>a &lt;= b</code></td>
<td><code>le(a, b)</code></td>
</tr>
<tr>
<td>Equality</td>
<td><code>a == b</code></td>
<td><code>eq(a, b)</code></td>
</tr>
</tbody>
</table>

Continued on next page
10.3.2 Inplace Operators

Many operations have an “in-place” version. Listed below are functions providing a more primitive access
in-place operators than the usual syntax does; for example, the statement \( x += y \) is equivalent to \( x = \)
operator.iadd(x, y). Another way to put it is to say that \( z = \) operator.iadd(x, y) is equivalent
to the compound statement \( z = x; \) \( z += y \).

In those examples, note that when an in-place method is called, the computation and assignment are performed
in two separate steps. The in-place functions listed below only do the first step, calling the in-place method. The
second step, assignment, is not handled.

For immutable targets such as strings, numbers, and tuples, the updated value is computed, but not assigned back
to the input variable:

```python
>>> a = 'hello'
>>> iadd(a, ' world')
'hello world'
>>> a
'hello'
```

For mutable targets such as lists and dictionaries, the inplace method will perform the update, so no subsequent
assignment is necessary:

```python
>>> s = ['h', 'e', 'l', 'l', 'o']
>>> iadd(s, [' ', 'w', 'o', 'r', 'l', 'd'])
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> s
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

```python
operator.iadd(a, b)
operator._iadd__(a, b)
    a = iadd(a, b) is equivalent to a += b.
operator.iand(a, b)
operator._iand__(a, b)
    a = iand(a, b) is equivalent to a &= b.
operator.iconcat(a, b)
operator._iconcat__(a, b)
    a = iconcat(a, b) is equivalent to a += b for a and b sequences.
operator.ifloordiv(a, b)
operator._ifloordiv__(a, b)
    a = ifloordiv(a, b) is equivalent to a //= b.
operator.ilshift(a, b)
operator._ilshift__(a, b)
    a = ilshift(a, b) is equivalent to a <<= b.
operator.imod(a, b)
operator._imod__(a, b)
    a = imod(a, b) is equivalent to a %= b.
operator.imul(a, b)
operator._imul__(a, b)
    a = imul(a, b) is equivalent to a *= b.
operator.iord(a, b)
```

---

Table 10.1 – continued from previous page

<table>
<thead>
<tr>
<th>Difference</th>
<th>a != b</th>
<th>ne(a, b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ordering</td>
<td>a &gt;= b</td>
<td>ge(a, b)</td>
</tr>
<tr>
<td>Ordering</td>
<td>a &gt; b</td>
<td>gt(a, b)</td>
</tr>
</tbody>
</table>
operator.__ior__(a, b)
    a = ior(a, b) is equivalent to a |= b.
operator.ipow(a, b)
operator.__ipow__(a, b)
    a = ipow(a, b) is equivalent to a **= b.
operator.irshift(a, b)
operator.__irshift__(a, b)
    a = irshift(a, b) is equivalent to a >>= b.
operator.isub(a, b)
operator.__isub__(a, b)
    a = isub(a, b) is equivalent to a -= b.
operator.itruediv(a, b)
operator.__itruediv__(a, b)
    a = itruediv(a, b) is equivalent to a /= b.
operator.ixor(a, b)
operator.__ixor__(a, b)
    a = ixor(a, b) is equivalent to a ^= b.
The modules described in this chapter deal with disk files and directories. For example, there are modules for reading the properties of files, manipulating paths in a portable way, and creating temporary files. The full list of modules in this chapter is:

11.1 os.path — Common pathname manipulations

This module implements some useful functions on pathnames. To read or write files see open(), and for accessing the filesystem see the os module. The path parameters can be passed as either strings, or bytes. Applications are encouraged to represent file names as (Unicode) character strings. Unfortunately, some file names may not be representable as strings on Unix, so applications that need to support arbitrary file names on Unix should use bytes objects to represent path names. Vice versa, using bytes objects cannot represent all file names on Windows (in the standard mbcs encoding), hence Windows applications should use string objects to access all files.

Unlike a unix shell, Python does not do any automatic path expansions. Functions such as expanduser() and expandvars() can be invoked explicitly when an application desires shell-like path expansion. (See also the glob module.)

Note: All of these functions accept either only bytes or only string objects as their parameters. The result is an object of the same type, if a path or file name is returned.

Note: Since different operating systems have different path name conventions, there are several versions of this module in the standard library. The os.path module is always the path module suitable for the operating system Python is running on, and therefore usable for local paths. However, you can also import and use the individual modules if you want to manipulate a path that is always in one of the different formats. They all have the same interface:

- posixpath for UNIX-style paths
- ntpath for Windows paths
- macpath for old-style MacOS paths
- os2emxpath for OS/2 EMX paths

os.path.abspath(path)
Return a normalized absolutized version of the pathname path. On most platforms, this is equivalent to calling the function normpath() as follows: normpath(join(os.getcwd(), path)).

os.path.basename(path)
Return the base name of pathname path. This is the second element of the pair returned by passing path to the function split(). Note that the result of this function is different from the Unix basename program; where basename for '/foo/bar/' returns 'bar', the basename() function returns an empty string ('').
os.path.commonprefix(list)
Return the longest path prefix (taken character-by-character) that is a prefix of all paths in list. If list is empty, return the empty string (""). Note that this may return invalid paths because it works a character at a time.

os.path.dirname(path)
Return the directory name of pathname path. This is the first element of the pair returned by passing path to the function split().

os.path.exists(path)
Return True if path refers to an existing path or an open file descriptor. Returns False for broken symbolic links. On some platforms, this function may return False if permission is not granted to execute os.stat() on the requested file, even if the path physically exists. Changed in version 3.3: path can now be an integer: True is returned if it is an open file descriptor, False otherwise.

os.path.lexists(path)
Return True if path refers to an existing path. Returns True for broken symbolic links. Equivalent to exists() on platforms lacking os.lstat().

os.path.expanduser(path)
On Unix and Windows, return the argument with an initial component of ~ or ~user replaced by that user's home directory.

On Unix, an initial ~ is replaced by the environment variable HOME if it is set; otherwise the current user’s home directory is looked up in the password directory through the built-in module pwd. An initial ~user is looked up directly in the password directory.

On Windows, HOME and USERPROFILE will be used if set, otherwise a combination of HOMEPATH and HOMEDRIVE will be used. An initial ~user is handled by stripping the last directory component from the created user path derived above.

If the expansion fails or if the path does not begin with a tilde, the path is returned unchanged.

os.path.expandvars(path)
Return the argument with environment variables expanded. Substrings of the form $name or ${name} are replaced by the value of environment variable name. Malformed variable names and references to non-existing variables are left unchanged.

On Windows, %name% expansions are supported in addition to $name and ${name}.

os.path.getatime(path)
Return the time of last access of path. The return value is a number giving the number of seconds since the epoch (see the time module). Raise OSError if the file does not exist or is inaccessible.

If os.stat_float_times() returns True, the result is a floating point number.

os.path.getctime(path)
Return the system’s ctime which, on some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time for path. The return value is a number giving the number of seconds since the epoch (see the time module). Raise OSError if the file does not exist or is inaccessible.

os.path.getsize(path)
Return the size, in bytes, of path. Raise OSError if the file does not exist or is inaccessible.

os.path.isabs(path)
Return True if path is an absolute pathname. On Unix, that means it begins with a slash, on Windows that it begins with a (back)slash after chopping off a potential drive letter.

os.path.isfile(path)
Return True if path is an existing regular file. This follows symbolic links, so both islink() and.isfile() can be true for the same path.
os.path.isdir(path)
    Return True if path is an existing directory. This follows symbolic links, so both islink() and
    isdir() can be true for the same path.

os.path.islink(path)
    Return True if path refers to a directory entry that is a symbolic link. Always False if symbolic links are
    not supported.

os.path.ismount(path)
    Return True if pathname path is a mount point: a point in a file system where a different file system has
    been mounted. The function checks whether path’s parent, path/.., is on a different device than path, or
    whether path/.. and path point to the same i-node on the same device — this should detect mount points
    for all Unix and POSIX variants.

os.path.join(path1[, path2[, ...]])
    Join one or more path components intelligently. If any component is an absolute path, all previous com-
    ponents (on Windows, including the previous drive letter, if there was one) are thrown away, and joining
    continues. The return value is the concatenation of path1, and optionally path2, etc., with exactly one direc-
    tory separator (os.sep) following each non-empty part except the last. (This means that an empty last part
    will result in a path that ends with a separator.) Note that on Windows, since there is a current directory for
each drive, os.path.join("c:\", "foo") represents a path relative to the current directory on drive
C:\(c:\foo), not c:\foo.

os.path.normcase(path)
    Normalize the case of a pathname. On Unix and Mac OS X, this returns the path unchanged; on case-
    insensitive filesystems, it converts the path to lowercase. On Windows, it also converts forward slashes to
    backward slashes. Raise a TypeError if the type of path is not str or bytes.

os.path.normpath(path)
    Normalize a pathname by collapsing redundant separators and up-level references so that A//B, A/B/,
    A/./B and A/foo/../B all become A/B. This string manipulation may change the meaning of a path
    that contains symbolic links. On Windows, it converts forward slashes to backward slashes. To normalize
case, use normcase().

os.path.realpath(path)
    Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path
    (if they are supported by the operating system).

os.path.relpath(path, start=None)
    Return a relative filepath to path either from the current directory or from an optional start directory. This
    is a path computation: the filesystem is not accessed to confirm the existence or nature of path or start.

    start defaults to os.curdir.

    Availability: Unix, Windows.

os.path.samefile(path1, path2)
    Return True if both pathname arguments refer to the same file or directory. On Unix, this is determined
    by the device number and i-node number and raises an exception if a os.stat() call on either pathname
    fails.

    On Windows, two files are the same if they resolve to the same final path name using the Windows API call
GetFinalPathNameByHandle. This function raises an exception if handles cannot be obtained to either file.


os.path.sameopenfile(fp1, fp2)
    Return True if the file descriptors fp1 and fp2 refer to the same file.


os.path.samestat(stat1, stat2)
    Return True if the stat tuples stat1 and stat2 refer to the same file. These structures may have been returned
    by os.fstat(), os.lstat(), or os.stat(). This function implements the underlying comparison
    used by samefile() and sameopenfile().
The Python Library Reference, Release 3.3.3

Availability: Unix.

os.path.split(path)
Split the pathname path into a pair, (head, tail) where tail is the last pathname component and head is everything leading up to that. The tail part will never contain a slash; if path ends in a slash, tail will be empty. If there is no slash in path, head will be empty. If path is empty, both head and tail are empty. Trailing slashes are stripped from head unless it is the root (one or more slashes only). In all cases, join(head, tail) returns a path to the same location as path (but the strings may differ). Also see the functions dirname() and basename().

os.path.splitdrive(path)
Split the pathname path into a pair (drive, tail) where drive is either a mount point or the empty string. On systems which do not use drive specifications, drive will always be the empty string. In all cases, drive + tail will be the same as path.
On Windows, splits a pathname into drive/UNC sharepoint and relative path.
If the path contains a drive letter, drive will contain everything up to and including the colon. e.g. splitdrive("c:/dir") returns ("c:", "/dir")
If the path contains a UNC path, drive will contain the host name and share, up to but not including the fourth separator. e.g. splitdrive("//host/computer/dir") returns ("//host/computer", "/dir")

os.path.splitext(path)
Split the pathname path into a pair (root, ext) such that root + ext == path, and ext is empty or begins with a period and contains at most one period. Leading periods on the basename are ignored; splitext('.cshrc') returns ('.cshrc', ").

os.path.splitunc(path)
Deprecated since version 3.1: Use splitdrive instead. Split the pathname path into a pair (unc, rest) so that unc is the UNC mount point (such as r'\host\mount'), if present, and rest the rest of the path (such as r'\path\file.ext'). For paths containing drive letters, unc will always be the empty string.
Availability: Windows.

os.path.supports_unicode_filenames
True if arbitrary Unicode strings can be used as file names (within limitations imposed by the file system).

11.2 fileinput — Iterate over lines from multiple input streams

Source code: Lib/fileinput.py

This module implements a helper class and functions to quickly write a loop over standard input or a list of files. If you just want to read or write one file see open().

The typical use is:

import fileinput
for line in fileinput.input():
    process(line)

This iterates over the lines of all files listed in sys.argv[1:], defaulting to sys.stdin if the list is empty. If a filename is ‘-‘, it is also replaced by sys.stdin. To specify an alternative list of filenames, pass it as the first argument to input(). A single file name is also allowed.

All files are opened in text mode by default, but you can override this by specifying the mode parameter in the call to input() or FileInput. If an I/O error occurs during opening or reading a file, OSError is raised. Changed in version 3.3: IOError used to be raised; it is now an alias of OSError. If sys.stdin is used more than once, the second and further use will return no lines, except perhaps for interactive use, or if it has been explicitly reset (e.g. using sys.stdin.seek(0)).
Empty files are opened and immediately closed; the only time their presence in the list of filenames is noticeable at all is when the last file opened is empty.

Lines are returned with any newlines intact, which means that the last line in a file may not have one.

You can control how files are opened by providing an opening hook via the `openhook` parameter to `fileinput.input()` or `FileInput()`. The hook must be a function that takes two arguments, `filename` and `mode`, and returns an accordingly opened file-like object. Two useful hooks are already provided by this module.

The following function is the primary interface of this module:

```python
fileinput.input (files=None, inplace=False, backup='', bufsize=0, mode='r', openhook=None)
```

Create an instance of the `FileInput` class. The instance will be used as global state for the functions of this module, and is also returned to use during iteration. The parameters to this function will be passed along to the constructor of the `FileInput` class.

The `FileInput` instance can be used as a context manager in the `with` statement. In this example, `input` is closed after the `with` statement is exited, even if an exception occurs:

```python
with fileinput.input(files=('spam.txt', 'eggs.txt')) as f:
    for line in f:
        process(line)
```

Changed in version 3.2: Can be used as a context manager.

The following functions use the global state created by `fileinput.input()`; if there is no active state, `RuntimeError` is raised.

```python
fileinput.filename ()
```

Return the name of the file currently being read. Before the first line has been read, returns `None`.

```python
fileinput.fileno ()
```

Return the integer “file descriptor” for the current file. When no file is opened (before the first line and between files), returns `-1`.

```python
fileinput.lineno ()
```

Return the cumulative line number of the line that has just been read. Before the first line has been read, returns `0`. After the last line of the last file has been read, returns the line number of that line.

```python
fileinput.filelineno ()
```

Return the line number in the current file. Before the first line has been read, returns `0`. After the last line of the last file has been read, returns the line number of that line within the file.

```python
fileinput.isfirstline ()
```

Returns true if the line just read is the first line of its file, otherwise returns `false`.

```python
fileinput.isstdin ()
```

Returns true if the last line was read from `sys.stdin`, otherwise returns `false`.

```python
fileinput.nextfile ()
```

Close the current file so that the next iteration will read the first line from the next file (if any); lines not read from the file will not count towards the cumulative line count. The filename is not changed until after the first line of the next file has been read. Before the first line has been read, this function has no effect; it cannot be used to skip the first file. After the last line of the last file has been read, this function has no effect.

```python
fileinput.close ()
```

Close the sequence.

The class which implements the sequence behavior provided by the module is available for subclassing as well:

```python
class fileinput.FileInput (files=None, inplace=False, backup='', bufsize=0, mode='r', openhook=None)
```

Class `FileInput` is the implementation; its methods `filename()`, `fileno()`, `lineno()`, `filelineno()`, `isfirstline()`, `isstdin()`, `nextfile()` and `close()` correspond to the
functions of the same name in the module. In addition it has a `readline()` method which returns the
next input line, and a `__getitem__()` method which implements the sequence behavior. The sequence
must be accessed in strictly sequential order; random access and `readline()` cannot be mixed.

With `mode` you can specify which file mode will be passed to `open()`. It must be one of `'r'`, `'rU'`, `'U'
and `'rb'`.

The `openhook`, when given, must be a function that takes two arguments, `filename` and `mode`, and returns an
accordingly opened file-like object. You cannot use `inplace` and `openhook` together.

A `FileInput` instance can be used as a context manager in the `with` statement. In this example, `input` is
closed after the `with` statement is exited, even if an exception occurs:

```python
with FileInput(files=('spam.txt', 'eggs.txt')) as input:
    process(input)
```

Changed in version 3.2: Can be used as a context manager.

**Optional in-place filtering:** if the keyword argument `inplace=True` is passed to `fileinput.input()` or
to the `FileInput` constructor, the file is moved to a backup file and standard output is directed to the input file
(if a file of the same name as the backup file already exists, it will be replaced silently). This makes it possible to
write a filter that rewrites its input file in place. If the `backup` parameter is given (typically as `backup='.<some
extension>'`), it specifies the extension for the backup file, and the backup file remains around; by default, the
extension is `'.bak'` and it is deleted when the output file is closed. In-place filtering is disabled when standard
input is read.

The two following opening hooks are provided by this module:

```python
fileinput.hook_compressed(filename, mode)
```

Transparently opens files compressed with gzip and bzip2 (recognized by the extensions `'.gz'` and
`'.bz2'`) using the `gzip` and `bz2` modules. If the filename extension is not `'.gz'` or `'.bz2'`, the
file is opened normally (i.e., using `open()` without any decompression).

Usage example: `fi = fileinput.FileInput(openhook=fileinput.hook_compressed)`

```python
fileinput.hook_encoded(encoding)
```

Returns a hook which opens each file with `codecs.open()`, using the given `encoding` to read the file.

Usage example: `fi = fileinput.FileInput(openhook=fileinput.hook_encoded("iso-8859-1"))`

### 11.3 stat — Interpreting stat() results

**Source code:** `Lib/stat.py`

The `stat` module defines constants and functions for interpreting the results of `os.stat()`, `os.fstat()` and
`os.lstat()` (if they exist). For complete details about the `stat()`, `fstat()` and `lstat()` calls, consult
the documentation for your system.

The `stat` module defines the following functions to test for specific file types:

```python
stat.S_ISDIR(mode)
```

Return non-zero if the mode is from a directory.

```python
stat.S_ISCHR(mode)
```

Return non-zero if the mode is from a character special device file.

```python
stat.S_ISBLK(mode)
```

Return non-zero if the mode is from a block special device file.

```python
stat.S_ISREG(mode)
```

Return non-zero if the mode is from a regular file.
The Python Library Reference, Release 3.3.3

\[
\text{stat.} \text{S_ISFIFO}(\text{mode}) \\
\text{Return non-zero if the mode is from a FIFO (named pipe).}
\]

\[
\text{stat.} \text{S_ISLNK}(\text{mode}) \\
\text{Return non-zero if the mode is from a symbolic link.}
\]

\[
\text{stat.} \text{S_ISSOCK}(\text{mode}) \\
\text{Return non-zero if the mode is from a socket.}
\]

Two additional functions are defined for more general manipulation of the file’s mode:

\[
\text{stat.} \text{S_IMODE}(\text{mode}) \\
\text{Return the portion of the file’s mode that can be set by os.chmod()—that is, the file’s permission bits, plus the sticky bit, set-group-id, and set-user-id bits (on systems that support them).}
\]

\[
\text{stat.} \text{S_IFMT}(\text{mode}) \\
\text{Return the portion of the file’s mode that describes the file type (used by the S_IS*() functions above).}
\]

Normally, you would use the os.path.is*() functions for testing the type of a file; the functions here are useful when you are doing multiple tests of the same file and wish to avoid the overhead of the stat() system call for each test. These are also useful when checking for information about a file that isn’t handled by os.path, like the tests for block and character devices.

Example:

```python
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted at top, calling the callback function for each regular file'"
    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.stat(pathname).st_mode
        if S_ISDIR(mode):
            # It’s a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It’s a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print('Skipping %s' % pathname)

def visitfile(file):
    print('visiting', file)

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

An additional utility function is provided to covert a file’s mode in a human readable string:

\[
\text{stat.} \text{filemode}(\text{mode}) \\
\text{Convert a file’s mode to a string of the form `-rwxrwxrwx’. New in version 3.3.}
\]

All the variables below are simply symbolic indexes into the 10-tuple returned by os.stat(), os.fstat() or os.lstat().

\[
\text{stat.} \text{ST_MODE} \\
\text{Inode protection mode.}
\]

\[
\text{stat.} \text{ST_INO} \\
\text{Inode number.}
\]
The Python Library Reference, Release 3.3.3

stat.ST_DEV
Device inode resides on.

stat.ST_NLINK
Number of links to the inode.

stat.ST_UID
User id of the owner.

stat.ST_GID
Group id of the owner.

stat.ST_SIZE
Size in bytes of a plain file; amount of data waiting on some special files.

stat.ST_ATIME
Time of last access.

stat.ST_MTIME
Time of last modification.

stat.ST_CTIME
The “ctime” as reported by the operating system. On some systems (like Unix) is the time of the last meta-
data change, and, on others (like Windows), is the creation time (see platform documentation for details).

The interpretation of “file size” changes according to the file type. For plain files this is the size of the file in bytes. For FIFOs and sockets under most flavors of Unix (including Linux in particular), the “size” is the number of bytes waiting to be read at the time of the call to os.stat(), os.fstat(), or os.lstat(); this can sometimes be useful, especially for polling one of these special files after a non-blocking open. The meaning of the size field for other character and block devices varies more, depending on the implementation of the underlying system call.

The variables below define the flags used in the ST_MODE field.

Use of the functions above is more portable than use of the first set of flags:

stat.S_IFSOCK
Socket.

stat.S_IFLNK
Symbolic link.

stat.S_IFREG
Regular file.

stat.S_IFBLK
Block device.

stat.S_IFDIR
Directory.

stat.S_IFCHR
Character device.

stat.S_IFIFO
FIFO.

The following flags can also be used in the mode argument of os.chmod():

stat.S_ISUID
Set UID bit.

stat.S_ISGID
Set-group-ID bit. This bit has several special uses. For a directory it indicates that BSD semantics is to be used for that directory: files created there inherit their group ID from the directory, not from the effective group ID of the creating process, and directories created there will also get the S_ISGID bit set. For a file that does not have the group execution bit (S_IXGRP) set, the set-group-ID bit indicates mandatory file/record locking (see also S_ENFMT).
The following flags can be used in the flags argument of os.chflags():

- **stat.S_IFMT**: System V file locking enforcement. This flag is shared with **S_ISGID**: file/record locking is enforced on files that do not have the group execution bit (**S_IXGRP**) set.

- **stat.S_IWRITE**: Unix V7 synonym for **S_IWUSR**.

- **stat.S_IEXEC**: Unix V7 synonym for **S_IXUSR**.

- **stat.S_IRWXU**: Mask for file owner permissions.

- **stat.S_IRUSR**: Owner has read permission.

- **stat.S_IWUSR**: Owner has write permission.

- **stat.S_IXUSR**: Owner has execute permission.

- **stat.S_IRWXG**: Mask for group permissions.

- **stat.S_IRGRP**: Group has read permission.

- **stat.S_IWGRP**: Group has write permission.

- **stat.S_IXGRP**: Group has execute permission.

- **stat.S_IRWXO**: Mask for permissions for others (not in group).

- **stat.S_IROTH**: Others have read permission.

- **stat.S_IWOTH**: Others have write permission.

- **stat.S_IXOTH**: Others have execute permission.
The Python Library Reference, Release 3.3.3

The file is stored compressed (Mac OS X 10.6+).

The file should not be displayed in a GUI (Mac OS X 10.5+).

The file may be archived.

The file may not be changed.

The file may only be appended to.

The file may not be renamed or deleted.

The file is a snapshot file.

See the *BSD or Mac OS systems man page *chflags(2)* for more information.

11.4 filecmp — File and Directory Comparisons

Source code: Lib/filecmp.py

The filecmp module defines functions to compare files and directories, with various optional time/correctness trade-offs. For comparing files, see also the difflib module.

The filecmp module defines the following functions:

filecmp.cmp(f1, f2, shallow=True)

Compare the files named f1 and f2, returning True if they seem equal, False otherwise.

If shallow is true, files with identical os.stat() signatures are taken to be equal. Otherwise, the contents of the files are compared.

Note that no external programs are called from this function, giving it portability and efficiency.

filecmp.cmpfiles(dir1, dir2, common, shallow=True)

Compare the files in the two directories dir1 and dir2 whose names are given by common.

Returns three lists of file names: match, mismatch, errors. match contains the list of files that match, mismatch contains the names of those that don’t, and errors lists the names of files which could not be compared. Files are listed in errors if they don’t exist in one of the directories, the user lacks permission to read them or if the comparison could not be done for some other reason.

The shallow parameter has the same meaning and default value as for filecmp.cmp().

For example, cmpfiles(‘a’, ‘b’, [‘c’, ‘d/e’]) will compare a/c with b/c and a/d/e with b/d/e. ‘c’ and ‘d/e’ will each be in one of the three returned lists.

11.4.1 The dircmp class

class filecmp.dircmp(a, b, ignore=None, hide=None)

Construct a new directory comparison object, to compare the directories a and b. ignore is a list of names to ignore, and defaults to [‘RCS’, ‘CVS’, ‘tags’]. hide is a list of names to hide, and defaults to [os.curdir, os.pardir].

The dircmp class compares files by doing shallow comparisons as described for filecmp.cmp().

The dircmp class provides the following methods:
report()  
Print (to sys.stdout) a comparison between a and b.

report_partial_closure()  
Print a comparison between a and b and common immediate subdirectories.

report_full_closure()  
Print a comparison between a and b and common subdirectories (recursively).

The dircmp class offers a number of interesting attributes that may be used to get various bits of information about the directory trees being compared.

Note that via __getattr__() hooks, all attributes are computed lazily, so there is no speed penalty if only those attributes which are lightweight to compute are used.

left  
The directory a.

right  
The directory b.

left_list  
Files and subdirectories in a, filtered by hide and ignore.

right_list  
Files and subdirectories in b, filtered by hide and ignore.

common  
Files and subdirectories in both a and b.

left_only  
Files and subdirectories only in a.

right_only  
Files and subdirectories only in b.

common_dirs  
Subdirectories in both a and b.

common_files  
Files in both a and b

common_funny  
Names in both a and b, such that the type differs between the directories, or names for which os.stat() reports an error.

same_files  
Files which are identical in both a and b, using the class’s file comparison operator.

diff_files  
Files which are in both a and b, whose contents differ according to the class’s file comparison operator.

funny_files  
Files which are in both a and b, but could not be compared.

subdirs  
A dictionary mapping names in common_dirs to dircmp objects.

Here is a simplified example of using the subdirs attribute to search recursively through two directories to show common different files:

```python
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...     for name in dcmp.diff_files:
...         print("diff_file %s found in %s and %s" % (name, dcmp.left, dcmp.right))
...     for sub_dcmp in dcmp.subdirs.values():
...         print_diff_files(sub_dcmp)
```

11.4. filecmp — File and Directory Comparisons
... dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)

11.5 tempfile — Generate temporary files and directories

Source code: Lib/tempfile.py

This module generates temporary files and directories. It works on all supported platforms. It provides three new functions, NamedTemporaryFile(), mkstemp(), and mkdtemp(), which should eliminate all remaining need to use the insecure mktemp() function. Temporary file names created by this module no longer contain the process ID; instead a string of six random characters is used.

Also, all the user-callable functions now take additional arguments which allow direct control over the location and name of temporary files. It is no longer necessary to use the global tempdir variable. To maintain backward compatibility, the argument order is somewhat odd; it is recommended to use keyword arguments for clarity.

The module defines the following user-callable items:

tempfile.TemporaryFile (mode='w+b', buffering=None, encoding=None, newline=None, suffix='', prefix='tmp', dir=None)

Return a file-like object that can be used as a temporary storage area. The file is created using mkstemp(). It will be destroyed as soon as it is closed (including an implicit close when the object is garbage collected). Under Unix, the directory entry for the file is removed immediately after the file is created. Other platforms do not support this; your code should not rely on a temporary file created using this function having or not having a visible name in the file system.

The mode parameter defaults to ‘w+b’ so that the file created can be read and written without being closed. Binary mode is used so that it behaves consistently on all platforms without regard for the data that is stored. buffering, encoding and newline are interpreted as for open().

The dir, prefix and suffix parameters are passed to mkstemp().

The returned object is a true file object on POSIX platforms. On other platforms, it is a file-like object whose file attribute is the underlying true file object. This file-like object can be used in a with statement, just like a normal file.

tempfile.NamedTemporaryFile (mode='w+b', buffering=None, encoding=None, newline=None, suffix='', prefix='tmp', dir=None, delete=True)

This function operates exactly as TemporaryFile() does, except that the file is guaranteed to have a visible name in the file system (on Unix, the directory entry is not unlinked). That name can be retrieved from the name attribute of the file object. Whether the name can be used to open the file a second time, while the named temporary file is still open, varies across platforms (it can be so used on Unix; it cannot on Windows NT or later). If delete is true (the default), the file is deleted as soon as it is closed. The returned object is always a file-like object whose file attribute is the underlying true file object. This file-like object can be used in a with statement, just like a normal file.

tempfile.SpooledTemporaryFile (max_size=0, mode='w+b', buffering=None, encoding=None, newline=None, suffix='', prefix='tmp', dir=None)

This function operates exactly as TemporaryFile() does, except that data is spooled in memory until the file size exceeds max_size, or until the file’s fileno() method is called, at which point the contents are written to disk and operation proceeds as with TemporaryFile().

The resulting file has one additional method, rollover(), which causes the file to roll over to an on-disk file regardless of its size.

The returned object is a file-like object whose _file attribute is either a io.BytesIO or io.StringIO object (depending on whether binary or text mode was specified) or a true file object, depending on whether rollover() has been called. This file-like object can be used in a with statement, just like a normal file. Changed in version 3.3: the truncate method now accepts a size argument.
**tempfile.**

**TemporaryDirectory**(suffix='', prefix='tmp', dir=None)

This function creates a temporary directory using `mkdtemp()` (the supplied arguments are passed directly to the underlying function). The resulting object can be used as a context manager (see context-managers). On completion of the context (or destruction of the temporary directory object), the newly created temporary directory and all its contents are removed from the filesystem.

The directory name can be retrieved from the `name` attribute of the returned object.

The directory can be explicitly cleaned up by calling the `cleanup()` method. New in version 3.2.

**tempfile.**

**mkstemp**(suffix='', prefix='tmp', dir=None, text=False)

Creates a temporary file in the most secure manner possible. There are no race conditions in the file’s creation, assuming that the platform properly implements the `os.O_EXCL` flag for `os.open()`. The file is readable and writable only by the creating user ID. If the platform uses permission bits to indicate whether a file is executable, the file is executable by no one. The file descriptor is not inherited by child processes.

Unlike `TemporaryFile()`, the user of `mkstemp()` is responsible for deleting the temporary file when done with it.

If `suffix` is specified, the file name will end with that suffix, otherwise there will be no suffix. `mkstemp()` does not put a dot between the file name and the suffix; if you need one, put it at the beginning of `suffix`.

If `prefix` is specified, the file name will begin with that prefix; otherwise, a default prefix is used.

If `dir` is specified, the file will be created in that directory; otherwise, a default directory is used. The default directory is chosen from a platform-dependent list, but the user of the application can control the directory location by setting the `TMPDIR`, `TEMP` or `TMP` environment variables. There is thus no guarantee that the generated filename will have any nice properties, such as not requiring quoting when passed to external commands via `os.popen()`.

If `text` is specified, it indicates whether to open the file in binary mode (the default) or text mode. On some platforms, this makes no difference.

`mkstemp()` returns a tuple containing an OS-level handle to an open file (as would be returned by `os.open()`) and the absolute pathname of that file, in that order.

**tempfile.**

**mkdtemp**(suffix='', prefix='tmp', dir=None)

Creates a temporary directory in the most secure manner possible. There are no race conditions in the directory’s creation. The directory is readable, writable, and searchable only by the creating user ID.

The user of `mkdtemp()` is responsible for deleting the temporary directory and its contents when done with it.

The `prefix`, `suffix`, and `dir` arguments are the same as for `mkstemp()`.

`mkdtemp()` returns the absolute pathname of the new directory.

**tempfile.**

**mktemp**(suffix='', prefix='tmp', dir=None)

Deprecated since version 2.3: Use `mkstemp()` instead. Return an absolute pathname of a file that did not exist at the time the call is made. The `prefix`, `suffix`, and `dir` arguments are the same as for `mkstemp()`.

**Warning:** Use of this function may introduce a security hole in your program. By the time you get around to doing anything with the file name it returns, someone else may have beaten you to the punch. `mktemp()` usage can be replaced easily with `NamedTemporaryFile()`, passing it the `delete=False` parameter:

```python
>>> f = NamedTemporaryFile(delete=False)
>>> f.name
'/tmp/tmptujujtt'
>>> f.write(b"Hello World!\n")
13
>>> f.close()
>>> os.unlink(f.name)
>>> os.path.exists(f.name)
False
```
The module uses two global variables that tell it how to construct a temporary name. They are initialized at the first call to any of the functions above. The caller may change them, but this is discouraged; use the appropriate function arguments, instead.

**tempfile.tempdir**

When set to a value other than None, this variable defines the default value for the *dir* argument to all the functions defined in this module.

If `tempdir` is unset or `None` at any call to any of the above functions, Python searches a standard list of directories and sets `tempdir` to the first one which the calling user can create files in. The list is:

1. The directory named by the `TMPDIR` environment variable.
2. The directory named by the `TEMP` environment variable.
3. The directory named by the `TMP` environment variable.
4. A platform-specific location:
   - On Windows, the directories `C:\TEMP`, `C:\TMP`, `\TEMP`, and `\TMP`, in that order.
   - On all other platforms, the directories `/tmp`, `/var/tmp`, and `/usr/tmp`, in that order.
5. As a last resort, the current working directory.

**tempfile.gettempdir()**

Return the directory currently selected to create temporary files in. If `tempdir` is not `None`, this simply returns its contents; otherwise, the search described above is performed, and the result returned.

**tempfile.gettempprefix()**

Return the filename prefix used to create temporary files. This does not contain the directory component.

### 11.5.1 Examples

Here are some examples of typical usage of the `tempfile` module:

```python
>>> import tempfile

# create a temporary file and write some data to it
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# read data from file
>>> fp.seek(0)
>>> fp.read()
b'Hello world!
# close the file, it will be removed
>>> fp.close()

# create a temporary file using a context manager
>>> with tempfile.TemporaryFile() as fp:
...    fp.write(b'Hello world!')
...    fp.seek(0)
...    fp.read()
b'Hello world!

# file is now closed and removed

# create a temporary directory using the context manager
>>> with tempfile.TemporaryDirectory() as tmpdirname:
...    print('created temporary directory', tmpdirname)

# directory and contents have been removed
```
11.6  glob — Unix style pathname pattern expansion

Source code: Lib/glob.py

The glob module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell. No tilde expansion is done, but *, ?, and character ranges expressed with [] will be correctly matched. This is done by using the os.listdir() and fnmatch.fnmatch() functions in concert, and not by actually invoking a subshell. Note that unlike fnmatch.fnmatch(), glob treats filenames beginning with a dot (.) as special cases. (For tilde and shell variable expansion, use os.path.expanduser() and os.path.expandvars().)

For a literal match, wrap the meta-characters in brackets. For example, ’[?]’ matches the character ‘?’.

glob.glob(pathname)

Return a possibly-empty list of path names that match pathname, which must be a string containing a path specification. pathname can be either absolute (like /usr/src/Python-1.5/Makefile) or relative (like ../../Tools/*//*.gif), and can contain shell-style wildcards. Broken symlinks are included in the results (as in the shell).

glob.iglob(pathname)

Return an iterator which yields the same values as glob() without actually storing them all simultaneously.

For example, consider a directory containing only the following files: 1.gif, 2.txt, and card.gif. glob() will produce the following results. Notice how any leading components of the path are preserved.

>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']

If the directory contains files starting with . they won’t be matched by default. For example, consider a directory containing card.gif and .card.gif:

>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']

See Also:
Module fnmatch. Shell-style filename (not path) expansion

11.7  fnmatch — Unix filename pattern matching

Source code: Lib/fnmatch.py

This module provides support for Unix shell-style wildcards, which are not the same as regular expressions (which are documented in the re module). The special characters used in shell-style wildcards are:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>matches everything</td>
</tr>
<tr>
<td>?</td>
<td>matches any single character</td>
</tr>
<tr>
<td>[seq]</td>
<td>matches any character in seq</td>
</tr>
<tr>
<td>![seq]</td>
<td>matches any character not in seq</td>
</tr>
</tbody>
</table>

11.6.  glob — Unix style pathname pattern expansion
For a literal match, wrap the meta-characters in brackets. For example, ’[?]’ matches the character ’?’.

Note that the filename separator (’/’ on Unix) is not special to this module. See module glob for pathname expansion (glob uses fnmatch() to match pathname segments). Similarly, filenames starting with a period are not special for this module, and are matched by the * and ? patterns.

\texttt{fnmatch.fnmatch(filename, pattern)}

Test whether the filename string matches the pattern string, returning True or False. If the operating system is case-insensitive, then both parameters will be normalized to all lower- or upper-case before the comparison is performed. \texttt{fnmatchcase()} can be used to perform a case-sensitive comparison, regardless of whether that’s standard for the operating system.

This example will print all file names in the current directory with the extension .txt:

\begin{verbatim}
import fnmatch
import os

for file in os.listdir('.'):  
    if fnmatch.fnmatch(file, '*.txt'):  
        print(file)
\end{verbatim}

\texttt{fnmatch.fnmatchcase(filename, pattern)}

Test whether filename matches pattern, returning True or False; the comparison is case-sensitive.

\texttt{fnmatch.filter(names, pattern)}

Return the subset of the list of names that match pattern. It is the same as \texttt{[n for n in names if fnmatch(n, pattern)]}, but implemented more efficiently.

\texttt{fnmatch.translate(pattern)}

Return the shell-style pattern converted to a regular expression.

Example:

\begin{verbatim}
>>> import fnmatch, re

>>> regex = fnmatch.translate('*.txt')
>>> regex
'.*\.txt$'
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<_sre.SRE_Match object at 0x...>
\end{verbatim}

See Also:

Module glob Unix shell-style path expansion.

\section*{11.8 linecache — Random access to text lines}

Source code: Lib/linecache.py

The linecache module allows one to get any line from any file, while attempting to optimize internally, using a cache, the common case where many lines are read from a single file. This is used by the traceback module to retrieve source lines for inclusion in the formatted traceback.

The linecache module defines the following functions:

\texttt{linecache.getline(filename, lineno, module\_globals=None)}

Get line lineno from file named filename. This function will never raise an exception — it will return ” on errors (the terminating newline character will be included for lines that are found).
If a file named `filename` is not found, the function will look for it in the module search path, `sys.path`, after first checking for a PEP 302 `__loader__` in `module_globals`, in case the module was imported from a zipfile or other non-filesystem import source.

```python
linecache.clearcache()
```

Clear the cache. Use this function if you no longer need lines from files previously read using `getline()`.

```python
linecache.checkcache(filename=None)
```

Check the cache for validity. Use this function if files in the cache may have changed on disk, and you require the updated version. If `filename` is omitted, it will check all the entries in the cache.

Example:

```python
>>> import linecache
>>> linecache.getline('/etc/passwd', 4)
'sys:x:3:3:/dev:/bin/sh
'```

### 11.9 `shutil` — High-level file operations

Source code: `Lib/shutil.py`

The `shutil` module offers a number of high-level operations on files and collections of files. In particular, functions are provided which support file copying and removal. For operations on individual files, see also the `os` module.

**Warning:** Even the higher-level file copying functions (`shutil.copy()`, `shutil.copy2()`) cannot copy all file metadata. On POSIX platforms, this means that file owner and group are lost as well as ACLs. On Mac OS, the resource fork and other metadata are not used. This means that resources will be lost and file type and creator codes will not be correct. On Windows, file owners, ACLs and alternate data streams are not copied.

#### 11.9.1 Directory and files operations

```python
shutil.copyfileobj(src, dst[, length])
```

Copy the contents of the file-like object `src` to the file-like object `dst`. The integer `length`, if given, is the buffer size. In particular, a negative `length` value means to copy the data without looping over the source data in chunks; by default the data is read in chunks to avoid uncontrolled memory consumption. Note that if the current file position of the `src` object is not 0, only the contents from the current file position to the end of the file will be copied.

```python
shutil.copyfile(src, dst, *, follow_symlinks=True)
```

Copy the contents (no metadata) of the file named `src` to a file named `dst` and return `dst`. `src` and `dst` are path names given as strings. `dst` must be the complete target file name; look at `shutil.copy()` for a copy that accepts a target directory path. If `src` and `dst` specify the same file, `Error` is raised.

The destination location must be writable; otherwise, an `OSError` exception will be raised. If `dst` already exists, it will be replaced. Special files such as character or block devices and pipes cannot be copied with this function.

If `follow_symlinks` is false and `src` is a symbolic link, a new symbolic link will be created instead of copying the file `src` points to. Changed in version 3.3: `IOError` used to be raised instead of `OSError`. Added `follow_symlinks` argument. Now returns `dst`.

```python
shutil.copymode(src, dst, *, follow_symlinks=True)
```

Copy the permission bits from `src` to `dst`. The file contents, owner, and group are unaffected. `src` and `dst` are path names given as strings. If `follow_symlinks` is false, and both `src` and `dst` are symbolic links, `copymode()` will attempt to modify the mode of `dst` itself (rather than the file it points to). This functionality is not available on every platform; please see `copystat()` for more information. If `copymode()`
cannot modify symbolic links on the local platform, and it is asked to do so, it will do nothing and return. Changed in version 3.3: Added follow_symlinks argument.

```
shutil.copystat(src, dst, *, follow_symlinks=True)
```

Copy the permission bits, last access time, last modification time, and flags from src to dst. On Linux, copystat() also copies the “extended attributes” where possible. The file contents, owner, and group are unaffected. src and dst are path names given as strings.

If follow_symlinks is false, and src and dst both refer to symbolic links, copystat() will operate on the symbolic links themselves rather than the files the symbolic links refer to—reading the information from the src symbolic link, and writing the information to the dst symbolic link.

**Note:** Not all platforms provide the ability to examine and modify symbolic links. Python itself can tell you what functionality is locally available.

- If os.chmod in os.supports_follow_symlinks is True, copystat() can modify the permission bits of a symbolic link.
- If os.utime in os.supports_follow_symlinks is True, copystat() can modify the last access and modification times of a symbolic link.
- If os.chflags in os.supports_follow_symlinks is True, copystat() can modify the flags of a symbolic link. (os.chflags is not available on all platforms.)

On platforms where some or all of this functionality is unavailable, when asked to modify a symbolic link, copystat() will copy everything it can. copystat() never returns failure.

Please see os.supports_follow_symlinks for more information.

Changed in version 3.3: Added follow_symlinks argument and support for Linux extended attributes.

```
shutil.copy(src, dst, *, follow_symlinks=True)
```

Copies the file src to the file or directory dst. src and dst should be strings. If dst specifies a directory, the file will be copied into dst using the base filename from src. Returns the path to the newly created file.

If follow_symlinks is false, and src is a symbolic link, dst will be created as a symbolic link. If follow_symlinks is true and src is a symbolic link, dst will be a copy of the file src refers to.

copy() copies the file data and the file’s permission mode (see os.chmod()). Other metadata, like the file’s creation and modification times, is not preserved. To preserve all file metadata from the original, use copy2() instead. Changed in version 3.3: Added follow_symlinks argument. Now returns path to the newly created file.

```
shutil.copy2(src, dst, *, follow_symlinks=True)
```

Identical to copy() except that copy2() also attempts to preserve all file metadata.

When follow_symlinks is false, and src is a symbolic link, copy2() attempts to copy all metadata from the src symbolic link to the newly-created dst symbolic link. However, this functionality is not available on all platforms. On platforms where some or all of this functionality is unavailable, copy2() will preserve all the metadata it can; copy2() never returns failure.

copy2() uses copystat() to copy the file metadata. Please see copystat() for more information about platform support for modifying symbolic link metadata. Changed in version 3.3: Added follow_symlinks argument, try to copy extended file system attributes too (currently Linux only). Now returns path to the newly created file.

```
shutil.ignore_patterns(*patterns)
```

This factory function creates a function that can be used as a callable for copytree()’s ignore argument, ignoring files and directories that match one of the glob-style patterns provided. See the example below.

```
shutil.copytree(src, dst, symlinks=False, ignore=None, copy_function=copy2, ignore_dangling_symlinks=False)
```

Recursively copy an entire directory tree rooted at src, returning the destination directory. The destination directory, named by dst, must not already exist; it will be created as well as missing parent directo-
ries. Permissions and times of directories are copied with `copystat()`, individual files are copied using `shutil.copy2()`.

If `symlinks` is true, symbolic links in the source tree are represented as symbolic links in the new tree and the metadata of the original links will be copied as far as the platform allows; if false or omitted, the contents and metadata of the linked files are copied to the new tree.

When `symlinks` is false, if the file pointed by the symlink doesn’t exist, a exception will be added in the list of errors raised in a `Error` exception at the end of the copy process. You can set the optional `ignore_dangling_symlinks` flag to true if you want to silence this exception. Notice that this option has no effect on platforms that don’t support `os.symlink()`.

If `ignore` is given, it must be a callable that will receive as its arguments the directory being visited by `copytree()`, and a list of its contents, as returned by `os.listdir()`. Since `copytree()` is called recursively, the `ignore` callable will be called once for each directory that is copied. The callable must return a sequence of directory and file names relative to the current directory (i.e. a subset of the items in its second argument); these names will then be ignored in the copy process. `ignore_patterns()` can be used to create such a callable that ignores names based on glob-style patterns.

If exception(s) occur, an `Error` is raised with a list of reasons.

If `copy_function` is given, it must be a callable that will be used to copy each file. It will be called with the source path and the destination path as arguments. By default, `shutil.copy2()` is used, but any function that supports the same signature (like `shutil.copy()`) can be used. Changed in version 3.3: Copy metadata when `symlinks` is false. Now returns `dst`. Changed in version 3.2: Added the `copy_function` argument to be able to provide a custom copy function. Added the `ignore_dangling_symlinks` argument to silent dangling symlinks errors when `symlinks` is false.

`shutil.rmtree(path, ignore_errors=False, onerror=None)`

Delete an entire directory tree; `path` must point to a directory (but not a symbolic link to a directory). If `ignore_errors` is true, errors resulting from failed removals will be ignored; if false or omitted, such errors are handled by calling a handler specified by `onerror` or, if that is omitted, they raise an exception.

Note: On platforms that support the necessary fd-based functions a symlink attack resistant version of `rmtree()` is used by default. On other platforms, the `rmtree()` implementation is susceptible to a symlink attack: given proper timing and circumstances, attackers can manipulate symlinks on the filesystem to delete files they wouldn’t be able to access otherwise. Applications can use the `rmtree.avoids_symlink_attacks` function attribute to determine which case applies.

If `onerror` is provided, it must be a callable that accepts three parameters: `function`, `path`, and `excinfo`.

The first parameter, `function`, is the function which raised the exception; it depends on the platform and implementation. The second parameter, `path`, will be the path name passed to `function`. The third parameter, `excinfo`, will be the exception information returned by `sys.exc_info()`. Exceptions raised by `onerror` will not be caught. Changed in version 3.3: Added a symlink attack resistant version that is used automatically if platform supports fd-based functions.

`rmtree.avoids_symlink_attacks`

Indicates whether the current platform and implementation provides a symlink attack resistant version of `rmtree()`. Currently this is only true for platforms supporting fd-based directory access functions. New in version 3.3.

`shutil.move(src, dst)`

Recursively move a file or directory (`src`) to another location (`dst`) and return the destination.

If the destination is a directory or a symlink to a directory, then `src` is moved inside that directory.

The destination directory must not already exist. If the destination already exists but is not a directory, it may be overwritten depending on `os.rename()` semantics.

If the destination is on the current filesystem, then `os.rename()` is used. Otherwise, `src` is copied (using `shutil.copy2()`) to `dst` and then removed. In case of symlinks, a new symlink pointing to the target of
src will be created in or as dst and src will be removed. Changed in version 3.3: Added explicit symlink handling for foreign filesystems, thus adapting it to the behavior of GNU’s mv. Now returns dst.

```
shutil.disk_usage(path)
Return disk usage statistics about the given path as a named tuple with the attributes total, used and free, which are the amount of total, used and free space, in bytes. New in version 3.3. Availability: Unix, Windows.
```

```
shutil.chown(path, user=None, group=None)
Change owner user and/or group of the given path.
user can be a system user name or a uid; the same applies to group. At least one argument is required.
See also os.chown(), the underlying function.
Availability: Unix. New in version 3.3.
```

```
shutil.which(cmd, mode=os.F_OK | os.X_OK, path=None)
Return the path to an executable which would be run if the given cmd was called. If no cmd would be called, return None.
mode is a permission mask passed a to os.access(), by default determining if the file exists and executable.
When no path is specified, the results of os.environ() are used, returning either the “PATH” value or a fallback of os.defpath.
On Windows, the current directory is always prepended to the path whether or not you use the default or provide your own, which is the behavior the command shell uses when finding executables. Additionally, when finding the cmd in the path, the PATHEXT environment variable is checked. For example, if you call shutil.which("python"), which() will search PATHEXT to know that it should look for python.exe within the path directories. For example, on Windows:
```
>>> shutil.which("python")
C:\\Python33\\python.EXE
```
New in version 3.3.

```
exception shutil.Error
This exception collects exceptions that are raised during a multi-file operation. For copytree(), the exception argument is a list of 3-tuples (srcname, dstname, exception).
```

copytree example

This example is the implementation of the copytree() function, described above, with the docstring omitted. It demonstrates many of the other functions provided by this module.

```
def copytree(src, dst, symlinks=False):
    names = os.listdir(src)
    os.makedirs(dst)
    errors = []
    for name in names:
        srcname = os.path.join(src, name)
        dstname = os.path.join(dst, name)
        try:
            if symlinks and os.path.islink(srcname):
                linkto = os.readlink(srcname)
                os.symlink(linkto, dstname)
            elif os.path.isdir(srcname):
                copytree(srcname, dstname, symlinks)
            else:
                copy2(srcname, dstname)
        # XXX What about devices, sockets etc.?
```
```python
def _logpath(path, names):
    logging.info('Working in %s' % path)
    return []  # nothing will be ignored
```

### 11.9.2 Archiving operations

New in version 3.2. High-level utilities to create and read compressed and archived files are also provided. They rely on the `zipfile` and `tarfile` modules.

```python
shutil.make_archive(base_name, format[, root_dir[, base_dir[, verbose[, dry_run[, owner[, group[, logger]]]]]]])
```

Create an archive file (such as zip or tar) and return its name.

- `base_name` is the name of the file to create, including the path, minus any format-specific extension. `format` is the archive format: one of “zip”, “tar”, “bztar” (if the `bz2` module is available) or “gztar”.
- `root_dir` is a directory that will be the root directory of the archive; for example, we typically chdir into `root_dir` before creating the archive.
- `base_dir` is the directory where we start archiving from; i.e. `base_dir` will be the common prefix of all files and directories in the archive.
- `root_dir` and `base_dir` both default to the current directory.
- `owner` and `group` are used when creating a tar archive. By default, uses the current owner and group.
- `logger` must be an object compatible with PEP 282, usually an instance of `logging.Logger`.

```python
shutil.get_archive_formats()
```

Return a list of supported formats for archiving. Each element of the returned sequence is a tuple `(name, description)`.

By default `shutil` provides these formats:
shutil.register_archive_format(name, function[, extra_args[, description]])
Register an archiver for the format name. function is a callable that will be used to invoke the archiver.
If given, extra_args is a sequence of (name, value) pairs that will be used as extra keywords arguments when the archiver callable is used.
description is used by get_archive_formats() which returns the list of archivers. Defaults to an empty list.

shutil.unregister_archive_format(name)
Remove the archive format name from the list of supported formats.

shutil.unpack_archive(filename[, extract_dir[, format]])
Unpack an archive. filename is the full path of the archive.
extract_dir is the name of the target directory where the archive is unpacked. If not provided, the current working directory is used.
format is the archive format: one of “zip”, “tar”, or “gztar”. Or any other format registered with register_unpack_format(). If not provided, unpack_archive() will use the archive file name extension and see if an unpacker was registered for that extension. In case none is found, a ValueError is raised.

shutil.register_unpack_format(name, extensions, function[, extra_args[, description]])
Registers an unpack format. name is the name of the format and extensions is a list of extensions corresponding to the format, like .zip for Zip files.
function is the callable that will be used to unpack archives. The callable will receive the path of the archive, followed by the directory the archive must be extracted to.
When provided, extra_args is a sequence of (name, value) tuples that will be passed as keywords arguments to the callable.
description can be provided to describe the format, and will be returned by the get_unpack_formats() function.

shutil.unregister_unpack_format(name)
Unregister an unpack format. name is the name of the format.

shutil.get_unpack_formats()
Return a list of all registered formats for unpacking. Each element of the returned sequence is a tuple (name, extensions, description).

By default shutil provides these formats:
• *gztar: gzip’ed tar-file
• *bztar: bzip2’ed tar-file (if the bz2 module is available.)
• *tar: uncompressed tar file
• *zip: ZIP file

You can register new formats or provide your own unpacker for any existing formats, by using register_unpack_format().

shutil.register_unpack_format(name, extensions, function[, extra_args[, description]])
Registers an unpack format. name is the name of the format and extensions is a list of extensions corresponding to the format, like .zip for Zip files.
function is the callable that will be used to unpack archives. The callable will receive the path of the archive, followed by the directory the archive must be extracted to.
When provided, extra_args is a sequence of (name, value) tuples that will be passed as keywords arguments to the callable.
description can be provided to describe the format, and will be returned by the get_unpack_formats() function.

shutil.unregister_unpack_format(name)
Unregister an unpack format. name is the name of the format.

shutil.get_unpack_formats()
Return a list of all registered formats for unpacking. Each element of the returned sequence is a tuple (name, extensions, description).

By default shutil provides these formats:
• *gztar: gzip’ed tar-file
• *bztar: bzip2’ed tar-file (if the bz2 module is available.)
• *tar: uncompressed tar file
• *zip: ZIP file

You can register new formats or provide your own unpacker for any existing formats, by using register_unpack_format().

You can register new formats or provide your own archiver for any existing formats, by using register_archive_format().

shutil.register_archive_format(name, function[, extra_args[, description]])
Register an archiver for the format name. function is a callable that will be used to invoke the archiver.
If given, extra_args is a sequence of (name, value) pairs that will be used as extra keywords arguments when the archiver callable is used.
description is used by get_archive_formats() which returns the list of archivers. Defaults to an empty list.

shutil.unregister_archive_format(name)
Remove the archive format name from the list of supported formats.

shutil.get_archive_formats()
Return a list of all registered formats for archiving. Each element of the returned sequence is a tuple (name, format, description).

By default shutil provides these formats:
• *gztar: gzip’ed tar-file
• *bztar: bzip2’ed tar-file (if the bz2 module is available.)
• *tar: uncompressed tar file
• *zip: ZIP file

You can register new formats or provide your own archiver for any existing formats, by using register_archive_format().
Archiving example

In this example, we create a gzip’ed tar-file archive containing all files found in the .ssh directory of the user:

```python
>>> from shutil import make_archive
>>> import os

archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

The resulting archive contains:

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
```

11.9.3 Querying the size of the output terminal

New in version 3.3.

```python
shutil.get_terminal_size(fallback=(columns, lines))
```

Get the size of the terminal window.

For each of the two dimensions, the environment variable, COLUMNS and LINES respectively, is checked. If the variable is defined and the value is a positive integer, it is used.

When COLUMNS or LINES is not defined, which is the common case, the terminal connected to `sys.__stdout__` is queried by invoking `os.get_terminal_size()`.

If the terminal size cannot be successfully queried, either because the system doesn’t support querying, or because we are not connected to a terminal, the value given in `fallback` parameter is used. `fallback` defaults to (80, 24) which is the default size used by many terminal emulators.

The value returned is a named tuple of type `os.terminal_size`.

See also: The Single UNIX Specification, Version 2, Other Environment Variables.

11.10 `macpath` — Mac OS 9 path manipulation functions

This module is the Mac OS 9 (and earlier) implementation of the `os.path` module. It can be used to manipulate old-style Macintosh pathnames on Mac OS X (or any other platform).

The following functions are available in this module: `normcase()`, `normpath()`, `isabs()`, `join()`, `split()`, `isdir()`, `isfile()`, `walk()`, `exists()`. For other functions available in `os.path` dummy counterparts are available.

See Also:

Module `os` Operating system interfaces, including functions to work with files at a lower level than Python file objects.

Module `io` Python’s built-in I/O library, including both abstract classes and some concrete classes such as file I/O.

Built-in function `open()` The standard way to open files for reading and writing with Python.
The modules described in this chapter support storing Python data in a persistent form on disk. The `pickle` and `marshal` modules can turn many Python data types into a stream of bytes and then recreate the objects from the bytes. The various DBM-related modules support a family of hash-based file formats that store a mapping of strings to other strings.

The list of modules described in this chapter is:

### 12.1 pickle — Python object serialization

The `pickle` module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. “Pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation, whereby a byte stream is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as “serialization”, “marshalling,” ¹ or “flattening”, however, to avoid confusion, the terms used here are “pickling” and “unpickling”.

**Warning:** The `pickle` module is not intended to be secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.

#### 12.1.1 Relationship to other Python modules

The `pickle` module has an transparent optimizer (`_pickle`) written in C. It is used whenever available. Otherwise the pure Python implementation is used.

Python has a more primitive serialization module called `marshal`, but in general `pickle` should always be the preferred way to serialize Python objects. `marshal` exists primarily to support Python’s `.pyc` files.

The `pickle` module differs from `marshal` in several significant ways:

- The `pickle` module keeps track of the objects it has already serialized, so that later references to the same object won’t be serialized again. `marshal` doesn’t do this.

  This has implications both for recursive objects and object sharing. Recursive objects are objects that contain references to themselves. These are not handled by marshal, and in fact, attempting to marshal recursive objects will crash your Python interpreter. Object sharing happens when there are multiple references to the same object in different places in the object hierarchy being serialized. `pickle` stores such objects only once, and ensures that all other references point to the master copy. Shared objects remain shared, which can be very important for mutable objects.

- `marshal` cannot be used to serialize user-defined classes and their instances. `pickle` can save and restore class instances transparently, however the class definition must be importable and live in the same module as when the object was stored.

¹ Don’t confuse this with the `marshal` module
• The marshal serialization format is not guaranteed to be portable across Python versions. Because its primary job in life is to support .pyc files, the Python implementers reserve the right to change the serialization format in non-backwards compatible ways should the need arise. The pickle serialization format is guaranteed to be backwards compatible across Python releases.

Note that serialization is a more primitive notion than persistence; although pickle reads and writes file objects, it does not handle the issue of naming persistent objects, nor the (even more complicated) issue of concurrent access to persistent objects. The pickle module can transform a complex object into a byte stream and it can transform the byte stream into an object with the same internal structure. Perhaps the most obvious thing to do with these byte streams is to write them onto a file, but it is also conceivable to send them across a network or store them in a database. The module shelve provides a simple interface to pickle and unpickle objects on DBM-style database files.

12.1.2 Data stream format

The data format used by pickle is Python-specific. This has the advantage that there are no restrictions imposed by external standards such as JSON or XDR (which can’t represent pointer sharing); however it means that non-Python programs may not be able to reconstruct pickled Python objects.

By default, the pickle data format uses a relatively compact binary representation. If you need optimal size characteristics, you can efficiently compress pickled data.

The module pickletools contains tools for analyzing data streams generated by pickle. pickletools source code has extensive comments about opcodes used by pickle protocols.

There are currently 4 different protocols which can be used for pickling.

• Protocol version 0 is the original “human-readable” protocol and is backwards compatible with earlier versions of Python.
• Protocol version 1 is an old binary format which is also compatible with earlier versions of Python.
• Protocol version 2 was introduced in Python 2.3. It provides much more efficient pickling of new-style classes. Refer to PEP 307 for information about improvements brought by protocol 2.
• Protocol version 3 was added in Python 3. It has explicit support for bytes objects and cannot be unpickled by Python 2.x. This is the default as well as the current recommended protocol; use it whenever possible.

12.1.3 Module Interface

To serialize an object hierarchy, you simply call the dumps() function. Similarly, to de-serialize a data stream, you call the loads() function. However, if you want more control over serialization and de-serialization, you can create a Pickler or an Unpickler object, respectively.

The pickle module provides the following constants:

pickle.HIGHEST_PROTOCOL
The highest protocol version available. This value can be passed as a protocol value.

pickle.DEFAULT_PROTOCOL
The default protocol used for pickling. May be less than HIGHEST_PROTOCOL. Currently the default protocol is 3, a new protocol designed for Python 3.0.

The pickle module provides the following functions to make the pickling process more convenient:

pickle.dump (obj, file, protocol=None, *, fix_imports=True)
Write a pickled representation of obj to the open file object file. This is equivalent to Pickler(file, protocol).dump(obj).

The optional protocol argument tells the pickler to use the given protocol; supported protocols are 0, 1, 2, 3. The default protocol is 3; a backward-incompatible protocol designed for Python 3.0.

Specifying a negative protocol version selects the highest protocol version supported. The higher the protocol used, the more recent the version of Python needed to read the pickle produced.
The file argument must have a write() method that accepts a single bytes argument. It can thus be an on-disk file opened for binary writing, a `io.BytesIO` instance, or any other custom object that meets this interface.

If fix_imports is True and protocol is less than 3, pickle will try to map the new Python 3.x names to the old module names used in Python 2.x, so that the pickle data stream is readable with Python 2.x.

```python
def dumps(obj, protocol=None, *, fix_imports=True)
```  
Return the pickled representation of the object as a bytes object, instead of writing it to a file.

The optional protocol argument tells the pickler to use the given protocol; supported protocols are 0, 1, 2, 3. The default protocol is 3; a backward-incompatible protocol designed for Python 3.0.

Specifying a negative protocol version selects the highest protocol version supported. The higher the protocol used, the more recent the version of Python needed to read the pickle produced.

If fix_imports is True and protocol is less than 3, pickle will try to map the new Python 3.x names to the old module names used in Python 2.x, so that the pickle data stream is readable with Python 2.x.

```python
def load(file, *, fix_imports=True, encoding="ASCII", errors="strict")
```  
Read a pickled object representation from the open file object file and return the reconstituted object hierarchy specified therein. This is equivalent to `Unpickler(file).load()`.

The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled object’s representation are ignored.

The argument file must have two methods, a read() method that takes an integer argument, and a readline() method that requires no arguments. Both methods should return bytes. Thus file can be an on-disk file opened for binary reading, a `io.BytesIO` object, or any other custom object that meets this interface.

Optional keyword arguments are fix_imports, encoding and errors, which are used to control compatibility support for pickle stream generated by Python 2.x. If fix_imports is True, pickle will try to map the old Python 2.x names to the new names used in Python 3.x. The encoding and errors tell pickle how to decode 8-bit string instances pickled by Python 2.x; these default to ‘ASCII’ and ‘strict’, respectively.

```python
def loads(bytes_object, *, fix_imports=True, encoding="ASCII", errors="strict")
```  
Read a pickled object hierarchy from a bytes object and return the reconstituted object hierarchy specified therein.

The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled object’s representation are ignored.

Optional keyword arguments are fix_imports, encoding and errors, which are used to control compatibility support for pickle stream generated by Python 2.x. If fix_imports is True, pickle will try to map the old Python 2.x names to the new names used in Python 3.x. The encoding and errors tell pickle how to decode 8-bit string instances pickled by Python 2.x; these default to ‘ASCII’ and ‘strict’, respectively.

The pickle module defines three exceptions:

1. **Exception pickle.PickleError**

   Common base class for the other pickling exceptions. It inherits Exception.

2. **Exception pickle.PicklingError**

   Error raised when an unpicklable object is encountered by Pickler. It inherits PickleError.

   Refer to `What can be pickled and unpickled?` to learn what kinds of objects can be pickled.

3. **Exception pickle.UnpicklingError**

   Error raised when there is a problem unpickling an object, such as a data corruption or a security violation. It inherits PickleError.

   Note that other exceptions may also be raised during unpickling, including (but not necessarily limited to) AttributeError, EOFError, ImportError, and IndexError.

The pickle module exports two classes, Pickler and Unpickler:

```python
class Pickler(file, protocol=None, *, fix_imports=True)
```  
This takes a binary file for writing a pickle data stream.
The optional protocol argument tells the pickler to use the given protocol; supported protocols are 0, 1, 2, 3. The default protocol is 3; a backward-incompatible protocol designed for Python 3.0.

Specifying a negative protocol version selects the highest protocol version supported. The higher the protocol used, the more recent the version of Python needed to read the pickle produced.

The file argument must have a write() method that accepts a single bytes argument. It can thus be an on-disk file opened for binary writing, a io.BytesIO instance, or any other custom object that meets this interface.

If fix_imports is True and protocol is less than 3, pickle will try to map the new Python 3.x names to the old module names used in Python 2.x, so that the pickle data stream is readable with Python 2.x.

dump(obj)
Write a pickled representation of obj to the open file object given in the constructor.

persistent_id(obj)
Do nothing by default. This exists so a subclass can override it.

If persistent_id() returns None, obj is pickled as usual. Any other value causes Pickler to emit the returned value as a persistent ID for obj. The meaning of this persistent ID should be defined by Unpickler.persistent_load(). Note that the value returned by persistent_id() cannot itself have a persistent ID.

See Persistence of External Objects for details and examples of uses.

dispatch_table
A pickler object’s dispatch table is a registry of reduction functions of the kind which can be declared using copyreg.pickle(). It is a mapping whose keys are classes and whose values are reduction functions. A reduction function takes a single argument of the associated class and should conform to the same interface as a __reduce__() method.

By default, a pickler object will not have a dispatch_table attribute, and it will instead use the global dispatch table managed by the copyreg module. However, to customize the pickling for a specific pickler object one can set the dispatch_table attribute to a dict-like object. Alternatively, if a subclass of Pickler has a dispatch_table attribute then this will be used as the default dispatch table for instances of that class.

See Dispatch Tables for usage examples. New in version 3.3.

fast
Deprecated. Enable fast mode if set to a true value. The fast mode disables the usage of memo, therefore speeding the pickling process by not generating superfluous PUT opcodes. It should not be used with self-referential objects, doing otherwise will cause Pickler to recurse infinitely.

Use pickletools.optimize() if you need more compact pickles.

class Unpickler(file, *, fix_imports=True, encoding="ASCII", errors="strict")
This takes a binary file for reading a pickle data stream.

The protocol version of the pickle is detected automatically, so no protocol argument is needed.

The argument file must have two methods, a read() method that takes an integer argument, and a readline() method that requires no arguments. Both methods should return bytes. Thus file can be an on-disk file object opened for binary reading, a io.BytesIO object, or any other custom object that meets this interface.

Optional keyword arguments are fix_imports, encoding and errors, which are used to control compatibility support for pickle stream generated by Python 2.x. If fix_imports is True, pickle will try to map the old Python 2.x names to the new names used in Python 3.x. The encoding and errors tell pickle how to decode 8-bit string instances pickled by Python 2.x; these default to ‘ASCII’ and ‘strict’, respectively.

load()
Read a pickled object representation from the open file object given in the constructor, and return the reconstituted object hierarchy specified therein. Bytes past the pickled object’s representation are ignored.
**persistent_load** *(pid)*

Raise an **UnpicklingError** by default.

If defined, **persistent_load()** should return the object specified by the persistent ID *pid*. If an invalid persistent ID is encountered, an **UnpicklingError** should be raised.

See *Persistence of External Objects* for details and examples of uses.

**find_class**(module, name)

Import *module* if necessary and return the object called *name* from it, where the *module* and *name* arguments are **str** objects. Note, unlike its name suggests, **find_class()** is also used for finding functions.

Subclasses may override this to gain control over what type of objects and how they can be loaded, potentially reducing security risks. Refer to *Restricting Globals* for details.

### 12.1.4 What can be pickled and unpickled?

The following types can be pickled:

- None, True, and False
- integers, floating point numbers, complex numbers
- strings, bytes, bytearrays
- tuples, lists, sets, and dictionaries containing only picklable objects
- functions defined at the top level of a module
- built-in functions defined at the top level of a module
- classes that are defined at the top level of a module
- instances of such classes whose **dict** or the result of calling **getstate()** is picklable (see section *Pickling Class Instances* for details).

Attempts to pickle unpicklable objects will raise the **PicklingError** exception; when this happens, an unspecified number of bytes may have already been written to the underlying file. Trying to pickle a highly recursive data structure may exceed the maximum recursion depth, a **RuntimeError** will be raised in this case. You can carefully raise this limit with **sys.setrecursionlimit()**.

Note that functions (built-in and user-defined) are pickled by “fully qualified” name reference, not by value. This means that only the function name is pickled, along with the name of the module the function is defined in. Neither the function’s code, nor any of its function attributes are pickled. Thus the defining module must be importable in the unpickling environment, and the module must contain the named object, otherwise an exception will be raised.

Similarly, classes are pickled by named reference, so the same restrictions in the unpickling environment apply. Note that none of the class’s code or data is pickled, so in the following example the class attribute *attr* is not restored in the unpickling environment:

```python
class Foo:
    attr = ‘A class attribute’
```

picklestring = pickle.dumps(Foo)

These restrictions are why picklable functions and classes must be defined in the top level of a module.

Similarly, when class instances are pickled, their class’s code and data are not pickled along with them. Only the instance data are pickled. This is done on purpose, so you can fix bugs in a class or add methods to the class and still load objects that were created with an earlier version of the class. If you plan to have long-lived objects that will see many versions of a class, it may be worthwhile to put a version number in the objects so that suitable conversions can be made by the class’s **setstate()** method.

---

---

1. The exception raised will likely be an **ImportError** or an **AttributeError** but it could be something else.
12.1.5 Pickling Class Instances

In this section, we describe the general mechanisms available to you to define, customize, and control how class instances are pickled and unpickled.

In most cases, no additional code is needed to make instances picklable. By default, pickle will retrieve the class and the attributes of an instance via introspection. When a class instance is unpickled, its __init__() method is usually not invoked. The default behaviour first creates an uninitialized instance and then restores the saved attributes. The following code shows an implementation of this behaviour:

```python
def save(obj):
    return (obj.__class__, obj.__dict__)

def load(cls, attributes):
    obj = cls.__new__(cls)
    obj.__dict__.update(attributes)
    return obj
```

Classes can alter the default behaviour by providing one or several special methods:

```python
object.__getnewargs__()
```

In protocol 2 and newer, classes that implements the __getnewargs__() method can dictate the values passed to the __new__() method upon unpickling. This is often needed for classes whose __new__() method requires arguments.

```python
object.__getstate__()
```

Classes can further influence how their instances are pickled; if the class defines the method __getstate__(), it is called and the returned object is pickled as the contents for the instance, instead of the contents of the instance’s dictionary. If the __getstate__() method is absent, the instance’s __dict__ is pickled as usual.

```python
object.__setstate__(state)
```

Upon unpickling, if the class defines __setstate__(), it is called with the unpickled state. In that case, there is no requirement for the state object to be a dictionary. Otherwise, the pickled state must be a dictionary and its items are assigned to the new instance’s dictionary.

**Note:** If __getstate__() returns a false value, the __setstate__() method will not be called upon unpickling.

Refer to the section Handling Stateful Objects for more information about how to use the methods __getstate__() and __setstate__().

**Note:** At unpickling time, some methods like __getattr__() or __setattr__() may be called upon the instance. In case those methods rely on some internal invariant being true, the type should implement __getnewargs__() to establish such an invariant; otherwise, neither __new__() nor __init__() will be called.

As we shall see, pickle does not use directly the methods described above. In fact, these methods are part of the copy protocol which implements the __reduce__() special method. The copy protocol provides a unified interface for retrieving the data necessary for pickling and copying objects.\(^3\)

Although powerful, implementing __reduce__() directly in your classes is error prone. For this reason, class designers should use the high-level interface (i.e., __getnewargs__(), __getstate__() and __setstate__()) whenever possible. We will show, however, cases where using __reduce__() is the only option or leads to more efficient pickling or both.

```python
object.__reduce__()
```

The interface is currently defined as follows. The __reduce__() method takes no argument and shall return either a string or preferably a tuple (the returned object is often referred to as the “reduce value”).

---

\(^3\) The copy module uses this protocol for shallow and deep copying operations.
If a string is returned, the string should be interpreted as the name of a global variable. It should be the object’s local name relative to its module; the pickle module searches the module namespace to determine the object’s module. This behaviour is typically useful for singletons.

When a tuple is returned, it must be between two and five items long. Optional items can either be omitted, or `None` can be provided as their value. The semantics of each item are in order:

- A callable object that will be called to create the initial version of the object.
- A tuple of arguments for the callable object. An empty tuple must be given if the callable does not accept any argument.
- Optionally, the object’s state, which will be passed to the object’s `__setstate__` method as previously described. If the object has no such method then, the value must be a dictionary and it will be added to the object’s `__dict__` attribute.
- Optionally, an iterator (and not a sequence) yielding successive items. These items will be appended to the object either using `obj.append(item)` or, in batch, using `obj.extend(list_of_items)`. This is primarily used for list subclasses, but may be used by other classes as long as they have `append()` and `extend()` methods with the appropriate signature. (Whether `append()` or `extend()` is used depends on which pickle protocol version is used as well as the number of items to append, so both must be supported.)
- Optionally, an iterator (not a sequence) yielding successive key-value pairs. These items will be stored to the object using `obj[key] = value`. This is primarily used for dictionary subclasses, but may be used by other classes as long as they implement `__setitem__()`.

```python
object.__reduce_ex__(protocol)
```

Alternatively, a `__reduce_ex__()` method may be defined. The only difference is this method should take a single integer argument, the protocol version. When defined, pickle will prefer it over the `__reduce__()` method. In addition, `__reduce__()` automatically becomes a synonym for the extended version. The main use for this method is to provide backwards-compatible reduce values for older Python releases.

### Persistence of External Objects

For the benefit of object persistence, the `pickle` module supports the notion of a reference to an object outside the pickled data stream. Such objects are referenced by a persistent ID, which should be either a string of alphanumeric characters (for protocol 0) \(^4\) or just an arbitrary object (for any newer protocol).

The resolution of such persistent IDs is not defined by the `pickle` module; it will delegate this resolution to the user-defined methods on the pickler and unpickler, `persistent_id()` and `persistent_load()` respectively.

To pickle objects that have an external persistent id, the pickler must have a custom `persistent_id()` method that takes an object as an argument and returns either `None` or the persistent id for that object. When `None` is returned, the pickler simply pickles the object as normal. When a persistent ID string is returned, the pickler will pickle that object, along with a marker so that the unpickler will recognize it as a persistent ID.

To unpickle external objects, the unpickler must have a custom `persistent_load()` method that takes a persistent ID object and returns the referenced object.

Here is a comprehensive example presenting how persistent ID can be used to pickle external objects by reference.

```python
# Simple example presenting how persistent ID can be used to pickle external objects by reference.

import pickle
import sqlite3
from collections import namedtuple

---

\(^4\) The limitation on alphanumeric characters is due to the fact the persistent IDs, in protocol 0, are delimited by the newline character. Therefore if any kind of newline characters occurs in persistent IDs, the resulting pickle will become unreadable.
# Simple class representing a record in our database.
MemoRecord = namedtuple("MemoRecord", "key, task")

class DBPickler(pickle.Pickler):
    def persistent_id(self, obj):
        # Instead of pickling MemoRecord as a regular class instance, we emit a
        # persistent ID.
        if isinstance(obj, MemoRecord):
            # Here, our persistent ID is simply a tuple, containing a tag and a
            # key, which refers to a specific record in the database.
            return ("MemoRecord", obj.key)
        else:
            # If obj does not have a persistent ID, return None. This means obj
            # needs to be pickled as usual.
            return None

class DBUnpickler(pickle.Unpickler):
    def __init__(self, file, connection):
        super().__init__(file)
        self.connection = connection
    def persistent_load(self, pid):
        # This method is invoked whenever a persistent ID is encountered.
        # Here, pid is the tuple returned by DBPickler.
        cursor = self.connection.cursor()
        type_tag, key_id = pid
        if type_tag == "MemoRecord":
            # Fetch the referenced record from the database and return it.
            key, task = cursor.execute("SELECT * FROM memos WHERE key=?", (str(key_id),)).fetchone()
            return MemoRecord(key, task)
        else:
            # Always raises an error if you cannot return the correct object.
            # Otherwise, the unpickler will think None is the object referenced
            # by the persistent ID.
            raise pickle.UnpicklingError("unsupported persistent object")

def main():
    import io
    import pprint

    # Initialize and populate our database.
    conn = sqlite3.connect(":memory:")
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE memos(key INTEGER PRIMARY KEY, task TEXT)")
    tasks = 
    "give food to fish",
    "prepare group meeting",
    "fight with a zebra",
    )
    for task in tasks:
        cursor.execute("INSERT INTO memos VALUES(NULL, ?)", (task,))

    # Fetch the records to be pickled.
cursor.execute("SELECT * FROM memos")
memos = [MemoRecord(key, task) for key, task in cursor]
# Save the records using our custom DBPickler.
file = io.BytesIO()
DBPickler(file).dump(memos)

print("Pickled records:")
pprint.pprint(memos)

# Update a record, just for good measure.
cursor.execute("UPDATE memos SET task='learn italian' WHERE key=1")

# Load the records from the pickle data stream.
file.seek(0)
memos = DBUnpickler(file, conn).load()

print("Unpickled records")
pprint.pprint(memos)

if __name__ == '__main__':
    main()

Dispatch Tables

If one wants to customize pickling of some classes without disturbing any other code which depends on pickling, then one can create a pickler with a private dispatch table.

The global dispatch table managed by the copyreg module is available as copyreg.dispatch_table. Therefore, one may choose to use a modified copy of copyreg.dispatch_table as a private dispatch table.

For example
f = io.BytesIO()
p = pickle.Pickler(f)
p.dispatch_table = copyreg.dispatch_table.copy()
p.dispatch_table[SomeClass] = reduce_SomeClass
creates an instance of pickle.Pickler with a private dispatch table which handles the SomeClass class specially. Alternatively, the code

class MyPickler(pickle.Pickler):
    dispatch_table = copyreg.dispatch_table.copy()
    dispatch_table[SomeClass] = reduce_SomeClass
f = io.BytesIO()
p = MyPickler(f)

does the same, but all instances of MyPickler will by default share the same dispatch table. The equivalent code using the copyreg module is

    copyreg.pickle(SomeClass, reduce_SomeClass)
f = io.BytesIO()
p = pickle.Pickler(f)

Handling Stateful Objects

Here's an example that shows how to modify pickling behavior for a class. The TextReader class opens a text file, and returns the line number and line contents each time its readline() method is called. If a TextReader instance is pickled, all attributes except the file object member are saved. When the instance
is unpickled, the file is reopened, and reading resumes from the last location. The __setstate__() and __getstate__() methods are used to implement this behavior.

class TextReader:
    """Print and number lines in a text file."""
    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename)
        self.lineno = 0
    def readline(self):
        self.lineno += 1
        line = self.file.readline()
        if not line:
            return None
        if line.endswith(\n):
            line = line[:-1]
        return "%i: %s" % (self.lineno, line)
    def __getstate__(self):
        # Copy the object’s state from self.__dict__ which contains
        # all our instance attributes. Always use the dict.copy()
        # method to avoid modifying the original state.
        state = self.__dict__.copy()
        # Remove the unpicklable entries.
        del state['file']
        return state
    def __setstate__(self, state):
        # Restore instance attributes (i.e., filename and lineno).
        self.__dict__.update(state)
        # Restore the previously opened file’s state. To do so, we need to
        # reopen it and read from it until the line count is restored.
        file = open(self.filename)
        for _ in range(self.lineno):
            file.readline()
        # Finally, save the file.
        self.file = file

A sample usage might be something like this:

```python
>>> reader = TextReader("hello.txt")
>>> reader.readline()
'1: Hello world!'
>>> reader.readline()
'2: I am line number two.'
>>> new_reader = pickle.loads(pickle.dumps(reader))
>>> new_reader.readline()
'3: Goodbye!'
```

### 12.1.6 Restricting Globals

By default, unpickling will import any class or function that it finds in the pickle data. For many applications, this behaviour is unacceptable as it permits the unpickler to import and invoke arbitrary code. Just consider what this hand-crafted pickle data stream does when loaded:

```python
>>> import pickle
>>> pickle.loads(b"cos\nsystem\n(S'echo hello world'\ntr."
```
hello world

In this example, the unpickler imports the `os.system()` function and then apply the string argument “echo hello world”. Although this example is inoffensive, it is not difficult to imagine one that could damage your system.

For this reason, you may want to control what gets unpickled by customizing `Unpickler.find_class()`. Unlike its name suggests, `Unpickler.find_class()` is called whenever a global (i.e., a class or a function) is requested. Thus it is possible to either completely forbid globals or restrict them to a safe subset.

Here is an example of an unpickler allowing only few safe classes from the `builtins` module to be loaded:

```python
import builtins
import io
import pickle

safe_builtins = {
    'range',
    'complex',
    'set',
    'frozenset',
    'slice',
}

class RestrictedUnpickler(pickle.Unpickler):
    def find_class(self, module, name):
        # Only allow safe classes from builtins.
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # Forbid everything else.
        raise pickle.UnpicklingError("global '%s.%s' is forbidden" %
            (module, name))

def restricted_loads(s):
    """Helper function analogous to pickle.loads().""
    return RestrictedUnpickler(io.BytesIO(s)).load()

A sample usage of our unpickler working has intended:

```python
>>> restricted_loads(pickle.dumps([1, 2, range(15)]))
[1, 2, range(0, 15)]
>>> restricted_loads(b"cos\n\nsystem\n\n(S'echo hello world')")
Traceback (most recent call last):
  ...pickle.UnpicklingError: global 'os.system' is forbidden
>>> restricted_loads(b'cbuiltins\neval\n'
  ...b'(S\'getattr(__import__("os"), "system")'
  ...b'("echo hello world")')
Traceback (most recent call last):
  ...pickle.UnpicklingError: global 'builtins.eval' is forbidden
```

As our examples shows, you have to be careful with what you allow to be unpickled. Therefore if security is a concern, you may want to consider alternatives such as the marshalling API in `xmlrpc.client` or third-party solutions.

### 12.1.7 Examples

For the simplest code, use the `dump()` and `load()` functions.
import pickle

# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ("character string", b"byte string"),
    'c': set([None, True, False])
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)

The following example reads the resulting pickled data.

import pickle

with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so we do not
    # have to specify it.
    data = pickle.load(f)

See Also:
Module copyreg  Pickle interface constructor registration for extension types.
Module pickletools  Tools for working with and analyzing pickled data.
Module shelve  Indexed databases of objects; uses pickle.
Module copy  Shallow and deep object copying.
Module marshal  High-performance serialization of built-in types.

12.2 copyreg — Register pickle support functions

The copyreg module offers a way to define functions used while pickling specific objects. The pickle and copy modules use those functions when pickling/copying those objects. The module provides configuration information about object constructors which are not classes. Such constructors may be factory functions or class instances.

copyreg.constructor(object)
    Declares object to be a valid constructor. If object is not callable (and hence not valid as a constructor), raises TypeError.

copyreg.pickle(type, function, constructor=\None)
    Declares that function should be used as a “reduction” function for objects of type type. function should return either a string or a tuple containing two or three elements.

    The optional constructor parameter, if provided, is a callable object which can be used to reconstruct the object when called with the tuple of arguments returned by function at pickling time. TypeError will be raised if object is a class or constructor is not callable.

    See the pickle module for more details on the interface expected of function and constructor. Note that the dispatch_table attribute of a pickler object or subclass of pickle.Pickler can also be used for declaring reduction functions.

12.2.1 Example

The example below would like to show how to register a pickle function and how it will be used:
>>> import copyreg, copy, pickle
>>> class C(object):
...     def __init__(self, a):
...         self.a = a
...     def pickle_c(c):
...         print("pickling a C instance...")
...         return C, (c.a,)
...     copyreg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)
pickling a C instance...
>>> p = pickle.dumps(c)
pickling a C instance...

12.3 shelf — Python object persistence

Source code: Lib/shelve.py

A “shelf” is a persistent, dictionary-like object. The difference with “dbm” databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects — anything that the pickle module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys are ordinary strings.

shelf.open(filename, flag='c', protocol=None, writeback=False)
Open a persistent dictionary. The filename specified is the base filename for the underlying database. As a side-effect, an extension may be added to the filename and more than one file may be created. By default, the underlying database file is opened for reading and writing. The optional flag parameter has the same interpretation as the flag parameter of dbm.open().

By default, version 3 pickles are used to serialize values. The version of the pickle protocol can be specified with the protocol parameter.

Because of Python semantics, a shelf cannot know when a mutable persistent-dictionary entry is modified. By default modified objects are written only when assigned to the shelf (see Example). If the optional writeback parameter is set to True, all entries accessed are also cached in memory, and written back on sync() and close(); this can make it handier to mutate mutable entries in the persistent dictionary, but, if many entries are accessed, it can consume vast amounts of memory for the cache, and it can make the close operation very slow since all accessed entries are written back (there is no way to determine which accessed entries are mutable, nor which ones were actually mutated).

Note: Do not rely on the shelf being closed automatically; always call close() explicitly when you don’t need it any more, or use a with statement with contextlib.closing().

Warning: Because the shelf module is backed by pickle, it is insecure to load a shelf from an untrusted source. Like with pickle, loading a shelf can execute arbitrary code.

Shelf objects support all methods supported by dictionaries. This eases the transition from dictionary based scripts to those requiring persistent storage.

Two additional methods are supported:

Shelf.sync()
Write back all entries in the cache if the shelf was opened with writeback set to True. Also empty the cache and synchronize the persistent dictionary on disk, if feasible. This is called automatically when the shelf is closed with close().
Shelf.close()

Synchronize and close the persistent dict object. Operations on a closed shelf will fail with a ValueError.

See Also:
Persistent dictionary recipe with widely supported storage formats and having the speed of native dictionaries.

12.3.1 Restrictions

- The choice of which database package will be used (such as dbm.ndbm or dbm.gnu) depends on which interface is available. Therefore it is not safe to open the database directly using dbm. The database is also (unfortunately) subject to the limitations of dbm, if it is used — this means that (the pickled representation of) the objects stored in the database should be fairly small, and in rare cases key collisions may cause the database to refuse updates.

- The shelf module does not support concurrent read/write access to shelved objects. (Multiple simultaneous read accesses are safe.) When a program has a shelf open for writing, no other program should have it open for reading or writing. Unix file locking can be used to solve this, but this differs across Unix versions and requires knowledge about the database implementation used.

class shelf.Shelf(dict, protocol=None, writeback=False, keyencoding='utf-8')
A subclass of collections.abc.MutableMapping which stores pickled values in the dict object. By default, version 0 pickles are used to serialize values. The version of the pickle protocol can be specified with the protocol parameter. See the pickle documentation for a discussion of the pickle protocols.

If the writeback parameter is True, the object will hold a cache of all entries accessed and write them back to the dict at sync and close times. This allows natural operations on mutable entries, but can consume much more memory and make sync and close take a long time.

The keyencoding parameter is the encoding used to encode keys before they are used with the underlying dict. New in version 3.2: The keyencoding parameter; previously, keys were always encoded in UTF-8.

class shelf.BsdDbShelf(dict, protocol=None, writeback=False, keyencoding='utf-8')
A subclass of Shelf which exposes first(), next(), previous(), last() and set_location() which are available in the third-party bsddb module from pybsddb but not in other database modules. The dict object passed to the constructor must support those methods. This is generally accomplished by calling one of bsddb.hashopen(), bsddb.btopen() or bsddb.rnopen(). The optional protocol, writeback, and keyencoding parameters have the same interpretation as for the Shelf class.

class shelf.DbfilenameShelf(filename, flag='c', protocol=None, writeback=False)
A subclass of Shelf which accepts a filename instead of a dict-like object. The underlying file will be opened using dbm.open(). By default, the file will be created and opened for both read and write. The optional flag parameter has the same interpretation as for the open() function. The optional protocol and writeback parameters have the same interpretation as for the Shelf class.

12.3.2 Example

To summarize the interface (key is a string, data is an arbitrary object):

```python
import shelve

d = shelve.open(filename)  # open -- file may get suffix added by low-level # library

d[key] = data  # store data at key (overwrites old data if # using an existing key)
data = d[key]  # retrieve a COPY of data at key (raise KeyError if no # such key)
```
The Python Library Reference, Release 3.3.3

\[\textbf{del d[key]} \] # delete data stored at key (raises KeyError
# if no such key)
\[\text{flag = key in d} \] # true if the key exists
\[\text{klist = list(d.keys())} \] # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
\[\text{d['xx'] = [0, 1, 2]} \] # this works as expected, but...
\[\text{d['xx'].append(3)} \] # *this doesn't!* -- d['xx'] is STILL [0, 1, 2]!

# having opened d without writeback=True, you need to code carefully:
\[\text{temp = d['xx']} \] # extracts the copy
\[\text{temp.append(5)} \] # mutates the copy
\[\text{d['xx'] = temp} \] # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

\[\text{d.close()} \] # close it

See Also:

Module \texttt{dbm} Generic interface to \texttt{dbm}-style databases.
Module \texttt{pickle} Object serialization used by \texttt{shelve}.

12.4 \texttt{marshal} — Internal Python object serialization

This module contains functions that can read and write Python values in a binary format. The format is specific
to Python, but independent of machine architecture issues (e.g., you can write a Python value to a file on a PC,
transport the file to a Sun, and read it back there). Details of the format are undocumented on purpose; it may
change between Python versions (although it rarely does).\(^5\)

This is not a general “persistence” module. For general persistence and transfer of Python objects through RPC
calls, see the modules \texttt{pickle} and \texttt{shelve}. The \texttt{marshal} module exists mainly to support reading and writing
the “pseudo-compiled” code for Python modules of \texttt{.pyc} files. Therefore, the Python maintainers reserve the
right to modify the marshal format in backward incompatible ways should the need arise. If you’re serializing
and de-serializing Python objects, use the \texttt{pickle} module instead – the performance is comparable, version
independence is guaranteed, and pickle supports a substantially wider range of objects than marshal.

\textbf{Warning:} The \texttt{marshal} module is not intended to be secure against erroneous or maliciously constructed
data. Never unmarshal data received from an untrusted or unauthenticated source.

Not all Python object types are supported; in general, only objects whose value is independent from a particular in-
vocation of Python can be written and read by this module. The following types are supported: booleans, integers,
floating point numbers, complex numbers, strings, bytes, bytearray, tuples, lists, sets, frozensets, dictionaries, and
code objects, where it should be understood that tuples, lists, sets, frozensets and dictionaries are only supported
as long as the values contained therein are themselves supported; and recursive lists, sets and dictionaries should
not be written (they will cause infinite loops). The singletons \texttt{None}, \texttt{Ellipsis} and \texttt{StopIteration} can also
be marshalled and unmarshalled.

There are functions that read/write files as well as functions operating on strings.

The module defines these functions:

\begin{verbatim}
marshal.dump(value, file[, version])
Write the value on the open file. The value must be a supported type. The file must be an open file object
\end{verbatim}

\(^5\) The name of this module stems from a bit of terminology used by the designers of Modula-3 (amongst others), who use the term
“marshalling” for shipping of data around in a self-contained form. Strictly speaking, “to marshal” means to convert some data from internal
to external form (in an RPC buffer for instance) and “unmarshalling” for the reverse process.
such as `sys.stdout` or returned by `open()` or `os.popen()`. It must be opened in binary mode (`'wb'` or `'w+b'`).

If the value has (or contains an object that has) an unsupported type, a `ValueError` exception is raised — but garbage data will also be written to the file. The object will not be properly read back by `load()`.

The `version` argument indicates the data format that `dump` should use (see below).

```
marshal.load(file)
```

Read one value from the open file and return it. If no valid value is read (e.g. because the data has a different Python version’s incompatible marshal format), raise `EOFError`, `ValueError` or `TypeError`. The file must be an open file object opened in binary mode (`'rb'` or `'r+b'`).

**Note:** If an object containing an unsupported type was marshalled with `dump()`, `load()` will substitute `None` for the unmarshallable type.

```
marshal.dumps(value[, version])
```

Return the string that would be written to a file by `dump(value, file)`. The value must be a supported type. Raise a `ValueError` exception if value has (or contains an object that has) an unsupported type.

The `version` argument indicates the data format that `dumps` should use (see below).

```
marshal.loads(string)
```

Convert the string to a value. If no valid value is found, raise `EOFError`, `ValueError` or `TypeError`. Extra characters in the string are ignored.

In addition, the following constants are defined:

```
marshal.version
```

Indicates the format that the module uses. Version 0 is the historical format, version 1 shares interned strings and version 2 uses a binary format for floating point numbers. The current version is 2.

### 12.5 `dbm` — Interfaces to Unix “databases”

`dbm` is a generic interface to variants of the DBM database — `dbm.gnu` or `dbm.ndbm`. If none of these modules is installed, the slow-but-simple implementation in module `dbm.dumb` will be used. There is a third party interface to the Oracle Berkeley DB.

```
exception dbm.error
```

A tuple containing the exceptions that can be raised by each of the supported modules, with a unique exception also named `dbm.error` as the first item — the latter is used when `dbm.error` is raised.

```
dbm.whichdb(filename)
```

This function attempts to guess which of the several simple database modules available — `dbm.gnu`, `dbm.ndbm` or `dbm.dumb` — should be used to open a given file.

Returns one of the following values: `None` if the file can’t be opened because it’s unreadable or doesn’t exist; the empty string (`''`) if the file’s format can’t be guessed; or a string containing the required module name, such as `'dbm.ndbm'` or `'dbm.gnu'`.

```
dbm.open(file, flag='r', mode=0o666)
```

Open the database file `file` and return a corresponding object.

If the database file already exists, the `whichdb()` function is used to determine its type and the appropriate module is used; if it does not exist, the first module listed above that can be imported is used.

The optional `flag` argument can be:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>'r'</code></td>
<td>Open existing database for reading only (default)</td>
</tr>
<tr>
<td><code>'w'</code></td>
<td>Open existing database for reading and writing</td>
</tr>
<tr>
<td><code>'c'</code></td>
<td>Open database for reading and writing, creating it if it doesn’t exist</td>
</tr>
<tr>
<td><code>'n'</code></td>
<td>Always create a new, empty database, open for reading and writing</td>
</tr>
</tbody>
</table>
The optional *mode* argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0o666 (and will be modified by the prevailing umask).

The object returned by `open()` supports the same basic functionality as dictionaries; keys and their corresponding values can be stored, retrieved, and deleted, and the `in` operator and the `keys()` method are available, as well as `get()` and `setdefault()`. Changed in version 3.2: `get()` and `setdefault()` are now available in all database modules. Key and values are always stored as bytes. This means that when strings are used they are implicitly converted to the default encoding before being stored.

The following example records some hostnames and a corresponding title, and then prints out the contents of the database:

```python
import dbm

db = dbm.open('cache', 'c')

# Record some values
db[b'hello'] = b'there'
db['www.python.org'] = 'Python Website'
db['www.cnn.com'] = 'Cable News Network'

# Note that the keys are considered bytes now.
assert db[b'www.python.org'] == b'Python Website'
# Notice how the value is now in bytes.
assert db['www.cnn.com'] == b'Cable News Network'

# Often-used methods of the dict interface work too.
print(db.get('python.org', b'not present'))

# Storing a non-string key or value will raise an exception (most likely a TypeError).
db['www.yahoo.com'] = 4

# Close when done.
db.close()
```

See Also:

- **Module shelve** Persistence module which stores non-string data.
The individual submodules are described in the following sections.

### 12.5.1 dbm.gnu — GNU’s reinterpretation of dbm

*Platforms:* Unix

This module is quite similar to the `dbm` module, but uses the GNU library `gdbm` instead to provide some additional functionality. Please note that the file formats created by `dbm.gnu` and `dbm.ndbm` are incompatible.

The `dbm.gnu` module provides an interface to the GNU DBM library. `dbm.gnu.gdbm` objects behave like mappings (dictionaries), except that keys and values are always converted to bytes before storing. Printing a `gdbm` object doesn’t print the keys and values, and the `items()` and `values()` methods are not supported.

*exception* `dbm.gnu.error`

Raised on `dbm.gnu`-specific errors, such as I/O errors. `KeyError` is raised for general mapping errors like specifying an incorrect key.

```python
dbm.gnu.open(filename[, flag[, mode ]])
```

Open a `gdbm` database and return a `gdbm` object. The `filename` argument is the name of the database file.

The optional `flag` argument can be:
### The Python Library Reference, Release 3.3.3

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r</code></td>
<td>Open existing database for reading only (default)</td>
</tr>
<tr>
<td><code>w</code></td>
<td>Open existing database for reading and writing</td>
</tr>
<tr>
<td><code>c</code></td>
<td>Open database for reading and writing, creating it if it doesn’t exist</td>
</tr>
<tr>
<td><code>n</code></td>
<td>Always create a new, empty database, open for reading and writing</td>
</tr>
</tbody>
</table>

The following additional characters may be appended to the flag to control how the database is opened:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f</code></td>
<td>Open the database in fast mode. Writes to the database will not be synchronized.</td>
</tr>
<tr>
<td><code>s</code></td>
<td>Synchronized mode. This will cause changes to the database to be immediately written to the file.</td>
</tr>
<tr>
<td><code>u</code></td>
<td>Do not lock database.</td>
</tr>
</tbody>
</table>

Not all flags are valid for all versions of `gdbm`. The module constant `open_flags` is a string of supported flag characters. The exception `error` is raised if an invalid flag is specified.

The optional `mode` argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal `0o666`.

In addition to the dictionary-like methods, `gdbm` objects have the following methods:

**`gdbm.firstkey()`**

It’s possible to loop over every key in the database using this method and the `nextkey()` method. The traversal is ordered by `gdbm`’s internal hash values, and won’t be sorted by the key values. This method returns the starting key.

**`gdbm.nextkey(key)`**

Returns the key that follows `key` in the traversal. The following code prints every key in the database `db`, without having to create a list in memory that contains them all:

```python
k = db.firstkey()
while k != None:
    print(k)
    k = db.nextkey(k)
```

**`gdbm.reorganize()`**

If you have carried out a lot of deletions and would like to shrink the space used by the `gdbm` file, this routine will reorganize the database. `gdbm` objects will not shorten the length of a database file except by using this reorganization; otherwise, deleted file space will be kept and reused as new (key, value) pairs are added.

**`gdbm.sync()`**

When the database has been opened in fast mode, this method forces any unwritten data to be written to the disk.

### 12.5.2 `dbm.ndbm` — Interface based on ndbm

**Platforms:** Unix

The `dbm.ndbm` module provides an interface to the Unix “(n)dbm” library. Dbm objects behave like mappings (dictionaries), except that keys and values are always stored as bytes. Printing a `dbm` object doesn’t print the keys and values, and the `items()` and `values()` methods are not supported.

This module can be used with the “classic” `ndbm` interface or the GNU GDBM compatibility interface. On Unix, the `configure` script will attempt to locate the appropriate header file to simplify building this module.

**exception `dbm.ndbm.error`**

Raised on `dbm.ndbm`-specific errors, such as I/O errors. `KeyError` is raised for general mapping errors like specifying an incorrect key.

**`dbm.ndbm.library`**

Name of the `ndbm` implementation library used.
The Python Library Reference, Release 3.3.3

dbm.ndbm.open(filename[, flag[, mode ]])
Open a dbm database and return a dbm object. The filename argument is the name of the database file (without the .dir or .pag extensions).

The optional flag argument must be one of these values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r'</td>
<td>Open existing database for reading only (default)</td>
</tr>
<tr>
<td>'w'</td>
<td>Open existing database for reading and writing</td>
</tr>
<tr>
<td>'c'</td>
<td>Open database for reading and writing, creating it if it doesn’t exist</td>
</tr>
<tr>
<td>'n'</td>
<td>Always create a new, empty database, open for reading and writing</td>
</tr>
</tbody>
</table>

The optional mode argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0o666 (and will be modified by the prevailing umask).

12.5.3 dbm.dumb — Portable DBM implementation

Note: The dbm.dumb module is intended as a last resort fallback for the dbm module when a more robust module is not available. The dbm.dumb module is not written for speed and is not nearly as heavily used as the other database modules.

The dbm.dumb module provides a persistent dictionary-like interface which is written entirely in Python. Unlike other modules such as dbm.gnu no external library is required. As with other persistent mappings, the keys and values are always stored as bytes.

The module defines the following:

exception dbm.dumb.error
Raised on dbm.dumb-specific errors, such as I/O errors. KeyError is raised for general mapping errors like specifying an incorrect key.

dbm.dumb.open(filename[, flag[, mode ]])
Open a dumbdbm database and return a dumbdbm object. The filename argument is the basename of the database file (without any specific extensions). When a dumbdbm database is created, files with .dat and .dir extensions are created.

The optional flag argument is currently ignored; the database is always opened for update, and will be created if it does not exist.

The optional mode argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0o666 (and will be modified by the prevailing umask).

In addition to the methods provided by the collections.abc.MutableMapping class, dumbdbm objects provide the following method:

dumbdbm.sync()
Synchronize the on-disk directory and data files. This method is called by the Shelve.sync() method.

12.6 sqlite3 — DB-API 2.0 interface for SQLite databases

SQLite is a C library that provides a lightweight disk-based database that doesn’t require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It’s also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

sqlite3 was written by Gerhard Häring and provides a SQL interface compliant with the DB-API 2.0 specification described by PEP 249.
To use the module, you must first create a `Connection` object that represents the database. Here the data will be stored in the `example.db` file:

```python
import sqlite3
conn = sqlite3.connect('example.db')
```

You can also supply the special name `:memory:` to create a database in RAM.

Once you have a `Connection`, you can create a `Cursor` object and call its `execute()` method to perform SQL commands:

```python
c = conn.cursor()

# Create table
c.execute('''create table stocks
    (date text, trans text, symbol text,
    qty real, price real)''')

# Insert a row of data
values = ('2006-01-05','BUY','RHAT',100,35.14)
c.execute('''insert into stocks
    values (?,?,?,?,'''')

# Save (commit) the changes
conn.commit()

# We can also close the cursor if we are done with it
```

Usually your SQL operations will need to use values from Python variables. You shouldn’t assemble your query using Python’s string operations because doing so is insecure; it makes your program vulnerable to an SQL injection attack.

Instead, use the DB-API’s parameter substitution. Put `?` as a placeholder wherever you want to use a value, and then provide a tuple of values as the second argument to the cursor’s `execute()` method. (Other database modules may use a different placeholder, such as `%s` or `:1`.) For example:

```python
symbol = ‘IBM’
c.execute("select * from stocks where symbol = '%s'" % symbol)
```

```python
# Do this instead
t = (‘IBM’,)
c.execute('select * from stocks where symbol=?', t)
```

```python
# Larger example
for t in [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
        ('2006-04-05', 'BUY', 'MSFT', 1000, 72.00),
        ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
    ]:
    c.execute('insert into stocks values (?, ?, ?, ?, ?)', t)
```

To retrieve data after executing a SELECT statement, you can either treat the cursor as an iterator, call the cursor’s `fetchone()` method to retrieve a single matching row, or call `fetchall()` to get a list of the matching rows.

This example uses the iterator form:

```python
>> c = conn.cursor()
>>> c.execute('select * from stocks order by price')
... for row in c:
...     print(row)
... [('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
  ('2006-03-28', 'BUY', 'IBM', 1000, 45.00)
See Also:

http://code.google.com/p/pysqlite/  The pysqlite web page – sqlite3 is developed externally under the name “pysqlite”.

http://www.sqlite.org  The SQLite web page; the documentation describes the syntax and the available data types for the supported SQL dialect.

PEP 249 - Database API Specification 2.0  PEP written by Marc-André Lemburg.

12.6.1 Module functions and constants

sqlite3.version
The version number of this module, as a string. This is not the version of the SQLite library.

sqlite3.version_info
The version number of this module, as a tuple of integers. This is not the version of the SQLite library.

sqlite3.sqlite_version
The version number of the run-time SQLite library, as a string.

sqlite3.sqlite_version_info
The version number of the run-time SQLite library, as a tuple of integers.

sqlite3.PARSE_DECLTYPES
This constant is meant to be used with the detect_types parameter of the connect() function.

Setting it makes the sqlite3 module parse the declared type for each column it returns. It will parse out the first word of the declared type, i.e. for “integer primary key”, it will parse out “integer”, or for “number(10)” it will parse out “number”. Then for that column, it will look into the converters dictionary and use the converter function registered for that type there.

sqlite3.PARSE_COLNAMES
This constant is meant to be used with the detect_types parameter of the connect() function.

Setting this makes the SQLite interface parse the column name for each column it returns. It will look for a string formed [mytype] in there, and then decide that ‘mytype’ is the type of the column. It will try to find an entry of ‘mytype’ in the converters dictionary and then use the converter function found there to return the value. The column name found in Cursor.description is only the first word of the column name, i.e. if you use something like ‘as "x [datetime]"’ in your SQL, then we will parse out everything until the first blank for the column name: the column name would simply be “x”.

sqlite3.connect(database[, timeout, detect_types, isolation_level, check_same_thread, factory, cached_statements])
Opens a connection to the SQLite database file database. You can use ":memory:" to open a database connection to a database that resides in RAM instead of on disk.

When a database is accessed by multiple connections, and one of the processes modifies the database, the SQLite database is locked until that transaction is committed. The timeout parameter specifies how long the connection should wait for the lock to go away until raising an exception. The default for the timeout parameter is 5.0 (five seconds).

For the isolation_level parameter, please see the Connection.isolation_level property of Connection objects.

SQLite natively supports only the types TEXT, INTEGER, REAL, BLOB and NULL. If you want to use other types you must add support for them yourself. The detect_types parameter and the using custom converters registered with the module-level register_converter() function allow you to easily do that.
detect_types defaults to 0 (i.e. off, no type detection), you can set it to any combination of PARSE_DECLTYPES and PARSE_COLNAMES to turn type detection on.

By default, the sqlite3 module uses its Connection class for the connect call. You can, however, subclass the Connection class and make connect() use your class instead by providing your class for the factory parameter.

Consult the section SQLite and Python types of this manual for details.

The sqlite3 module internally uses a statement cache to avoid SQL parsing overhead. If you want to explicitly set the number of statements that are cached for the connection, you can set the cached_statements parameter. The currently implemented default is to cache 100 statements.

sqlite3.register_converter(typename, callable)
Registers a callable to convert a bytestring from the database into a custom Python type. The callable will be invoked for all database values that are of the type typename. Confer the parameter detect_types of the connect() function for how the type detection works. Note that the case of typename and the name of the type in your query must match!

sqlite3.register_adapter(type, callable)
Registers a callable to convert the custom Python type type into one of SQLite’s supported types. The callable callable accepts as single parameter the Python value, and must return a value of the following types: int, float, str or bytes.

sqlite3.complete_statement(sql)
Returns True if the string sql contains one or more complete SQL statements terminated by semicolons. It does not verify that the SQL is syntactically correct, only that there are no unclosed string literals and the statement is terminated by a semicolon.

This can be used to build a shell for SQLite, as in the following example:

```python
# A minimal SQLite shell for experiments

import sqlite3

con = sqlite3.connect(":memory:")
con.isolation_level = None
cur = con.cursor()

buffer = ""

print("Enter your SQL commands to execute in sqlite3.")
print("Enter a blank line to exit.")

while True:
    line = input()
    if line == "":
        break
    buffer += line
    if sqlite3.complete_statement(buffer):
        try:
            buffer = buffer.strip()
            cur.execute(buffer)

            if buffer.lstrip().upper().startswith("SELECT"):
                print(cur.fetchall())
            except sqlite3.Error as e:
                print("An error occurred: ", e.args[0])
        buffer = ""

con.close()
```
.sqlite3.enable_callback_tracebacks(flag)

By default you will not get any tracebacks in user-defined functions, aggregates, converters, authorizer callbacks etc. If you want to debug them, you can call this function with flag as True. Afterwards, you will get tracebacks from callbacks on sys.stderr. Use False to disable the feature again.

12.6.2 Connection Objects

class sqlite3.Connection

A SQLite database connection has the following attributes and methods:

isolation_level
Get or set the current isolation level. None for autocommit mode or one of “DEFERRED”, “IMMEDIATE” or “EXCLUSIVE”. See section Controlling Transactions for a more detailed explanation.

in_transaction
True if a transaction is active (there are uncommitted changes), False otherwise. Read-only attribute. New in version 3.2.

cursor(cursorClass)
The cursor method accepts a single optional parameter cursorClass. If supplied, this must be a custom cursor class that extends sqlite3.Cursor.

commit()
This method commits the current transaction. If you don’t call this method, anything you did since the last call to commit() is not visible from other database connections. If you wonder why you don’t see the data you’ve written to the database, please check you didn’t forget to call this method.

rollback()
This method rolls back any changes to the database since the last call to commit().

close()
This closes the database connection. Note that this does not automatically call commit(). If you just close your database connection without calling commit() first, your changes will be lost!

execute(sql, parameters)
This is a nonstandard shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor’s execute method with the parameters given.

executemany(sql, parameters)
This is a nonstandard shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor’s executemany method with the parameters given.

executescript(sql_script)
This is a nonstandard shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor’s executescript method with the parameters given.

create_function(name, num_params, func)
Creates a user-defined function that you can later use from within SQL statements under the function name name. num_params is the number of parameters the function accepts, and func is a Python callable that is called as the SQL function.

The function can return any of the types supported by SQLite: bytes, str, int, float and None.

Example:

import sqlite3
import hashlib

def md5sum(t):
    return hashlib.md5(t).hexdigest()

con = sqlite3.connect(":memory:")
con.create_function("md5", 1, md5sum)
```python
cur = con.cursor()
cur.execute("select md5(?)", (b"foo",))
print(cur.fetchone()[0])
```

**create_aggregate** *(name, num_params, aggregate_class)*

Creates a user-defined aggregate function.

The aggregate class must implement a `step` method, which accepts the number of parameters `num_params`, and a `finalize` method which will return the final result of the aggregate.

The `finalize` method can return any of the types supported by SQLite: bytes, str, int, float and None.

Example:

```python
import sqlite3

class MySum:
    def __init__(self):
        self.count = 0

    def step(self, value):
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
cur = con.cursor()
cur.execute("create table test(i)")
cur.execute("insert into test(i) values (1)")
cur.execute("insert into test(i) values (2)")
cur.execute("select mysum(i) from test")
print(cur.fetchone()[0])
```

**create_collation** *(name, callable)*

Creates a collation with the specified `name` and `callable`. The callable will be passed two string arguments. It should return -1 if the first is ordered lower than the second, 0 if they are ordered equal and 1 if the first is ordered higher than the second. Note that this controls sorting (ORDER BY in SQL) so your comparisons don’t affect other SQL operations.

Note that the callable will get its parameters as Python bytestrings, which will normally be encoded in UTF-8.

The following example shows a custom collation that sorts “the wrong way”:

```python
import sqlite3
def collate_reverse(string1, string2):
    if string1 == string2:
        return 0
    elif string1 < string2:
        return 1
    else:
        return -1

con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)
cur = con.cursor()
```
cur.execute("create table test(x)")
cur.executemany("insert into test(x) values (?)", [("a"), ("b")])
cur.execute("select x from test order by x collate reverse")
for row in cur:
    print(row)
con.close()

To remove a collation, call create_collation with None as callable:

con.create_collation("reverse", None)

interrupt()
You can call this method from a different thread to abort any queries that might be executing on the connection. The query will then abort and the caller will get an exception.

set_authorizer(authorizer_callback)
This routine registers a callback. The callback is invoked for each attempt to access a column of a table in the database. The callback should return SQLITE_OK if access is allowed, SQLITE_DENY if the entire SQL statement should be aborted with an error and SQLITE_IGNORE if the column should be treated as a NULL value. These constants are available in the sqlite3 module.

The first argument to the callback signifies what kind of operation is to be authorized. The second and third argument will be arguments or None depending on the first argument. The 4th argument is the name of the database ("main", "temp", etc.) if applicable. The 5th argument is the name of the inner-most trigger or view that is responsible for the access attempt or None if this access attempt is directly from input SQL code.

Please consult the SQLite documentation about the possible values for the first argument and the meaning of the second and third argument depending on the first one. All necessary constants are available in the sqlite3 module.

set_progress_handler(handler, n)
This routine registers a callback. The callback is invoked for every n instructions of the SQLite virtual machine. This is useful if you want to get called from SQLite during long-running operations, for example to update a GUI.

If you want to clear any previously installed progress handler, call the method with None for handler.

set_trace_callback(trace_callback)
Registers trace_callback to be called for each SQL statement that is actually executed by the SQLite backend.

The only argument passed to the callback is the statement (as string) that is being executed. The return value of the callback is ignored. Note that the backend does not only run statements passed to the Cursor.execute() methods. Other sources include the transaction management of the Python module and the execution of triggers defined in the current database.

Passing None as trace_callback will disable the trace callback. New in version 3.3.

enable_load_extension(enabled)
This routine allows/disallows the SQLite engine to load SQLite extensions from shared libraries. SQLite extensions can define new functions, aggregates or whole new virtual table implementations. One well-known extension is the fulltext-search extension distributed with SQLite.

Loadable extensions are disabled by default. See \(^6\). New in version 3.2.

import sqlite3
con = sqlite3.connect(":memory:")

---

\(^6\) The sqlite3 module is not built with loadable extension support by default, because some platforms (notably Mac OS X) have SQLite libraries which are compiled without this feature. To get loadable extension support, you must pass --enable-loadable-sqlite-extensions to configure.
# enable extension loading
con.enable_load_extension(True)

# Load the fulltext search extension
con.execute("select load_extension('./fts3.so')")

# alternatively you can load the extension using an API call:
# con.load_extension("./fts3.so")

# disable extension loading again
con.enable_load_extension(False)

# example from SQLite wiki
con.execute("create virtual table recipe using fts3(name, ingredients)")
con.executescript(""
    insert into recipe (name, ingredients) values ('broccoli stew', 'broccoli peppers cheese tomatoes');
    insert into recipe (name, ingredients) values ('pumpkin stew', 'pumpkin onions garlic celery');
    insert into recipe (name, ingredients) values ('broccoli pie', 'broccoli cheese onions flour');
    insert into recipe (name, ingredients) values ('pumpkin pie', 'pumpkin sugar flour butter');
"")
for row in con.execute("select rowid, name, ingredients from recipe where name match 'pie'"):
    print(row)

load_extension(path)
This routine loads a SQLite extension from a shared library. You have to enable extension loading
with enable_load_extension() before you can use this routine.

Loadable extensions are disabled by default. See \(^1\). New in version 3.2.

row_factory
You can change this attribute to a callable that accepts the cursor and the original row as a tuple and
will return the real result row. This way, you can implement more advanced ways of returning results,
such as returning an object that can also access columns by name.

Example:

```python
import sqlite3
def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d

con = sqlite3.connect(':memory:')
con.row_factory = dict_factory
cur = con.cursor()
cur.execute("select 1 as a")
print(cur.fetchone()['a'])
```

If returning a tuple doesn’t suffice and you want name-based access to columns, you should consider
setting row_factory to the highly-optimized sqlite3.Row type. Row provides both index-based
and case-insensitive name-based access to columns with almost no memory overhead. It will probably
be better than your own custom dictionary-based approach or even a db_row based solution.

text_factory
Using this attribute you can control what objects are returned for the TEXT data type. By default, this
attribute is set to str and the sqlite3 module will return Unicode objects for TEXT. If you want to
return byte strings instead, you can set it to bytes.
For efficiency reasons, there’s also a way to return \texttt{str} objects only for non-ASCII data, and \texttt{bytes} otherwise. To activate it, set this attribute to \texttt{sqlite3.OptimizedUnicode}.

You can also set it to any other callable that accepts a single bytestring parameter and returns the resulting object.

See the following example code for illustration:

```python
import sqlite3

con = sqlite3.connect(":\memory:")
cur = con.cursor()

AUSTRIA = "\xd6sterreich"

# by default, rows are returned as Unicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert row[0] == AUSTRIA

# but we can make sqlite3 always return bytestrings ...
con.text_factory = bytes
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is bytes
# the bytestrings will be encoded in UTF-8, unless you stored garbage in the
# database ...
assert row[0] == AUSTRIA.encode("utf-8")

# we can also implement a custom text_factory ...
# here we implement one that appends "foo" to all strings
con.text_factory = lambda x: x.decode("utf-8") + "foo"
cur.execute("select ?", ("bar",))
row = cur.fetchone()
assert row[0] == "barfoo"
```

\textbf{total\_changes}

Returns the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.

\textbf{iterdump}

Returns an iterator to dump the database in an SQL text format. Useful when saving an in-memory database for later restoration. This function provides the same capabilities as the \texttt{.dump} command in the \texttt{sqlite3} shell.

Example:

```python
# Convert file existing\_db.db to SQL dump file dump.sql
import sqlite3, os

con = sqlite3.connect('existing\_db.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
```

\subsection{12.6.3 Cursor Objects}

A \texttt{Cursor} instance has the following attributes and methods.
execute(sql[, parameters])

Executes an SQL statement. The SQL statement may be parametrized (i.e. placeholders instead of SQL literals). The sqlite3 module supports two kinds of placeholders: question marks (qmark style) and named placeholders (named style).

Here’s an example of both styles:

```python
import sqlite3

con = sqlite3.connect(':memory:)
cur = con.cursor()
cur.execute("create table people (name_last, age)"

who = "Yeltsin"
age = 72

# This is the qmark style:
cur.execute("insert into people values (?, ?)", (who, age))

# And this is the named style:
cur.execute("select * from people where name_last=:who and age=:age", {"who": who, "age": age})

print(cur.fetchone())
```

execute() will only execute a single SQL statement. If you try to execute more than one statement with it, it will raise a Warning. Use executescript() if you want to execute multiple SQL statements with one call.

executemany(sql, seq_of_parameters)

Executes an SQL command against all parameter sequences or mappings found in the sequence sql. The sqlite3 module also allows using an iterator yielding parameters instead of a sequence.

```python
import sqlite3
class IterChars:
    def __init__(self):
        self.count = ord('a')

    def __iter__(self):
        return self

    def __next__(self):
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),) # this is a 1-tuple

con = sqlite3.connect(':memory:"
cur = con.cursor()
cur.execute("create table characters(c)"

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)

cur.execute("select c from characters")
print(cur.fetchall())
```

Here’s a shorter example using a generator:
import sqlite3
import string

def char_generator():
    for c in string.ascii_lowercase:
        yield (c,)

con = sqlite3.connect(':memory:)
cur = con.cursor()
cur.execute("create table characters(c)"

cur.executemany("insert into characters(c) values (?)", char_generator())

cur.execute("select c from characters")
print (cur.fetchall())

executemethod\nThis is a nonstandard convenience method for executing multiple SQL statements at once. It issues a
COMMIT statement first, then executes the SQL script it gets as a parameter.

sql_script can be an instance of str or bytes.
Example:

import sqlite3

con = sqlite3.connect(':memory:)
cur = con.cursor()
cur.executescript(""
create table person(      
    firstname,     
    lastname,  
    age
    
    create table book(        
        title,        
        author,     
        published
        
        insert into book(title, author, published)
        values {
            'Dirk Gently’s Holistic Detective Agency',
            'Douglas Adams',
            1987
        }
        
fetchone()

Fetches the next row of a query result set, returning a single sequence, or None when no more data is
available.

fetchmany(size=cursor.arraysize)

Fetches the next set of rows of a query result, returning a list. An empty list is returned when no more
rows are available.

The number of rows to fetch per call is specified by the size parameter. If it is not given, the cursor’s
arraysize determines the number of rows to be fetched. The method should try to fetch as many rows
as indicated by the size parameter. If this is not possible due to the specified number of rows not being
available, fewer rows may be returned.

Note there are performance considerations involved with the size parameter. For optimal performance,
it is usually best to use the arraysize attribute. If the size parameter is used, then it is best for it to retain
the same value from one fetchmany() call to the next.

fetchall()
Fetches all (remaining) rows of a query result, returning a list. Note that the cursor’s arraysize attribute
can affect the performance of this operation. An empty list is returned when no rows are available.

rowcount
Although the Cursor class of the sqlite3 module implements this attribute, the database engine’s
own support for the determination of “rows affected”/“rows selected” is quirky.

For executemany() statements, the number of modifications are summed up into rowcount.

As required by the Python DB API Spec, the rowcount attribute “is -1 in case no executeXX() has been performed on the cursor or the rowcount of the last operation is not determinable by the
interface”. This includes SELECT statements because we cannot determine the number of rows a
query produced until all rows were fetched.

With SQLite versions before 3.6.5, rowcount is set to 0 if you make a DELETE FROM table
without any condition.

class sqlite3.Row
A Row instance serves as a highly optimized row_factory for Connection objects. It tries to mimic
a tuple in most of its features.

It supports mapping access by column name and index, iteration, representation, equality testing and
len().

If two Row objects have exactly the same columns and their members are equal, they compare equal.

keys()
This method returns a tuple of column names. Immediately after a query, it is the first member of each
tuple in Cursor.description.

Let’s assume we initialize a table as in the example given above:

```python
c = conn.cursor()
c.execute('''create table stocks
(date text, trans text, symbol text,
  qty real, price real)''')
c.execute('''insert into stocks
VALUES ('2006-01-05','BUY','RHAT',100,35.14)'''
```
>>> conn.row_factory = sqlite3.Row
>>> c = conn.cursor()
>>> c.execute('select * from stocks')
<sqlite3.Cursor object at 0x7f4e7dd8fa80>
>>> r = c.fetchone()
>>> type(r)
<class 'sqlite3.Row'>
>>> tuple(r)
(2006-01-05, 'BUY', 'RHAT', 100.0, 35.14)
>>> len(r)
5
>>> r[2]
'RHAT'
>>> r.keys()
['date', 'trans', 'symbol', 'qty', 'price']
>>> r['qty']
100.0
>>> for member in r:
...    print(member)
...
2006-01-05
BUY
RHAT
100.0
35.14

12.6.5 SQLite and Python types

Introduction

SQLite natively supports the following types: NULL, INTEGER, REAL, TEXT, BLOB.

The following Python types can thus be sent to SQLite without any problem:

<table>
<thead>
<tr>
<th>Python type</th>
<th>SQLite type</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>NULL</td>
</tr>
<tr>
<td>int</td>
<td>INTEGER</td>
</tr>
<tr>
<td>float</td>
<td>REAL</td>
</tr>
<tr>
<td>str</td>
<td>TEXT</td>
</tr>
<tr>
<td>bytes</td>
<td>BLOB</td>
</tr>
</tbody>
</table>

This is how SQLite types are converted to Python types by default:

<table>
<thead>
<tr>
<th>SQLite type</th>
<th>Python type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>None</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>REAL</td>
<td>float</td>
</tr>
<tr>
<td>TEXT</td>
<td>depends on text_factory, str by default</td>
</tr>
<tr>
<td>BLOB</td>
<td>bytes</td>
</tr>
</tbody>
</table>

The type system of the sqlite3 module is extensible in two ways: you can store additional Python types in a SQLite database via object adaptation, and you can let the sqlite3 module convert SQLite types to different Python types via converters.

Using adapters to store additional Python types in SQLite databases

As described before, SQLite supports only a limited set of types natively. To use other Python types with SQLite, you must adapt them to one of the sqlite3 module’s supported types for SQLite: one of NoneType, int, float, str, bytes.
The SQLite module uses Python object adaptation, as described in
PEP 246 for this. The protocol to use is PrepareProtocol.

There are two ways to enable the SQLite module to adapt a custom Python type to one of the supported ones.

Letting your object adapt itself

This is a good approach if you write the class yourself. Let’s suppose you have a class like this:

```python
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
```

Now you want to store the point in a single SQLite column. First you’ll have to choose one of the supported types first to be used for representing the point. Let’s just use str and separate the coordinates using a semicolon. Then you need to give your class a method `__conform__(self, protocol)` which must return the converted value. The parameter `protocol` will be `PrepareProtocol`.

```python
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return "%.f;%.f" % (self.x, self.y)
```

```python
con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])
```

Registering an adapter callable

The other possibility is to create a function that converts the type to the string representation and register the function with `register_adapter()`.

```python
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def adapt_point(point):
        return "%.f;%.f" % (point.x, point.y)

sqlite3.register_adapter(Point, adapt_point)
```

```python
con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])
```
The `sqlite3` module has two default adapters for Python’s built-in `datetime.date` and `datetime.datetime` types. Now let’s suppose we want to store `datetime.datetime` objects not in ISO representation, but as a Unix timestamp.

```python
import sqlite3
import datetime
import time

def adapt_datetime(ts):
    return time.mktime(ts.timetuple())

sqlite3.register_adapter(datetime.datetime, adapt_datetime)

con = sqlite3.connect(':memory:)
cur = con.cursor()

now = datetime.datetime.now()
cur.execute("select ?", (now,))
print(cur.fetchone()[0])
```

### Converting SQLite values to custom Python types

Writing an adapter lets you send custom Python types to SQLite. But to make it really useful we need to make the Python to SQLite to Python roundtrip work.

Enter converters.

Let’s go back to the `Point` class. We stored the x and y coordinates separated via semicolons as strings in SQLite. First, we’ll define a converter function that accepts the string as a parameter and constructs a `Point` object from it.

**Note:** Converter functions always get called with a string, no matter under which data type you sent the value to SQLite.

```python
def convert_point(s):
    x, y = map(float, s.split(b";"))
    return Point(x, y)
```

Now you need to make the `sqlite3` module know that what you select from the database is actually a point. There are two ways of doing this:

- Implicitly via the declared type
- Explicitly via the column name

Both ways are described in section Module functions and constants, in the entries for the constants `PARSE_DECLTYPES` and `PARSE_COLNAMES`.

The following example illustrates both approaches.

```python
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "(%f;%f)" % (self.x, self.y)

    def adapt_point(point):
        return ("%f;%f" % (point.x, point.y)).encode('ascii')
```
```python
def convert_point(s):
    x, y = list(map(float, s.split(b";")))
    return Point(x, y)

# Register the adapter
sqlite3.register_adapter(Point, adapt_point)

# Register the converter
sqlite3.register_converter("point", convert_point)

p = Point(4.0, -3.2)

# 1) Using declared types
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(p point)"
)

cur.execute("insert into test(p) values (?)", (p,))
print("with declared types:", cur.fetchone()[0])
cur.close()
con.close()

# 1) Using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(p)"
)

cur.execute("insert into test(p) values (?)", (p,))
cur.execute('select p as "p [point]" from test')
print("with column names:", cur.fetchone()[0])
cur.close()
con.close()

Default adapters and converters

There are default adapters for the date and datetime types in the datetime module. They will be sent as ISO
dates/ISO timestamps to SQLite.
The default converters are registered under the name “date” for `datetime.date` and under the name “tim-
estamp” for `datetime.datetime`.
This way, you can use date/timestamps from Python without any additional fiddling in most cases. The format of
the adapters is also compatible with the experimental SQLite date/time functions.
The following example demonstrates this.

```
cur.execute("insert into test(d, ts) values (?, ?)", (today, now))
cur.execute("select d, ts from test")
row = cur.fetchone()
print(today, "=>", row[0], type(row[0]))
print(now, "=>", row[1], type(row[1]))

cur.execute('select current_date as "d [date]", current_timestamp as "ts [timestamp]"')
row = cur.fetchone()
print("current_date", row[0], type(row[0]))
print("current_timestamp", row[1], type(row[1]))

If a timestamp stored in SQLite has a fractional part longer than 6 numbers, its value will be truncated to microsecond precision by the timestamp converter.

### 12.6.6 Controlling Transactions

By default, the sqlite3 module opens transactions implicitly before a Data Modification Language (DML) statement (i.e. INSERT/UPDATE/DELETE/REPLACE), and commits transactions implicitly before a non-DML, non-query statement (i.e. anything other than SELECT or the aforementioned).

So if you are within a transaction and issue a command like CREATE TABLE ..., VACUUM, PRAGMA, the sqlite3 module will commit implicitly before executing that command. There are two reasons for doing that. The first is that some of these commands don’t work within transactions. The other reason is that sqlite3 needs to keep track of the transaction state (if a transaction is active or not). The current transaction state is exposed through the Connection.in_transaction attribute of the connection object.

You can control which kind of BEGIN statements sqlite3 implicitly executes (or none at all) via the isolation_level parameter to the connect() call, or via the isolation_level property of connections.

If you want autocommit mode, then set isolation_level to None.

Otherwise leave it at its default, which will result in a plain “BEGIN” statement, or set it to one of SQLite’s supported isolation levels: “DEFERRED”, “IMMEDIATE” or “EXCLUSIVE”.

### 12.6.7 Using sqlite3 efficiently

#### Using shortcut methods

Using the nonstandard execute(), executemany() and executescript() methods of the Connection object, your code can be written more concisely because you don’t have to create the (often superfluous) Cursor objects explicitly. Instead, the Cursor objects are created implicitly and these shortcut methods return the cursor objects. This way, you can execute a SELECT statement and iterate over it directly using only a single call on the Connection object.

```python
import sqlite3

persons = [
    ("Hugo", "Boss"),
    ("Calvin", "Klein")
]

con = sqlite3.connect(":memory:")

# Create the table
con.execute("create table person(firstname, lastname)")

# Fill the table
con.executemany("insert into person(firstname, lastname) values (?, ?)", persons)
```
# Print the table contents
    for row in con.execute("select firstname, lastname from person"): 
        print(row)

    print("I just deleted", con.execute("delete from person").rowcount, "rows")

Accessing columns by name instead of by index

One useful feature of the sqlite3 module is the built-in sqlite3.Row class designed to be used as a row factory.

Rows wrapped with this class can be accessed both by index (like tuples) and case-insensitively by name:

```python
import sqlite3
con = sqlite3.connect(":memory:")
con.row_factory = sqlite3.Row

cur = con.cursor()
cur.execute("select 'John' as name, 42 as age")
    for row in cur:
        assert row[0] == row["name"]
        assert row["name"] == row["nAmE"]
        assert row[1] == row["age"]
        assert row[1] == row["AgE"]
```

Using the connection as a context manager

Connection objects can be used as context managers that automatically commit or rollback transactions. In the event of an exception, the transaction is rolled back; otherwise, the transaction is committed:

```python
import sqlite3
con = sqlite3.connect(":memory:")

# Successful, con.commit() is called automatically afterwards
    with con:
        con.execute("create table person (id integer primary key, firstname varchar unique)")

# con.rollback() is called after the with block finishes with an exception, the exception is still raised and must be caught
    try:
        with con:
            con.execute("insert into person(firstname) values (?)", ("Joe",))
    except sqlite3.IntegrityError:
        print("couldn’t add Joe twice")
```

12.6.8 Common issues

Multithreading

Older SQLite versions had issues with sharing connections between threads. That’s why the Python module disallows sharing connections and cursors between threads. If you still try to do so, you will get an exception at runtime.

The only exception is calling the `interrupt()` method, which only makes sense to call from a different thread.
The modules described in this chapter support data compression with the zlib, gzip, bzip2 and lzma algorithms, and the creation of ZIP- and tar-format archives. See also Archiving operations provided by the shutil module.

13.1 zlib — Compression compatible with gzip

For applications that require data compression, the functions in this module allow compression and decompression, using the zlib library. The zlib library has its own home page at http://www.zlib.net. There are known incompatibilities between the Python module and versions of the zlib library earlier than 1.1.3; 1.1.3 has a security vulnerability, so we recommend using 1.1.4 or later.

zlib’s functions have many options and often need to be used in a particular order. This documentation doesn’t attempt to cover all of the permutations; consult the zlib manual at http://www.zlib.net/manual.html for authoritative information.

For reading and writing .gz files see the gzip module.

The available exception and functions in this module are:

```python
exception zlib.error
    Exception raised on compression and decompression errors.

zlib.adler32(data[, value])
    Computes a Adler-32 checksum of data. (An Adler-32 checksum is almost as reliable as a CRC32 but can be computed much more quickly.) If value is present, it is used as the starting value of the checksum; otherwise, a fixed default value is used. This allows computing a running checksum over the concatenation of several inputs. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.
    Always returns an unsigned 32-bit integer.
```

---

**Note:** To generate the same numeric value across all Python versions and platforms use adler32(data) & 0xffffffff. If you are only using the checksum in packed binary format this is not necessary as the return value is the correct 32bit binary representation regardless of sign.

```python
zlib.compress(data[, level])
    Compresses the bytes in data, returning a bytes object containing compressed data. level is an integer from 0 to 9 controlling the level of compression; 1 is fastest and produces the least compression, 9 is slowest and produces the most. 0 is no compression. The default value is 6. Raises the error exception if any error occurs.
```
** zlib.compressobj**(level=-1, method=DEFLATED, wbits=15, memlevel=8, strategy=Z_DEFAULT_STRATEGY, zdict=None)

Returns a compression object, to be used for compressing data streams that won’t fit into memory at once.

*level* is the compression level – an integer from 0 to 9. A value of 1 is fastest and produces the least compression, while a value of 9 is slowest and produces the most. 0 is no compression. The default value is 6.

*method* is the compression algorithm. Currently, the only supported value is DEFLATED.

*wbits* is the base two logarithm of the size of the window buffer. This should be an integer from 8 to 15. Higher values give better compression, but use more memory.

*memlevel* controls the amount of memory used for internal compression state. Valid values range from 1 to 9. Higher values using more memory, but are faster and produce smaller output.

*strategy* is used to tune the compression algorithm. Possible values are Z_DEFAULT_STRATEGY, Z_FILTERED, and Z_HUFFMAN_ONLY.

*zdict* is a predefined compression dictionary. This is a sequence of bytes (such as a bytes object) containing subsequences that are expected to occur frequently in the data that is to be compressed. Those subsequences that are expected to be most common should come at the end of the dictionary. Changed in version 3.3: Added the *zdict* parameter and keyword argument support.

** zlib.crc32**(data[, value])

Computes a CRC (Cyclic Redundancy Check) checksum of *data*. If *value* is present, it is used as the starting value of the checksum; otherwise, a fixed default value is used. This allows computing a running checksum over the concatenation of several inputs. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.

Always returns an unsigned 32-bit integer.

**Note:** To generate the same numeric value across all Python versions and platforms, use crc32(data) & 0xffffffff. If you are only using the checksum in packed binary format this is not necessary as the return value is the correct 32-bit binary representation regardless of sign.

** zlib.decompress**(data[, wbits[, bufsize]])

Decompresses the bytes in *data*, returning a bytes object containing the uncompressed data. The *wbits* parameter controls the size of the window buffer, and is discussed further below. If *bufsize* is given, it is used as the initial size of the output buffer. Raises the error exception if any error occurs.

The absolute value of *wbits* is the base two logarithm of the size of the history buffer (the “window size”) used when compressing data. Its absolute value should be between 8 and 15 for the most recent versions of the zlib library, larger values resulting in better compression at the expense of greater memory usage. When decompressing a stream, *wbits* must not be smaller than the size originally used to compress the stream; using a too-small value will result in an exception. The default value is therefore the highest value, 15. When *wbits* is negative, the standard gzip header is suppressed.

*bufsize* is the initial size of the buffer used to hold decompressed data. If more space is required, the buffer size will be increased as needed, so you don’t have to get this value exactly right; tuning it will only save a few calls to malloc(). The default size is 16384.

** zlib.decompressobj**(wbits=15[, zdict])

Returns a decompression object, to be used for decompressing data streams that won’t fit into memory at once.

The *wbits* parameter controls the size of the window buffer.

The *zdict* parameter specifies a predefined compression dictionary. If provided, this must be the same dictionary as was used by the compressor that produced the data that is to be decompressed.

**Note:** If *zdict* is a mutable object (such as a bytearray), you must not modify its contents between the call to decompressobj() and the first call to the decompressor’s decompress() method.
Changed in version 3.3: Added the \texttt{zdict} parameter.

Compression objects support the following methods:

\begin{itemize}
  \item \texttt{Compress.compress(\textit{data})}
    \begin{itemize}
      \item Compress \textit{data}, returning a bytes object containing compressed data for at least part of the data in \textit{data}. This data should be concatenated to the output produced by any preceding calls to the \texttt{compress()} method. Some input may be kept in internal buffers for later processing.
    \end{itemize}
  \item \texttt{Compress.flush([\textit{mode}])}
    \begin{itemize}
      \item All pending input is processed, and a bytes object containing the remaining compressed output is returned. \textit{mode} can be selected from the constants \texttt{Z_SYNC_FLUSH}, \texttt{Z_FULL_FLUSH}, or \texttt{Z_FINISH}, defaulting to \texttt{Z_FINISH}, \texttt{Z_SYNC_FLUSH} and \texttt{Z_FULL_FLUSH} allow compressing further bytestrings of data, while \texttt{Z_FINISH} finishes the compressed stream and prevents compressing any more data. After calling \texttt{flush()} with \textit{mode} set to \texttt{Z_FINISH}, the \texttt{compress()} method cannot be called again; the only realistic action is to delete the object.
    \end{itemize}
  \item \texttt{Compress.copy()}
    \begin{itemize}
      \item Returns a copy of the compression object. This can be used to efficiently compress a set of data that share a common initial prefix.
    \end{itemize}
\end{itemize}

Decompression objects support the following methods and attributes:

\begin{itemize}
  \item \texttt{Decompress.unused_data}
    \begin{itemize}
      \item A bytes object which contains any bytes past the end of the compressed data. That is, this remains \texttt{"\_"} until the last byte that contains compression data is available. If the whole bytestring turned out to contain compressed data, this is \texttt{b"\_"}, an empty bytes object.
    \end{itemize}
  \item \texttt{Decompress.unconsumed_tail}
    \begin{itemize}
      \item A bytes object that contains any data that was not consumed by the last \texttt{decompress()} call because it exceeded the limit for the uncompressed data buffer. This data has not yet been seen by the zlib machinery, so you must feed it (possibly with further data concatenated to it) back to a subsequent \texttt{decompress()} method call in order to get correct output.
    \end{itemize}
  \item \texttt{Decompress.eof}
    \begin{itemize}
      \item A boolean indicating whether the end of the compressed data stream has been reached.
      \begin{itemize}
        \item This makes it possible to distinguish between a properly-formed compressed stream, and an incomplete or truncated one. New in version 3.3.
      \end{itemize}
    \end{itemize}
  \item \texttt{Decompress.decompress(\textit{data[, max_length]})}
    \begin{itemize}
      \item Decompress \textit{data}, returning a bytes object containing the uncompressed data corresponding to at least part of the data in \textit{string}. This data should be concatenated to the output produced by any preceding calls to the \texttt{decompress()} method. Some of the input data may be preserved in internal buffers for later processing.
      \begin{itemize}
        \item If the optional parameter \textit{max_length} is supplied then the return value will be no longer than \textit{max_length}. This may mean that not all of the compressed input can be processed; and uncompressed data will be stored in the attribute \texttt{unconsumed_tail}. This bytestring must be passed to a subsequent call to \texttt{decompress()} if decompression is to continue. If \textit{max_length} is not supplied then the whole input is decompressed, and \texttt{unconsumed_tail} is empty.
      \end{itemize}
    \end{itemize}
  \item \texttt{Decompress.flush([\textit{length}])}
    \begin{itemize}
      \item All pending input is processed, and a bytes object containing the remaining uncompressed output is returned. After calling \texttt{flush()}, the \texttt{decompress()} method cannot be called again; the only realistic action is to delete the object.
      \begin{itemize}
        \item The optional parameter \textit{length} sets the initial size of the output buffer.
      \end{itemize}
    \end{itemize}
  \item \texttt{Decompress.copy()}
    \begin{itemize}
      \item Returns a copy of the decompression object. This can be used to save the state of the decompressor midway through the data stream in order to speed up random seeks into the stream at a future point.
    \end{itemize}
\end{itemize}

Information about the version of the zlib library in use is available through the following constants:
The Python Library Reference, Release 3.3.3

zlib.ZLIB_VERSION
The version string of the zlib library that was used for building the module. This may be different from the zlib library actually used at runtime, which is available as ZLIB_RUNTIME_VERSION.

zlib.ZLIB_RUNTIME_VERSION
The version string of the zlib library actually loaded by the interpreter. New in version 3.3.

See Also:
Module gzip Reading and writing gzip-format files.
http://www.zlib.net The zlib library home page.
http://www.zlib.net/manual.html The zlib manual explains the semantics and usage of the library’s many functions.

13.2 gzip — Support for gzip files

Source code: Lib/gzip.py

This module provides a simple interface to compress and decompress files just like the GNU programs gzip and gunzip would.

The data compression is provided by the zlib module.

The gzip module provides the GzipFile class, as well as the open(), compress() and decompress() convenience functions. The GzipFile class reads and writes gzip-format files, automatically compressing or decompressing the data so that it looks like an ordinary file object.

Note that additional file formats which can be decompressed by the gzip and gunzip programs, such as those produced by compress and pack, are not supported by this module.

The module defines the following items:

gzip.open (filename=None, mode='rb', compresslevel=9, encoding=None, errors=None, newline=None)
Open a gzip-compressed file in binary or text mode, returning a file object.

The filename argument can be an actual filename (a str or bytes object), or an existing file object to read from or write to.

The mode argument can be any of 'r', 'rb', 'a', 'ab', 'w', or 'wb' for binary mode, or 'rt', 'at', or 'wt' for text mode. The default is 'rb'.

The compresslevel argument is an integer from 0 to 9, as for the GzipFile constructor.

For binary mode, this function is equivalent to the GzipFile constructor: GzipFile(filename, mode, compresslevel). In this case, the encoding, errors and newline arguments must not be provided.

For text mode, a GzipFile object is created, and wrapped in an io.TextIOWrapper instance with the specified encoding, error handling behavior, and line ending(s). Changed in version 3.3: Added support for filename being a file object, support for text mode, and the encoding, errors and newline arguments.

class gzip.GzipFile (filename=None, mode=None, compresslevel=9, fileobj=None, mtime=None)
Constructor for the GzipFile class, which simulates most of the methods of a file object, with the exception of the truncate() method. At least one of fileobj and filename must be given a non-trivial value.

The new class instance is based on fileobj, which can be a regular file, a io.BytesIO object, or any other object which simulates a file. It defaults to None, in which case filename is opened to provide a file object.

When fileobj is not None, the filename argument is only used to be included in the gzip file header, which may include the original filename of the uncompressed file. It defaults to None, in which case filename is opened to provide a file object.

When fileobj is not None, the filename argument is only used to be included in the gzip file header, which may include the original filename of the uncompressed file. It defaults to None, in which case filename is opened to provide a file object.
The *mode* argument can be any of ‘r’, ‘rb’, ‘a’, ‘ab’, ‘w’, or ‘wb’, depending on whether the file will be read or written. The default is the mode of *fileobj* if discernible; otherwise, the default is ‘rb’.

Note that the file is always opened in binary mode. To open a compressed file in text mode, use `open()` (or wrap your `GzipFile` with an `io.TextIOWrapper`).

The *compresslevel* argument is an integer from 0 to 9 controlling the level of compression; 1 is fastest and produces the least compression, and 9 is slowest and produces the most compression. 0 is no compression. The default is 9.

The *mtime* argument is an optional numeric timestamp to be written to the stream when compressing. All gzip compressed streams are required to contain a timestamp. If omitted or `None`, the current time is used. This module ignores the timestamp when decompressing; however, some programs, such as `gunzip`, make use of it. The format of the timestamp is the same as that of the return value of `time.time()` and of the `st_mtime` attribute of the object returned by `os.stat()`.

Calling a `GzipFile` object’s `close()` method does not close *fileobj*, since you might wish to append more material after the compressed data. This also allows you to pass a `io.BytesIO` object opened for writing as *fileobj*, and retrieve the resulting memory buffer using the `io.BytesIO` object’s `getvalue()` method.

`GzipFile` supports the `io.BufferedIOBase` interface, including iteration and the `with` statement. Only the `truncate()` method isn’t implemented.

`GzipFile` also provides the following method:

```python
peek ([n])
```

Read *n* uncompressed bytes without advancing the file position. At most one single read on the compressed stream is done to satisfy the call. The number of bytes returned may be more or less than requested. New in version 3.2.

Changed in version 3.1: Support for the `with` statement was added, along with the *mtime* argument.

Changed in version 3.2: Support for zero-padded and unseekable files was added.

Changed in version 3.3: The `io.BufferedIOBase.read1()` method is now implemented.

```python
gzip.compress (data, compresslevel=9)
```

Compress the *data*, returning a `bytes` object containing the compressed data. `compresslevel` has the same meaning as in the `GzipFile` constructor above. New in version 3.2.

```python
gzip.decompress (data)
```

Decompress the *data*, returning a `bytes` object containing the uncompressed data. New in version 3.2.

### 13.2.1 Examples of usage

Example of how to read a compressed file:

```python
import gzip
with gzip.open('/home/joe/file.txt.gz', 'rb') as f:
    file_content = f.read()
```

Example of how to create a compressed GZIP file:

```python
import gzip
content = b"Lots of content here"
with gzip.open('/home/joe/file.txt.gz', 'wb') as f:
    f.write(content)
```

Example of how to GZIP compress an existing file:

```python
import gzip
with open('/home/joe/file.txt', 'rb') as f_in:
    with gzip.open('/home/joe/file.txt.gz', 'wb') as f_out:
        f_out.writelines(f_in)
```

Example of how to GZIP compress a binary string:

```python
```
import gzip
s_in = b"Lots of content here"
s_out = gzip.compress(s_in)

See Also:
Module zlib The basic data compression module needed to support the gzip file format.

13.3 bz2 — Support for bzip2 compression

This module provides a comprehensive interface for compressing and decompressing data using the bzip2 compression algorithm.

The bz2 module contains:

- The `open()` function and `BZ2File` class for reading and writing compressed files.
- The `BZ2Compressor` and `BZ2Decompressor` classes for incremental (de)compression.
- The `compress()` and `decompress()` functions for one-shot (de)compression.

All of the classes in this module may safely be accessed from multiple threads.

13.3.1 (De)compression of files

```python
bz2.open(filename='', mode='r', compresslevel=9, encoding=None, errors=None, newline=None)
```

Open a bzip2-compressed file in binary or text mode, returning a file object.

As with the constructor for `BZ2File`, the `filename` argument can be an actual filename (a str or bytes object), or an existing file object to read from or write to.

The `mode` argument can be any of `r`, `rb`, `w`, `wb`, `a`, or `ab` for binary mode, or `rt`, `wt`, or `at` for text mode. The default is `rb`.

The `compresslevel` argument is an integer from 1 to 9, as for the `BZ2File` constructor.

For binary mode, this function is equivalent to the `BZ2File` constructor: `BZ2File(filename, mode, compresslevel=compresslevel)`. In this case, the `encoding`, `errors` and `newline` arguments must not be provided.

For text mode, a `BZ2File` object is created, and wrapped in an `io.TextIOWrapper` instance with the specified encoding, error handling behavior, and line ending(s). New in version 3.3.

```python
class bz2.BZ2File(filename='', mode='r', buffering=None, compresslevel=9)
```

Open a bzip2-compressed file in binary mode.

If `filename` is a str or bytes object, open the named file directly. Otherwise, `filename` should be a file object, which will be used to read or write the compressed data.

The `mode` argument can be either `r` for reading (default), `w` for overwriting, or `a` for appending. These can equivalently be given as `rb`, `wb`, and `ab` respectively.

If `filename` is a file object (rather than an actual file name), a mode of `w` does not truncate the file, and is instead equivalent to `a`.

The `buffering` argument is ignored. Its use is deprecated.

If `mode` is `w` or `a`, `compresslevel` can be a number between 1 and 9 specifying the level of compression: 1 produces the least compression, and 9 (default) produces the most compression.

If `mode` is `r`, the input file may be the concatenation of multiple compressed streams.

`BZ2File` provides all of the members specified by the `io.BufferedIOBase`, except for `detach()` and `truncate()`. Iteration and the with statement are supported.

`BZ2File` also provides the following method:
peek([n])

Return buffered data without advancing the file position. At least one byte of data will be returned
(unless at EOF). The exact number of bytes returned is unspecified. New in version 3.3.

Changed in version 3.1: Support for the with statement was added. Changed in version 3.3: The
fileno(), readable(), seekable(), writable(), read(), and readinto() methods were
added. Changed in version 3.3: Support was added for filename being a file object instead of an actual
filename. Changed in version 3.3: The ’a’ (append) mode was added, along with support for reading multi-
stream files.

13.3.2 Incremental (de)compression

class bz2.BZ2Compressor (compresslevel=9)

Create a new compressor object. This object may be used to compress data incrementally. For one-shot
compression, use the compress() function instead.

compresslevel, if given, must be a number between 1 and 9. The default is 9.

compress(data)

Provide data to the compressor object. Returns a chunk of compressed data if possible, or an empty
byte string otherwise.

When you have finished providing data to the compressor, call the flush() method to finish the
compression process.

flush()

Finish the compression process. Returns the compressed data left in internal buffers.

The compressor object may not be used after this method has been called.

class bz2.BZ2Decompressor

Create a new decompressor object. This object may be used to decompress data incrementally. For one-shot
compression, use the decompress() function instead.

Note: This class does not transparently handle inputs containing multiple compressed streams,
unlike decompress() and BZ2File. If you need to decompress a multi-stream input with
BZ2Decompressor, you must use a new decompressor for each stream.

decompress(data)

Provide data to the decompressor object. Returns a chunk of decompressed data if possible, or an empty
byte string otherwise.

Attempting to decompress data after the end of the current stream is reached raises an EOFError. If
any data is found after the end of the stream, it is ignored and saved in the unused_data attribute.

eof

True if the end-of-stream marker has been reached. New in version 3.3.

unused_data

Data found after the end of the compressed stream.

If this attribute is accessed before the end of the stream has been reached, its value will be b"".

13.3.3 One-shot (de)compression

bz2.compress(data, compresslevel=9)

Compress data.

compresslevel, if given, must be a number between 1 and 9. The default is 9.

For incremental compression, use a BZ2Compressor instead.
bz2.decompress(data)
Decompress data.

If data is the concatenation of multiple compressed streams, decompress all of the streams.

For incremental decompression, use a BZ2Decompressor instead. Changed in version 3.3: Support for multi-stream inputs was added.

13.4  lzma — Compression using the LZMA algorithm

New in version 3.3. This module provides classes and convenience functions for compressing and decompressing data using the LZMA compression algorithm. Also included is a file interface supporting the .xz and legacy .lzma file formats used by the xz utility, as well as raw compressed streams.

The interface provided by this module is very similar to that of the bz2 module. However, note that LZMAFile is not thread-safe, unlike bz2.BZ2File, so if you need to use a single LZMAFile instance from multiple threads, it is necessary to protect it with a lock.

exception lzma.LZMAError
This exception is raised when an error occurs during compression or decompression, or while initializing the compressor/decompressor state.

13.4.1  Reading and writing compressed files

lzma.open(filename, mode="rb", *, format=None, check=-1, preset=None, filters=None, encoding=None, errors=None, newline=None)
Open an LZMA-compressed file in binary or text mode, returning a file object.

The filename argument can be either an actual file name (given as a str or bytes object), in which case the named file is opened, or it can be an existing file object to read from or write to.

The mode argument can be any of "r", "rb", "w", "wb", "a" or "ab" for binary mode, or "rt", "wt", or "at" for text mode. The default is "rb".

When opening a file for reading, the format and filters arguments have the same meanings as for LZMADecompressor. In this case, the check and preset arguments should not be used.

When opening a file for writing, the format, check, preset and filters arguments have the same meanings as for LZMACompressor.

For binary mode, this function is equivalent to the LZMAFile constructor: LZMAFile(filename, mode, ...). In this case, the encoding, errors and newline arguments must not be provided.

For text mode, a LZMAFile object is created, and wrapped in an io.TextIOWrapper instance with the specified encoding, error handling behavior, and line ending(s).

class lzma.LZMAFile(filename=None, mode="r", *, format=None, check=-1, preset=None, filters=None)
Open an LZMA-compressed file in binary mode.

An LZMAFile can wrap an already-open file object, or operate directly on a named file. The filename argument specifies either the file object to wrap, or the name of the file to open (as a str or bytes object). When wrapping an existing file object, the wrapped file will not be closed when the LZMAFile is closed.

The mode argument can be either "r" for reading (default), "w" for overwriting, or "a" for appending. These can equivalently be given as "rb", "wb", and "ab" respectively.

If filename is a file object (rather than an actual file name), a mode of "w" does not truncate the file, and is instead equivalent to "a".

When opening a file for reading, the input file may be the concatenation of multiple separate compressed streams. These are transparently decoded as a single logical stream.
When opening a file for reading, the `format` and `filters` arguments have the same meanings as for `LZMADecompressor`. In this case, the `check` and `preset` arguments should not be used.

When opening a file for writing, the `format`, `check`, `preset` and `filters` arguments have the same meanings as for `LZMADecompressor`.

`LZMAFile` supports all the members specified by `io.BufferedIOBase`, except for `detach()` and `truncate()`. Iteration and the `with` statement are supported.

The following method is also provided:

```python
peek(size=-1)
```

Return buffered data without advancing the file position. At least one byte of data will be returned, unless EOF has been reached. The exact number of bytes returned is unspecified (the `size` argument is ignored).

### 13.4.2 Compressing and decompressing data in memory

```python
class lzma.LZMACompressor (format=FORMAT_XZ, check=-1, preset=None, filters=None)
```

Create a compressor object, which can be used to compress data incrementally.

For a more convenient way of compressing a single chunk of data, see `compress()`.

The `format` argument specifies what container format should be used. Possible values are:

- **FORMAT_XZ**: The `.xz` container format. This is the default format.
- **FORMAT_ALONE**: The legacy `.lzma` container format. This format is more limited than `.xz` – it does not support integrity checks or multiple filters.
- **FORMAT_RAW**: A raw data stream, not using any container format. This format specifier does not support integrity checks, and requires that you always specify a custom filter chain (for both compression and decompression). Additionally, data compressed in this manner cannot be decompressed using FORMAT_AUTO (see `LZMADecompressor`).

The `check` argument specifies the type of integrity check to include in the compressed data. This check is used when decompressing, to ensure that the data has not been corrupted. Possible values are:

- **CHECK_NONE**: No integrity check. This is the default (and the only acceptable value) for FORMAT_ALONE and FORMAT_RAW.
- **CHECK_CRC32**: 32-bit Cyclic Redundancy Check.
- **CHECK_CRC64**: 64-bit Cyclic Redundancy Check. This is the default for FORMAT_XZ.
- **CHECK_SHA256**: 256-bit Secure Hash Algorithm.

If the specified check is not supported, an `LZMAError` is raised.

The compression settings can be specified either as a preset compression level (with the `preset` argument), or in detail as a custom filter chain (with the `filters` argument).

The `preset` argument (if provided) should be an integer between 0 and 9 (inclusive), optionally OR-ed with the constant PRESET_EXTREME. If neither `preset` nor `filters` are given, the default behavior is to use PRESET_DEFAULT (preset level 6). Higher presets produce smaller output, but make the compression process slower.

**Note:** In addition to being more CPU-intensive, compression with higher presets also requires much more memory (and produces output that needs more memory to decompress). With preset 9 for example, the overhead for an `LZMACompressor` object can be as high as 800 MiB. For this reason, it is generally best to stick with the default preset.

The `filters` argument (if provided) should be a filter chain specifier. See *Specifying custom filter chains* for details.
The Python Library Reference, Release 3.3.3

`compress(data)`
Compress data (a bytes object), returning a bytes object containing compressed data for at least part of the input. Some of data may be buffered internally, for use in later calls to `compress()` and `flush()`. The returned data should be concatenated with the output of any previous calls to `compress()`.

`flush()`
Finish the compression process, returning a bytes object containing any data stored in the compressor’s internal buffers.

The compressor cannot be used after this method has been called.

`class lzma.LZMADecompressor(format=FORMAT_AUTO, memlimit=None, filters=None)`
Create a decompressor object, which can be used to decompress data incrementally.

For a more convenient way of decompressing an entire compressed stream at once, see `decompress()`.

The `format` argument specifies the container format that should be used. The default is FORMAT_AUTO, which can decompress both .xz and .lzma files. Other possible values are FORMAT_XZ, FORMAT_ALONE, and FORMAT_RAW.

The `memlimit` argument specifies a limit (in bytes) on the amount of memory that the decompressor can use. When this argument is used, decompression will fail with an `LZMAError` if it is not possible to decompress the input within the given memory limit.

The `filters` argument specifies the filter chain that was used to create the stream being decompressed. This argument is required if `format` is FORMAT_RAW, but should not be used for other formats. See `Specifying custom filter chains` for more information about filter chains.

**Note:** This class does not transparently handle inputs containing multiple compressed streams, unlike `decompress()` and `LZMAFile`. To decompress a multi-stream input with `LZMADecompressor`, you must create a new decompressor for each stream.

`decompress(data)`
Decompress data (a bytes object), returning a bytes object containing the decompressed data for at least part of the input. Some of data may be buffered internally, for use in later calls to `decompress()`. The returned data should be concatenated with the output of any previous calls to `decompress()`.

`check`
The ID of the integrity check used by the input stream. This may be CHECK_UNKNOWN until enough of the input has been decoded to determine what integrity check it uses.

`eof`
True if the end-of-stream marker has been reached.

`unused_data`
Data found after the end of the compressed stream.

Before the end of the stream is reached, this will be b"".

`lzma.compress(data, format=FORMAT_XZ, check=-1, preset=None, filters=None)`
Compress data (a bytes object), returning the compressed data as a bytes object.

See `LZMACompressor` above for a description of the `format`, `check`, `preset` and `filters` arguments.

`lzma.decompress(data, format=FORMAT_AUTO, memlimit=None, filters=None)`
Decompress data (a bytes object), returning the uncompressed data as a bytes object.

If `data` is the concatenation of multiple distinct compressed streams, decompress all of these streams, and return the concatenation of the results.

See `LZMADecompressor` above for a description of the `format`, `memlimit` and `filters` arguments.

---

Chapter 13. Data Compression and Archiving
13.4.3 Miscellaneous

```
```
lzma.is_check_supported(check)
>
Returns true if the given integrity check is supported on this system.

CHECK_NONE and CHECK_CRC32 are always supported. CHECK_CRC64 and CHECK_SHA256 may be unavailable if you are using a version of liblzma that was compiled with a limited feature set.

13.4.4 Specifying custom filter chains

A filter chain specifier is a sequence of dictionaries, where each dictionary contains the ID and options for a single filter. Each dictionary must contain the key "id", and may contain additional keys to specify filter-dependent options. Valid filter IDs are as follows:

- **Compression filters:**
  - FILTER_LZMA1 (for use with FORMAT_ALONE)
  - FILTER_LZMA2 (for use with FORMAT_XZ and FORMAT_RAW)

- **Delta filter:**
  - FILTER_DELTA

- **Branch-Call-Jump (BCJ) filters:**
  - FILTER_X86
  - FILTER_IA64
  - FILTER_ARM
  - FILTER_ARMTHUMB
  - FILTER_POWERPC
  - FILTER_SPARC

A filter chain can consist of up to 4 filters, and cannot be empty. The last filter in the chain must be a compression filter, and any other filters must be delta or BCJ filters.

Compression filters support the following options (specified as additional entries in the dictionary representing the filter):

- **preset:** A compression preset to use as a source of default values for options that are not specified explicitly.
- **dict_size:** Dictionary size in bytes. This should be between 4 KiB and 1.5 GiB (inclusive).
- **lc:** Number of literal context bits.
- **lp:** Number of literal position bits. The sum lc + lp must be at most 4.
- **pb:** Number of position bits; must be at most 4.
- **mode:** MODE_FAST or MODE_NORMAL.
- **nice_len:** What should be considered a “nice length” for a match. This should be 273 or less.
- **mf:** What match finder to use – MF_HC3, MF_HC4, MF_BT2, MF_BT3, or MF_BT4.
- **depth:** Maximum search depth used by match finder. 0 (default) means to select automatically based on other filter options.

The delta filter stores the differences between bytes, producing more repetitive input for the compressor in certain circumstances. It only supports a single The delta filter supports only one option, dist. This indicates the distance between bytes to be subtracted. The default is 1, i.e. take the differences between adjacent bytes.

The BCJ filters are intended to be applied to machine code. They convert relative branches, calls and jumps in the code to use absolute addressing, with the aim of increasing the redundancy that can be exploited by the filters.
compressor. These filters support one option, start_offset. This specifies the address that should be mapped to the beginning of the input data. The default is 0.

13.4.5 Examples

Reading in a compressed file:

```python
code
import lzma
with lzma.open("file.xz") as f:
    file_content = f.read()
```

Creating a compressed file:

```python
code
import lzma
data = b"Insert Data Here"
with lzma.open("file.xz", "w") as f:
    f.write(data)
```

Compressing data in memory:

```python
code
import lzma
data_in = b"Insert Data Here"
data_out = lzma.compress(data_in)
```

Incremental compression:

```python
code
import lzma
lzc = lzma.LZMACompressor()
out1 = lzc.compress(b"Some data\n")
out2 = lzc.compress(b"Another piece of data\n")
out3 = lzc.compress(b"Even more data\n")
out4 = lzc.flush()
# Concatenate all the partial results:
result = b"".join([out1, out2, out3, out4])
```

Writing compressed data to an already-open file:

```python
code
import lzma
with open("file.xz", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with lzma.open(f, "w") as lzf:
        lzf.write(b"This *will* be compressed\n")
f.write(b"Not compressed\n")
```

Creating a compressed file using a custom filter chain:

```python
code
import lzma
my_filters = [
    {
        "id": lzma.FILTER_DELTA, "dist": 5,
    },
    {
        "id": lzma.FILTER_LZMA2, "preset": 7 | lzma.PRESET_EXTREME,
    }
]
with lzma.open("file.xz", "w", filters=my_filters) as f:
    f.write(b"blah blah blah blah")
```

13.5 zipfile — Work with ZIP archives

Source code: Lib/zipfile.py
The ZIP file format is a common archive and compression standard. This module provides tools to create, read, write, append, and list a ZIP file. Any advanced use of this module will require an understanding of the format, as defined in PKZIP Application Note.

This module does not currently handle multi-disk ZIP files. It can handle ZIP files that use the ZIP64 extensions (that is ZIP files that are more than 4 GiB in size). It supports decryption of encrypted files in ZIP archives, but it currently cannot create an encrypted file. Decryption is extremely slow as it is implemented in native Python rather than C.

The module defines the following items:

**exception zipfile.BadZipFile**
The error raised for bad ZIP files. New in version 3.2.

**exception zipfile.BadZipFile**
Alias of BadZipFile, for compatibility with older Python versions. Deprecated since version 3.2.

**exception zipfile.LargeZipFile**
The error raised when a ZIP file would require ZIP64 functionality but that has not been enabled.

**class zipfile.ZipFile**
The class for reading and writing ZIP files. See section ZipFile Objects for constructor details.

**class zipfile.PyZipFile**
Class for creating ZIP archives containing Python libraries.

**class zipfile.ZipInfo (filename='NoName', date_time=(1980, 1, 1, 0, 0, 0))**
Class used to represent information about a member of an archive. Instances of this class are returned by the getinfo() and infolist() methods of ZipFile objects. Most users of the zipfile module will not need to create these, but only use those created by this module. filename should be the full name of the archive member, and date_time should be a tuple containing six fields which describe the time of the last modification to the file; the fields are described in section ZipInfo Objects.

**zipfile.is_zipfile (filename)**
Returns True if filename is a valid ZIP file based on its magic number, otherwise returns False. filename may be a file or file-like object too. Changed in version 3.1: Support for file and file-like objects.

**zipfile.ZIP_STORED**
The numeric constant for an uncompressed archive member.

**zipfile.ZIP_DEFLATED**
The numeric constant for the usual ZIP compression method. This requires the zlib module.

**zipfile.ZIP_BZIP2**
The numeric constant for the BZIP2 compression method. This requires the bz2 module. New in version 3.3.

**zipfile.ZIP_LZMA**
The numeric constant for the LZMA compression method. This requires the lzma module. New in version 3.3.

**Note:** The ZIP file format specification has included support for bzip2 compression since 2001, and for LZMA compression since 2006. However, some tools (including older Python releases) do not support these compression methods, and may either refuse to process the ZIP file altogether, or fail to extract individual files.

**See Also:**

PKZIP Application Note  Documentation on the ZIP file format by Phil Katz, the creator of the format and algorithms used.

Info-ZIP Home Page  Information about the Info-ZIP project’s ZIP archive programs and development libraries.
13.5.1 ZipFile Objects

class zipfile.ZipFile(file, mode='r', compression=ZIP_STORED, allowZip64=False)

Open a ZIP file, where file can be either a path to a file (a string) or a file-like object. The mode parameter should be ‘r’ to read an existing file, ‘w’ to truncate and write a new file, or ‘a’ to append to an existing file. If mode is ‘a’ and file refers to an existing ZIP file, then additional files are added to it. If file does not refer to a ZIP file, then a new ZIP archive is appended to the file. This is meant for adding a ZIP archive to another file (such as python.exe). If mode is a and the file does not exist at all, it is created.

compression is the ZIP compression method to use when writing the archive, and should be ZIP_STORED, ZIP_DEFLATED, ZIP_BZ2 or ZIP_LZMA; unrecognized values will cause RuntimeError to be raised. If ZIP_DEFLATED, ZIP_BZ2 or ZIP_LZMA is specified but the corresponded module (zlib, bz2 or lzma) is not available, RuntimeError is also raised. The default is ZIP_STORED. If allowZip64 is True zipfile will create ZIP files that use the ZIP64 extensions when the zipfile is larger than 2 GiB. If it is false (the default) zipfile will raise an exception when the ZIP file would require ZIP64 extensions. ZIP64 extensions are disabled by default because the default zip and unzip commands on Unix (the InfoZIP utilities) don’t support these extensions.

If the file is created with mode ‘a’ or ‘w’ and then closed without adding any files to the archive, the appropriate ZIP structures for an empty archive will be written to the file.

ZipFile is also a context manager and therefore supports the with statement. In the example, myzip is closed after the with statement’s suite is finished—even if an exception occurs:

```python
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

New in version 3.2: Added the ability to use ZipFile as a context manager.Changed in version 3.3: Added support for bzip2 and lzma compression.

ZipFile.close()

Close the archive file. You must call close() before exiting your program or essential records will not be written.

ZipFile.getinfo(name)

Return a ZipInfo object with information about the archive member name. Calling getinfo() for a name not currently contained in the archive will raise a KeyError.

ZipFile.infolist()

Return a list containing a ZipInfo object for each member of the archive. The objects are in the same order as their entries in the actual ZIP file on disk if an existing archive was opened.

ZipFile.namelist()

Return a list of archive members by name.

ZipFile.open(name, mode='r', pwd=None)

Extract a member from the archive as a file-like object (ZipExtFile). name is the name of the file in the archive, or a ZipInfo object. The mode parameter, if included, must be one of the following: ‘r’ (the default), ‘U’, or ‘rU’. Choosing ‘U’ or ‘rU’ will enable universal newlines support in the read-only object. pwd is the password used for encrypted files. Calling open() on a closed ZipFile will raise a RuntimeError.

Note: The file-like object is read-only and provides the following methods: read(), readline(), readlines(), __iter__(), __next__().

Note: If the ZipFile was created by passing in a file-like object as the first argument to the constructor, then the object returned by open() shares the ZipFile’s file pointer. Under these circumstances, the object returned by open() should not be used after any additional operations are performed on the ZipFile object.

If the ZipFile was created by passing in a string (the filename) as the first argument to the constructor, then open() will create a new file object that will be held by the ZipExtFile, allowing it to operate independently of the ZipFile.
Note: The open(), read() and extract() methods can take a filename or a ZipInfo object. You will appreciate this when trying to read a ZIP file that contains members with duplicate names.

ZipFile.extract (member, path=None, pwd=None)
Extract a member from the archive to the current working directory; member must be its full name or a ZipInfo object). Its file information is extracted as accurately as possible. path specifies a different directory to extract to. member can be a filename or a ZipInfo object. pwd is the password used for encrypted files.

Note: If a member filename is an absolute path, a drive/UNC sharepoint and leading (back)slashes will be stripped, e.g.: ///foo/bar becomes foo/bar on Unix, and C:\foo\bar becomes foo\bar on Windows. And all ".." components in a member filename will be removed, e.g.: ../foo../ba..r becomes foo../ba..r. On Windows illegal characters (:, <, >, |, "", ?, and *) replaced by underscore (_).

ZipFile.extractall (path=None, members=None, pwd=None)
Extract all members from the archive to the current working directory. path specifies a different directory to extract to. members is optional and must be a subset of the list returned by namelist(). pwd is the password used for encrypted files.

Warning: Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of path, e.g. members that have absolute filenames starting with "/" or filenames with two dots "..".

Changed in version 3.3.1: The zipfile module attempts to prevent that. See extract() note.

ZipFile.printdir ()
Print a table of contents for the archive to sys.stdout.

ZipFile.setpassword (pwd)
Set pwd as default password to extract encrypted files.

ZipFile.read (name, pwd=None)
Return the bytes of the file name in the archive. name is the name of the file in the archive, or a ZipInfo object. The archive must be open for read or append. pwd is the password used for encrypted files and, if specified, it will override the default password set with setpassword(). Calling read() on a closed ZipFile will raise a RuntimeError.

ZipFile.testzip ()
Read all the files in the archive and check their CRC’s and file headers. Return the name of the first bad file, or else return None. Calling testzip() on a closed ZipFile will raise a RuntimeError.

ZipFile.write (filename, arcname=None, compress_type=None)
Write the file named filename to the archive, giving it the archive name arcname (by default, this will be the same as filename, but without a drive letter and with leading path separators removed). If given, compress_type overrides the value given for the compression parameter to the constructor for the new entry. The archive must be open with mode ‘w’ or ‘a’ – calling write() on a ZipFile created with mode ‘t’ will raise a RuntimeError. Calling write() on a closed ZipFile will raise a RuntimeError.

Note: There is no official file name encoding for ZIP files. If you have unicode file names, you must convert them to byte strings in your desired encoding before passing them to write(). WinZip interprets all file names as encoded in CP437, also known as DOS Latin.

Note: Archive names should be relative to the archive root, that is, they should not start with a path separator.
Note: If `arcname` (or `filename`, if `arcname` is not given) contains a null byte, the name of the file in the archive will be truncated at the null byte.

```python
ZipFile.writestr(zinfo_or_arcname, bytes[, compress_type])
```
Write the string `bytes` to the archive; `zinfo_or_arcname` is either the file name it will be given in the archive, or a `ZipInfo` instance. If it’s an instance, at least the filename, date, and time must be given. If it’s a name, the date and time is set to the current date and time. The archive must be opened with mode ‘w’ or ‘a’ – calling `writestr()` on a ZipFile created with mode ‘r’ will raise a `RuntimeError`. Calling `writestr()` on a closed ZipFile will raise a `RuntimeError`.

If given, `compress_type` overrides the value given for the `compression` parameter to the constructor for the new entry, or in the `zinfo_or_arcname` (if that is a `ZipInfo` instance).

Note: When passing a `ZipInfo` instance as the `zinfo_or_arcname` parameter, the compression method used will be that specified in the `compress_type` member of the given `ZipInfo` instance. By default, the `ZipInfo` constructor sets this member to `ZIP_STORED`.

Changed in version 3.2: The `compress_type` argument.

The following data attributes are also available:

```python
ZipFile.debug
```
The level of debug output to use. This may be set from 0 (the default, no output) to 3 (the most output). Debugging information is written to `sys.stdout`.

```python
ZipFile.comment
```
The comment text associated with the ZIP file. If assigning a comment to a `ZipFile` instance created with mode ‘a’ or ‘w’, this should be a string no longer than 65535 bytes. Comments longer than this will be truncated in the written archive when `close()` is called.

### 13.5.2 PyZipFile Objects

The `PyZipFile` constructor takes the same parameters as the `ZipFile` constructor, and one additional parameter, `optimize`.

```python
class zipfile.PyZipFile(file, mode='r', compression=ZIP_STORED, allowZip64=False, optimize=-1)
```
New in version 3.2: The `optimize` parameter. Instances have one method in addition to those of `ZipFile` objects:

```python
writepy(pathname, basename='')
```
Search for files `*.py` and add the corresponding file to the archive.

If the `optimize` parameter to `PyZipFile` was not given or -1, the corresponding file is a `*.pyo` file if available, else a `*.pyc` file, compiling if necessary.

If the `optimize` parameter to `PyZipFile` was 0, 1 or 2, only files with that optimization level (see `compile()` are added to the archive, compiling if necessary.

If the pathname is a file, the filename must end with `.py`, and just the (corresponding `*.py[co]`) file is added at the top level (no path information). If the pathname is a file that does not end with `.py`, a `RuntimeError` will be raised. If it is a directory, and the directory is not a package directory, then all the files `*.py[co]` are added at the top level. If the directory is a package directory, then all `*.py[co]` are added under the package name as a file path, and if any subdirectories are package directories, all of these are added recursively. `basename` is intended for internal use only. The `writepy()` method makes archives with file names like this:

```python
string.pyc  # Top level name
test/__init__.pyc  # Package directory
test/testall.pyc  # Module test.testall
```
13.5.3 ZipInfo Objects

Instances of the `ZipInfo` class are returned by the `getinfo()` and `infolist()` methods of `ZipFile` objects. Each object stores information about a single member of the ZIP archive.

Instances have the following attributes:

`ZipInfo.filename`
Name of the file in the archive.

`ZipInfo.date_time`
The time and date of the last modification to the archive member. This is a tuple of six values:

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Year (&gt;= 1980)</td>
</tr>
<tr>
<td>1</td>
<td>Month (one-based)</td>
</tr>
<tr>
<td>2</td>
<td>Day of month (one-based)</td>
</tr>
<tr>
<td>3</td>
<td>Hours (zero-based)</td>
</tr>
<tr>
<td>4</td>
<td>Minutes (zero-based)</td>
</tr>
<tr>
<td>5</td>
<td>Seconds (zero-based)</td>
</tr>
</tbody>
</table>

**Note:** The ZIP file format does not support timestamps before 1980.

`ZipInfo.compress_type`
Type of compression for the archive member.

`ZipInfo.comment`
Comment for the individual archive member.

`ZipInfo.extra`
Expansion field data. The PKZIP Application Note contains some comments on the internal structure of the data contained in this string.

`ZipInfo.create_system`
System which created ZIP archive.

`ZipInfo.create_version`
PKZIP version which created ZIP archive.

`ZipInfo.extract_version`
PKZIP version needed to extract archive.

`ZipInfo.reserved`
Must be zero.

`ZipInfo.flag_bits`
ZIP flag bits.

`ZipInfo.volume`
Volume number of file header.

`ZipInfo.internal_attr`
Internal attributes.

`ZipInfo.external_attr`
External file attributes.

`ZipInfo.header_offset`
Byte offset to the file header.

`ZipInfo.CRC`
CRC-32 of the uncompressed file.
The Python Library Reference, Release 3.3.3

ZipInfo.compress_size
Size of the compressed data.

ZipInfo.file_size
Size of the uncompressed file.

13.6 tarfile — Read and write tar archive files

Source code: Lib/tarfile.py

The tarfile module makes it possible to read and write tar archives, including those using gzip, bzip2 and lzma compression. Use the zipfile module to read or write .zip files, or the higher-level functions in shutil.

Some facts and figures:

- reads and writes gzip, bzip2 and lzma compressed archives.
- read/write support for the POSIX.1-1988 (ustar) format.
- read/write support for the GNU tar format including longname and longlink extensions, read-only support for all variants of the sparse extension including restoration of sparse files.
- read/write support for the POSIX.1-2001 (pax) format.
- handles directories, regular files, hardlinks, symbolic links, fifos, character devices and block devices and is able to acquire and restore file information like timestamp, access permissions and owner.

Changed in version 3.3: Added support for lzma compression.

```
tarfile.open(name=None, mode='r', fileobj=None, bufsize=10240, **kwargs)
```

Return a TarFile object for the pathname name. For detailed information on TarFile objects and the keyword arguments that are allowed, see TarFile Objects.

mode has to be a string of the form ‘filemode[][compression]’, it defaults to ‘r’. Here is a full list of mode combinations:

<table>
<thead>
<tr>
<th>mode</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘r’ or ‘r:*’</td>
<td>Open for reading with transparent compression (recommended).</td>
</tr>
<tr>
<td>‘r’</td>
<td>Open for reading exclusively without compression.</td>
</tr>
<tr>
<td>‘r:gz’</td>
<td>Open for reading with gzip compression.</td>
</tr>
<tr>
<td>‘r:bz2’</td>
<td>Open for reading with bzip2 compression.</td>
</tr>
<tr>
<td>‘r:xz’</td>
<td>Open for reading with lzma compression.</td>
</tr>
<tr>
<td>‘a’ or ‘a:’</td>
<td>Open for appending with no compression. The file is created if it does not exist.</td>
</tr>
<tr>
<td>‘w’ or ‘w:’</td>
<td>Open for uncompressed writing.</td>
</tr>
<tr>
<td>‘w:gz’</td>
<td>Open for gzip compressed writing.</td>
</tr>
<tr>
<td>‘w:bz2’</td>
<td>Open for bzip2 compressed writing.</td>
</tr>
<tr>
<td>‘w:xz’</td>
<td>Open for lzma compressed writing.</td>
</tr>
</tbody>
</table>

Note that ‘a:gz’, ‘a:bz2’ or ‘a:xz’ is not possible. If mode is not suitable to open a certain (compressed) file for reading, ReadError is raised. Use mode ‘r’ to avoid this. If a compression method is not supported, CompressionError is raised.

If fileobj is specified, it is used as an alternative to a file object opened in binary mode for name. It is supposed to be at position 0.

For special purposes, there is a second format for mode: ‘filemode|[compression]’. tarfile.open() will return a TarFile object that processes its data as a stream of blocks. No random seeking will be done on the file. If given, fileobj may be any object that has a read() or write() method (depending on the mode). bufsize specifies the blocksize and defaults to 20 * 512 bytes. Use this variant in combination with e.g. sys.stdin, a socket file object or a tape device. However, such a TarFile object is limited in that it does not allow to be accessed randomly, see Examples. The currently possible modes:
The Python Library Reference, Release 3.3.3

<table>
<thead>
<tr>
<th>Mode</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘r</td>
<td>*’</td>
</tr>
<tr>
<td>‘r</td>
<td>’</td>
</tr>
<tr>
<td>‘r</td>
<td>gz’</td>
</tr>
<tr>
<td>‘r</td>
<td>bz2’</td>
</tr>
<tr>
<td>‘r</td>
<td>xz’</td>
</tr>
<tr>
<td>‘w</td>
<td>’</td>
</tr>
<tr>
<td>‘w</td>
<td>gz’</td>
</tr>
<tr>
<td>‘w</td>
<td>bz2’</td>
</tr>
<tr>
<td>‘w</td>
<td>xz’</td>
</tr>
</tbody>
</table>

class tarfile.TarFile

Class for reading and writing tar archives. Do not use this class directly, better use `tarfile.open()` instead. See TarFile Objects.

tarfile.is_tarfile(name)

Return True if name is a tar archive file, that the tarfile module can read.

The tarfile module defines the following exceptions:

exception tarfile.TarError

Base class for all tarfile exceptions.

exception tarfile.ReadError

Is raised when a tar archive is opened, that either cannot be handled by the tarfile module or is somehow invalid.

exception tarfile.CompressionError

Is raised when a compression method is not supported or when the data cannot be decoded properly.

exception tarfile.StreamError

Is raised for the limitations that are typical for stream-like TarFile objects.

exception tarfile.ExtractError

Is raised for non-fatal errors when using TarFile.extract(), but only if TarFile.errorlevel== 2.

exception tarfile.HeaderError

Is raised by TarInfo.frombuf() if the buffer it gets is invalid.

Each of the following constants defines a tar archive format that the tarfile module is able to create. See section Supported tar formats for details.

tarfile.USTAR_FORMAT

POSIX.1-1988 (ustar) format.

tarfile.GNU_FORMAT

GNU tar format.

tarfile.PAX_FORMAT

POSIX.1-2001 (pax) format.

tarfile.DEFAULT_FORMAT

The default format for creating archives. This is currently GNU_FORMAT.

The following variables are available on module level:

tarfile.ENCODING

The default character encoding: ‘utf-8’ on Windows, `sys.getfilesystemencoding()` otherwise.

See Also:

Module zipfile Documentation of the zipfile standard module.

GNU tar manual, Basic Tar Format Documentation for tar archive files, including GNU tar extensions.
13.6.1 TarFile Objects

The `TarFile` object provides an interface to a tar archive. A tar archive is a sequence of blocks. An archive member (a stored file) is made up of a header block followed by data blocks. It is possible to store a file in a tar archive several times. Each archive member is represented by a `TarInfo` object, see `TarInfo Objects` for details.

A `TarFile` object can be used as a context manager in a `with` statement. It will automatically be closed when the block is completed. Please note that in the event of an exception an archive opened for writing will not be finalized; only the internally used file object will be closed. See the `Examples` section for a use case. New in version 3.2: Added support for the context manager protocol.

```
class tarfile.TarFile(name=None, mode='r', fileobj=None, format=DEFAULT_FORMAT, 
tarinfo=TarInfo, dereference=False, ignore_zeros=False, encoding=ENCODING, errors='surrogateescape', 
pax_headers=None, debug=0, errorlevel=0)
```

All following arguments are optional and can be accessed as instance attributes as well.

`name` is the pathname of the archive. It can be omitted if `fileobj` is given. In this case, the file object’s `name` attribute is used if it exists.

`mode` is either ‘r’ to read from an existing archive, ‘a’ to append data to an existing file or ‘w’ to create a new file overwriting an existing one.

If `fileobj` is given, it is used for reading or writing data. If it can be determined, `mode` is overridden by `fileobj`’s mode. `fileobj` will be used from position 0.

**Note:** `fileobj` is not closed, when `TarFile` is closed.

`format` controls the archive format. It must be one of the constants `USTAR_FORMAT`, `GNU_FORMAT` or `PAX_FORMAT` that are defined at module level.

The `tarinfo` argument can be used to replace the default `TarInfo` class with a different one.

If `dereference` is `False`, add symbolic and hard links to the archive. If it is `True`, add the content of the target files to the archive. This has no effect on systems that do not support symbolic links.

If `ignore_zeros` is `False`, treat an empty block as the end of the archive. If it is `True`, skip empty (and invalid) blocks and try to get as many members as possible. This is only useful for reading concatenated or damaged archives.

`debug` can be set from 0 (no debug messages) up to 3 (all debug messages). The messages are written to `sys.stderr`.

If `errorlevel` is 0, all errors are ignored when using `TarFile.extract()`. Nevertheless, they appear as error messages in the debug output, when debugging is enabled. If 1, all fatal errors are raised as `OSError` exceptions. If 2, all non-fatal errors are raised as `TarError` exceptions as well.

The `encoding` and `errors` arguments define the character encoding to be used for reading or writing the archive and how conversion errors are going to be handled. The default settings will work for most users. See section `Unicode issues` for in-depth information. Changed in version 3.2: Use ‘surrogateescape’ as the default for the `errors` argument. The `pax_headers` argument is an optional dictionary of strings which will be added as a pax global header if `format` is `PAX_FORMAT`.

`TarFile.open(...)`

Alternative constructor. The `tarfile.open()` function is actually a shortcut to this classmethod.

`TarFile.getmember(name)`

Return a `TarInfo` object for member `name`. If `name` can not be found in the archive, `KeyError` is raised.

**Note:** If a member occurs more than once in the archive, its last occurrence is assumed to be the most up-to-date version.
TarFile.getmembers()
  Return the members of the archive as a list of TarInfo objects. The list has the same order as the members in the archive.

TarFile.getnames()
  Return the members as a list of their names. It has the same order as the list returned by getmembers().

TarFile.list(verbosetrue=True)
  Print a table of contents to sys.stdout. If verbose is False, only the names of the members are printed. If it is True, output similar to that of ls -l is produced.

TarFile.next()
  Return the next member of the archive as a TarInfo object, when TarFile is opened for reading. Return None if there is no more available.

TarFile.extractall(path=".", members=None)
  Extract all members from the archive to the current working directory or directory path. If optional members is given, it must be a subset of the list returned by getmembers(). Directory information like owner, modification time and permissions are set after all members have been extracted. This is done to work around two problems: A directory’s modification time is reset each time a file is created in it. And, if a directory’s permissions do not allow writing, extracting files to it will fail.

Warning: Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of path, e.g. members that have absolute filenames starting with "/" or filenames with two dots ". ..".

TarFile.extract(member, path=", set_atts=True)
  Extract a member from the archive to the current working directory, using its full name. Its file information is extracted as accurately as possible. member may be a filename or a TarInfo object. You can specify a different directory using path. File attributes (owner, nttime, mode) are set unless set_atts is False.

Note: The extract() method does not take care of several extraction issues. In most cases you should consider using the extractall() method.

Warning: See the warning for extractall().

TarFile.extractfile(member)
  Extract a member from the archive as a file object. member may be a filename or a TarInfo object. If member is a regular file or a link, an io.BufferedReader object is returned. Otherwise, None is returned. Changed in version 3.3: Return an io.BufferedReader object.

TarFile.add(name, arcname=None, recursive=True, exclude=None, *, filter=None)
  Add the file name to the archive. name may be any type of file (directory, fifo, symbolic link, etc.). If given, arcname specifies an alternative name for the file in the archive. Directories are added recursively by default. This can be avoided by setting recursive to False. If exclude is given, it must be a function that takes one filename argument and returns a boolean value. Depending on this value the respective file is either excluded (True) or added (False). If filter is specified it must be a keyword argument. It should be a function that takes a TarInfo object argument and returns the changed TarInfo object. If it instead returns None the TarInfo object will be excluded from the archive. See Examples for an example. Changed in version 3.2: Added the filter parameter. Deprecated since version 3.2: The exclude parameter is deprecated, please use the filter parameter instead.

TarFile.addfile(tarinfo, fileobj=None)
  Add the TarInfo object tarinfo to the archive. If fileobj is given, tarinfo.size bytes are read from it and added to the archive. You can create TarInfo objects using gettarinfo().

Note: On Windows platforms, fileobj should always be opened with mode ’rb’ to avoid irritation about the file size.
TarFile.gettarinfo(name=None, arcname=None, fileobj=None)

Create a TarInfo object for either the file name or the file object fileobj (using os.fstat() on its file descriptor). You can modify some of the Tarinfo's attributes before you add it using addfile(). If given, arcname specifies an alternative name for the file in the archive.

TarFile.close()

Close the TarFile. In write mode, two finishing zero blocks are appended to the archive.

TarFile.pax_headers

A dictionary containing key-value pairs of pax global headers.

13.6.2 TarInfo Objects

A TarInfo object represents one member in a TarFile. Aside from storing all required attributes of a file (like file type, size, time, permissions, owner etc.), it provides some useful methods to determine its type. It does not contain the file's data itself.

TarInfo objects are returned by TarFile's methods getmember(), getmembers() and gettarinfo().

class tarfile.TarInfo(name="")

Create a TarInfo object.

TarInfo.frombuf(buf)

Create and return a TarInfo object from string buffer buf.

Raises HeaderError if the buffer is invalid.

TarInfo.fromtarfile(tarfile)

Read the next member from the TarFile object tarfile and return it as a TarInfo object.

TarInfo.tobuf(format=DEFAULT_FORMAT, encoding=ENCODING, errors='surrogateescape')

Create a string buffer from a TarInfo object. For information on the arguments see the constructor of the TarFile class. Changed in version 3.2: Use 'surrogateescape' as the default for the errors argument.

A TarInfo object has the following public data attributes:

TarInfo.name

Name of the archive member.

TarInfo.size

Size in bytes.

TarInfo.mtime

Time of last modification.

TarInfo.mode

Permission bits.

TarInfo.type

File type. type is usually one of these constants: REGTYPE, AREGTYPE, LNKTYPE, SYMTYPE, DIRTYPE, FIFOTYPE, CONTTYPE, CHRTYPE, BLKTYPE, GNUTYPE_SPARSE. To determine the type of a TarInfo object more conveniently, use the is_*() methods below.

TarInfo.linkname

Name of the target file name, which is only present in TarInfo objects of type LNKTYPE and SYMTYPE.

TarInfo.uid

User ID of the user who originally stored this member.

TarInfo.gid

Group ID of the user who originally stored this member.

TarInfo.uname

User name.
A `TarInfo` object also provides some convenient query methods:

- `isfile()`
  - Return `True` if the `TarInfo` object is a regular file.

- `isreg()`
  - Same as `isfile()`.

- `isdir()`
  - Return `True` if it is a directory.

- `issym()`
  - Return `True` if it is a symbolic link.

- `islnk()`
  - Return `True` if it is a hard link.

- `ischr()`
  - Return `True` if it is a character device.

- `isblk()`
  - Return `True` if it is a block device.

- `isfifo()`
  - Return `True` if it is a FIFO.

- `isdev()`
  - Return `True` if it is one of character device, block device or FIFO.

### 13.6.3 Examples

How to extract an entire tar archive to the current working directory:

```python
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall()
tar.close()
```

How to extract a subset of a tar archive with `TarFile.extractall()` using a generator function instead of a list:

```python
import os
import tarfile
def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py":
            yield tarinfo
tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

How to create an uncompressed tar archive from a list of filenames:

```python
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```
The same example using the with statement:

```python
import tarfile
with tarfile.open("sample.tar", "w") as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

How to read a gzip compressed tar archive and display some member information:

```python
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print(tarinfo.name, "is", tarinfo.size, "bytes in size and is", end="")
    if tarinfo.isreg():
        print("a regular file.")
    elif tarinfo.isdir():
        print("a directory.")
    else:
        print("something else.")
tar.close()
```

How to create an archive and reset the user information using the filter parameter in `TarFile.add()`:

```python
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
    tarinfo.uname = tarinfo.gname = "root"
    return tarinfo

tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```

### 13.6.4 Supported tar formats

There are three tar formats that can be created with the `tarfile` module:

- The POSIX.1-1988 ustar format (`USTAR_FORMAT`). It supports filenames up to a length of at best 256 characters and linknames up to 100 characters. The maximum file size is 8 GiB. This is an old and limited but widely supported format.

- The GNU tar format (`GNU_FORMAT`). It supports long filenames and linknames, files bigger than 8 GiB and sparse files. It is the de facto standard on GNU/Linux systems. `tarfile` fully supports the GNU tar extensions for long names, sparse file support is read-only.

- The POSIX.1-2001 pax format (`PAX_FORMAT`). It is the most flexible format with virtually no limits. It supports long filenames and linknames, large files and stores pathnames in a portable way. However, not all tar implementations today are able to handle pax archives properly.

The `pax` format is an extension to the existing `ustar` format. It uses extra headers for information that cannot be stored otherwise. There are two flavours of pax headers: Extended headers only affect the subsequent file header, global headers are valid for the complete archive and affect all following files. All the data in a pax header is encoded in UTF-8 for portability reasons.

There are some more variants of the tar format which can be read, but not created:

- The ancient V7 format. This is the first tar format from Unix Seventh Edition, storing only regular files and directories. Names must not be longer than 100 characters, there is no user/group name information. Some archives have miscalculated header checksums in case of fields with non-ASCII characters.

- The SunOS tar extended format. This format is a variant of the POSIX.1-2001 pax format, but is not compatible.
13.6.5 Unicode issues

The tar format was originally conceived to make backups on tape drives with the main focus on preserving file system information. Nowadays tar archives are commonly used for file distribution and exchanging archives over networks. One problem of the original format (which is the basis of all other formats) is that there is no concept of supporting different character encodings. For example, an ordinary tar archive created on a UTF-8 system cannot be read correctly on a Latin-1 system if it contains non-ASCII characters. Textual metadata (like filenames, linknames, user/group names) will appear damaged. Unfortunately, there is no way to autodetect the encoding of an archive. The pax format was designed to solve this problem. It stores non-ASCII metadata using the universal character encoding UTF-8.

The details of character conversion in `tarfile` are controlled by the `encoding` and `errors` keyword arguments of the `TarFile` class.

`encoding` defines the character encoding to use for the metadata in the archive. The default value is `sys.getfilesystemencoding()` or `'ascii'` as a fallback. Depending on whether the archive is read or written, the metadata must be either decoded or encoded. If `encoding` is not set appropriately, this conversion may fail.

The `errors` argument defines how characters are treated that cannot be converted. Possible values are listed in section Codec Base Classes. The default scheme is `'surrogateescape'` which Python also uses for its file system calls, see File Names, Command Line Arguments, and Environment Variables.

In case of `PAX_FORMAT` archives, `encoding` is generally not needed because all the metadata is stored using UTF-8. `encoding` is only used in the rare cases when binary pax headers are decoded or when strings with surrogate characters are stored.
The modules described in this chapter parse various miscellaneous file formats that aren’t markup languages and are not related to e-mail.

14.1 csv — CSV File Reading and Writing

The so-called CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases. CSV format was used for many years prior to attempts to describe the format in a standardized way in RFC 4180. The lack of a well-defined standard means that subtle differences often exist in the data produced and consumed by different applications. These differences can make it annoying to process CSV files from multiple sources. Still, while the delimiters and quoting characters vary, the overall format is similar enough that it is possible to write a single module which can efficiently manipulate such data, hiding the details of reading and writing the data from the programmer.

The csv module implements classes to read and write tabular data in CSV format. It allows programmers to say, “write this data in the format preferred by Excel,” or “read data from this file which was generated by Excel,” without knowing the precise details of the CSV format used by Excel. Programmers can also describe the CSV formats understood by other applications or define their own special-purpose CSV formats.

The csv module’s reader and writer objects read and write sequences. Programmers can also read and write data in dictionary form using the DictReader and DictWriter classes.

See Also:

PEP 305 - CSV File API The Python Enhancement Proposal which proposed this addition to Python.

14.1.1 Module Contents

The csv module defines the following functions:

`csv.reader (csvfile, dialect='excel', **fmtparams)`

Return a reader object which will iterate over lines in the given csvfile. csvfile can be any object which supports the iterator protocol and returns a string each time its __next__() method is called — file objects and list objects are both suitable. If csvfile is a file object, it should be opened with newline=”.  

An optional dialect parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the Dialect class or one of the strings returned by the list_dialects() function. The other optional fmtparams keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about the dialect and formatting parameters, see section Dialects and Formatting Parameters.

---

1 If newline=“ is not specified, newlines embedded inside quoted fields will not be interpreted correctly, and on platforms that use \r\n linendings on write an extra \r will be added. It should always be safe to specify newline=”, since the csv module does its own (universal) newline handling.
Each row read from the csv file is returned as a list of strings. No automatic data type conversion is performed unless the QUOTE_NONNUMERIC format option is specified (in which case unquoted fields are transformed into floats).

A short usage example:

```python
>>> import csv
>>> with open('eggs.csv', newline='') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
...     for row in spamreader:
...         print(', '.join(row))
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

```python
csv.writer(csvfile, dialect='excel', **fmtparams)
```

Return a writer object responsible for converting the user’s data into delimited strings on the given file-like object. `csvfile` can be any object with a `write()` method. If `csvfile` is a file object, it should be opened with `newline='\n'`. An optional `dialect` parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the `Dialect` class or one of the strings returned by the `list_dialects()` function. The other optional `fmtparams` keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about the dialect and formatting parameters, see section Dialects and Formatting Parameters. To make it as easy as possible to interface with modules which implement the DB API, the value `None` is written as the empty string. While this isn’t a reversible transformation, it makes it easier to dump SQL NULL data values to CSV files without preprocessing the data returned from a `cursor.fetch*` call. All other non-string data are stringified with `str()` before being written.

A short usage example:

```python
import csv
with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                           quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

```python
csv.register_dialect(name[, dialect], **fmtparams)
```

Associate `dialect` with `name`. `name` must be a string. The dialect can be specified either by passing a sub-class of `Dialect`, or by `fmtparams` keyword arguments, or both, with keyword arguments overriding parameters of the dialect. For full details about the dialect and formatting parameters, see section Dialects and Formatting Parameters.

```python
csv.unregister_dialect(name)
```

Delete the dialect associated with `name` from the dialect registry. An `Error` is raised if `name` is not a registered dialect name.

```python
csv.get_dialect(name)
```

Return the dialect associated with `name`. An `Error` is raised if `name` is not a registered dialect name. This function returns an immutable `Dialect`.

```python
csv.list_dialects()
```

Return the names of all registered dialects.

```python
csv.field_size_limit([new_limit])
```

Returns the current maximum field size allowed by the parser. If `new_limit` is given, this becomes the new limit.

The `csv` module defines the following classes:

```python
class csv.DictReader(csvfile, fieldnames=None, restkey=None, restval=None, dialect='excel', *args, **kwargs)
```

Create an object which operates like a regular reader but maps the information read into a dict whose keys...
are given by the optional `fieldnames` parameter. If the `fieldnames` parameter is omitted, the values in the first row of the `csvfile` will be used as the fieldnames. If the row read has more fields than the fieldnames sequence, the remaining data is added as a sequence keyed by the value of `restkey`. If the row read has fewer fields than the fieldnames sequence, the remaining keys take the value of the optional `restval` parameter. Any other optional or keyword arguments are passed to the underlying `reader` instance.

```python
class csv.DictWriter(csvfile, fieldnames, restval='', extrasaction='raise', dialect='excel', *args, **kwds)

Create an object which operates like a regular writer but maps dictionaries onto output rows. The `fieldnames` parameter identifies the order in which values in the dictionary passed to the `writerow()` method are written to the `csvfile`. The optional `restval` parameter specifies the value to be written if the dictionary is missing a key in `fieldnames`. If the dictionary passed to the `writerow()` method contains a key not found in `fieldnames`, the optional `extrasaction` parameter indicates what action to take. If it is set to `’raise’` a `ValueError` is raised. If it is set to `’ignore’`, extra values in the dictionary are ignored. Any other optional or keyword arguments are passed to the underlying `writer` instance.

Note that unlike the `DictReader` class, the `fieldnames` parameter of the `DictWriter` is not optional. Since Python’s `dict` objects are not ordered, there is not enough information available to deduce the order in which the row should be written to the `csvfile`.
```

class csv.Dialect

The `Dialect` class is a container class relied on primarily for its attributes, which are used to define the parameters for a specific `reader` or `writer` instance.

class csv.excel

The `excel` class defines the usual properties of an Excel-generated CSV file. It is registered with the dialect name `’excel’`.

class csv.excel_tab

The `excel_tab` class defines the usual properties of an Excel-generated TAB-delimited file. It is registered with the dialect name `’excel-tab’`.

class csv.unix_dialect

The `unix_dialect` class defines the usual properties of a CSV file generated on UNIX systems, i.e. using `‘\n’` as line terminator and quoting all fields. It is registered with the dialect name `’unix’`. New in version 3.2.

class csv.Sniffer

The `Sniffer` class is used to deduce the format of a CSV file.

The `Sniffer` class provides two methods:

```python
    sniff(sample, delimiters=None)
```

Analyze the given `sample` and return a `Dialect` subclass reflecting the parameters found. If the optional `delimiters` parameter is given, it is interpreted as a string containing possible valid delimiter characters.

```python
    has_header(sample)
```

Analyze the sample text (presumed to be in CSV format) and return `True` if the first row appears to be a series of column headers.

An example for `Sniffer` use:

```python
with open(’example.csv’) as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... process CSV file contents here ...
```

The `csv` module defines the following constants:

```python
csv.QUOTE_ALL
    Instructs `writer` objects to quote all fields.
```
The Python Library Reference, Release 3.3.3

**csv.QUOTE_MINIMAL**
Instructs `writer` objects to only quote those fields which contain special characters such as `delimiter`, `quotechar` or any of the characters in `lineterminator`.

**csv.QUOTE_NONNUMERIC**
Instructs `writer` objects to quote all non-numeric fields.
Instructs the reader to convert all non-quoted fields to type `float`.

**csv.QUOTE_NONE**
Instructs `writer` objects to never quote fields. When the current `delimiter` occurs in output data it is preceded by the current `escapechar` character. If `escapechar` is not set, the writer will raise `csv.Error` if any characters that require escaping are encountered.
Instructs `reader` to perform no special processing of quote characters.

The `csv` module defines the following exception:

**exception csv.Error**
Raised by any of the functions when an error is detected.

### 14.1.2 Dialects and Formatting Parameters

To make it easier to specify the format of input and output records, specific formatting parameters are grouped together into dialects. A dialect is a subclass of the `Dialect` class having a set of specific methods and a single `validate()` method. When creating `reader` or `writer` objects, the programmer can specify a string or a subclass of the `Dialect` class as the dialect parameter. In addition to, or instead of, the `dialect` parameter, the programmer can also specify individual formatting parameters, which have the same names as the attributes defined below for the `Dialect` class.

Dialects support the following attributes:

**Dialect.delimiter**
A one-character string used to separate fields. It defaults to ’,’.

**Dialect.doublequote**
Controls how instances of `quotechar` appearing inside a field should be themselves be quoted. When `True`, the character is doubled. When `False`, the `escapechar` is used as a prefix to the `quotechar`. It defaults to `True`.

On output, if `doublequote` is `False` and no `escapechar` is set, `Error` is raised if a `quotechar` is found in a field.

**Dialect.escapechar**
A one-character string used by the writer to escape the `delimiter` if quoting is set to `QUOTE_NONE` and the `quotechar` if `doublequote` is `False`. On reading, the `escapechar` removes any special meaning from the following character. It defaults to `None`, which disables escaping.

**Dialect.lineterminator**
The string used to terminate lines produced by the `writer`. It defaults to ‘\r\n’.

---

**Note:** The `reader` is hard-coded to recognise either ‘\r’ or ‘\n’ as end-of-line, and ignores `lineterminator`. This behavior may change in the future.

**Dialect.quotechar**
A one-character string used to quote fields containing special characters, such as the `delimiter` or `quotechar`, or which contain new-line characters. It defaults to ‘”’.

**Dialect.quoting**
Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the `QUOTE_*` constants (see section *Module Contents*) and defaults to `QUOTE_MINIMAL`.

**Dialect.skipinitialspace**
When `True`, whitespace immediately following the `delimiter` is ignored. The default is `False`. 

---

358 Chapter 14. File Formats
Dialect.

strict  
When True, raise exception Error on bad CSV input. The default is False.

14.1.3 Reader Objects

Reader objects (DictReader instances and objects returned by the reader() function) have the following public methods:

```
csvreader.__next__()
```

Return the next row of the reader’s iterable object as a list, parsed according to the current dialect. Usually you should call this as next(reader).

Reader objects have the following public attributes:

```
csvreader.dialect
```
A read-only description of the dialect in use by the parser.

```
csvreader.line_num
```
The number of lines read from the source iterator. This is not the same as the number of records returned, as records can span multiple lines.

DictReader objects have the following public attribute:

```
csvreader.fieldnames
```
If not passed as a parameter when creating the object, this attribute is initialized upon first access or when the first record is read from the file.

14.1.4 Writer Objects

Writer objects (DictWriter instances and objects returned by the writer() function) have the following public methods. A row must be a sequence of strings or numbers for Writer objects and a dictionary mapping fieldnames to strings or numbers (by passing them through str() first) for DictWriter objects. Note that complex numbers are written out surrounded by parens. This may cause some problems for other programs which read CSV files (assuming they support complex numbers at all).

```
csvwriter.writerow(row)
```
Write the row parameter to the writer’s file object, formatted according to the current dialect.

```
csvwriter.writerows(rows)
```
Write all the rows parameters (a list of row objects as described above) to the writer’s file object, formatted according to the current dialect.

Writer objects have the following public attribute:

```
csvwriter.dialect
```
A read-only description of the dialect in use by the writer.

DictWriter objects have the following public method:

```
DictWriter.writerow()
```
Write a row with the field names (as specified in the constructor). New in version 3.2.

14.1.5 Examples

The simplest example of reading a CSV file:

```
import csv
with open('some.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Reading a file with an alternate format:
```python
import csv
with open('passwd', newline='') as f:
    reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE_NONE)
    for row in reader:
        print(row)
```

The corresponding simplest possible writing example is:

```python
import csv
with open('some.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

Since `open()` is used to open a CSV file for reading, the file will by default be decoded into unicode using the system default encoding (see `locale.getpreferredencoding()`). To decode a file using a different encoding, use the `encoding` argument of `open`:

```python
import csv
with open('some.csv', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

The same applies to writing in something other than the system default encoding: specify the encoding argument when opening the output file.

Registering a new dialect:

```python
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
with open('passwd', newline='') as f:
    reader = csv.reader(f, 'unixpwd')
```

A slightly more advanced use of the reader — catching and reporting errors:

```python
import csv, sys
filename = 'some.csv'
with open(filename, newline='') as f:
    reader = csv.reader(f)
    try:
        for row in reader:
            print(row)
    except csv.Error as e:
        sys.exit('file {}, line {}: {}'.format(filename, reader.line_num, e))
```

And while the module doesn’t directly support parsing strings, it can easily be done:

```python
import csv
for row in csv.reader(['one,two,three']):
    print(row)
```

### 14.2 configparser — Configuration file parser

This module provides the `ConfigParser` class which implements a basic configuration language which provides a structure similar to what’s found in Microsoft Windows INI files. You can use this to write Python programs which can be customized by end users easily.

**Note:** This library does not interpret or write the value-type prefixes used in the Windows Registry extended version of INI syntax.

**See Also:**
Module `shlex` Support for a creating Unix shell-like mini-languages which can be used as an alternate format for application configuration files.

Module `json` The json module implements a subset of JavaScript syntax which can also be used for this purpose.

14.2.1 Quick Start

Let’s take a very basic configuration file that looks like this:

```ini
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressLevel = 9
ForwardX11 = yes

[bitbucket.org]
User = hg

[topsecret.server.com]
Port = 50022
ForwardX11 = no
```

The structure of INI files is described in the following section. Essentially, the file consists of sections, each of which contains keys with values. `configparser` classes can read and write such files. Let’s start by creating the above configuration file programatically.

```python
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config['DEFAULT'] = {'ServerAliveInterval': '45', ...
                     'Compression': 'yes', ...
                     'CompressLevel': '9'}
>>> config['bitbucket.org'] = {}
>>> config['bitbucket.org']['User'] = 'hg'
>>> config['topsecret.server.com'] = {}
>>> topsecret = config['topsecret.server.com']
>>> topsecret['Port'] = '50022'  # mutates the parser
>>> topsecret['ForwardX11'] = 'no'  # same here
>>> config['DEFAULT']['ForwardX11'] = 'yes'
>>> with open('example.ini', 'w') as configfile:
...     config.write(configfile)
...```

As you can see, we can treat a config parser much like a dictionary. There are differences, outlined later, but the behavior is very close to what you would expect from a dictionary.

Now that we have created and saved a configuration file, let’s read it back and explore the data it holds.

```python
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config.sections()
[]
>>> config.read('example.ini')
['example.ini']
>>> config.sections()
['bitbucket.org', 'topsecret.server.com']
>>> 'bitbucket.org' in config
True
>>> 'bytebong.com' in config
False
>>> config['bitbucket.org']['User']
'hg'
```
>>> config['DEFAULT']['Compression']
'yes'
>>> topsecret = config['topsecret.server.com']
>>> topsecret['ForwardX11']
'no'
>>> topsecret['Port']
'50022'
>>> for key in config['bitbucket.org']: print(key)
...
user
compressionlevel
serveraliveinterval
compression
forwardx11
>>> config['bitbucket.org']['ForwardX11']
'yes'

As we can see above, the API is pretty straightforward. The only bit of magic involves the DEFAULT section which provides default values for all other sections \(^2\). Note also that keys in sections are case-insensitive and stored in lowercase \(^1\).

### 14.2.2 Supported Datatypes

Config parsers do not guess datatypes of values in configuration files, always storing them internally as strings. This means that if you need other datatypes, you should convert on your own:

```python
>>> int(topsecret['Port'])
50022
>>> float(topsecret['CompressionLevel'])
9.0
```

Extracting Boolean values is not that simple, though. Passing the value to `bool()` would do no good since `bool('False')` is still `True`. This is why config parsers also provide `getboolean()`. This method is case-insensitive and recognizes Boolean values from `'yes'/'no'`, `'on'/'off'` and `'1'/'0'`. For example:

```python
>>> topsecret.getboolean('ForwardX11')
False
>>> config['bitbucket.org'].getboolean('ForwardX11')
True
>>> config.getboolean('bitbucket.org', 'Compression')
True
```

Apart from `getboolean()`, config parsers also provide equivalent `getint()` and `getfloat()` methods, but these are far less useful since conversion using `int()` and `float()` is sufficient for these types.

### 14.2.3 Fallback Values

As with a dictionary, you can use a section’s `get()` method to provide fallback values:

```python
>>> topsecret.get('Port')
'50022'
>>> topsecret.get('CompressionLevel')
'9'
>>> topsecret.get('Cipher')
>>> topsecret.get('Cipher', '3des-cbc')
'3des-cbc'
```

\(^2\) Config parsers allow for heavy customization. If you are interested in changing the behaviour outlined by the footnote reference, consult the Customizing Parser Behaviour section.
Please note that default values have precedence over fallback values. For instance, in our example the 'CompressionLevel' key was specified only in the 'DEFAULT' section. If we try to get it from the section 'topsecret.server.com', we will always get the default, even if we specify a fallback:

```python
>>> topsecret.get('CompressionLevel', '3')
'9'
```

One more thing to be aware of is that the parser-level get() method provides a custom, more complex interface, maintained for backwards compatibility. When using this method, a fallback value can be provided via the fallback keyword-only argument:

```python
>>> config.get('bitbucket.org', 'monster',
... fallback='No such things as monsters')
'No such things as monsters'
```

The same fallback argument can be used with the getint(), getfloat() and getboolean() methods, for example:

```python
>>> 'BatchMode' in topsecret
False
>>> topsecret.getboolean('BatchMode', fallback=True)
True
>>> config['DEFAULT']['BatchMode'] = 'no'
>>> topsecret.getboolean('BatchMode', fallback=True)
False
```

### 14.2.4 Supported INI File Structure

A configuration file consists of sections, each led by a [section] header, followed by key/value entries separated by a specific string ( = or : by default). By default, section names are case sensitive but keys are not. Leading and trailing whitespace is removed from keys and values. Values can be omitted, in which case the key/value delimiter may also be left out. Values can also span multiple lines, as long as they are indented deeper than the first line of the value. Depending on the parser’s mode, blank lines may be treated as parts of multiline values or ignored.

Configuration files may include comments, prefixed by specific characters (# and ; by default). Comments may appear on their own on an otherwise empty line, possibly indented.

For example:

```
[Simple Values]
key=value
spaces in keys=allowed
spaces in values=allowed as well
spaces around the delimiter = obviously
you can also use : to delimit keys from values

[All Values Are Strings]
values like this: 1000000
or this: 3.14159265359
are they treated as numbers? : no
integers, floats and booleans are held as: strings
can use the API to get converted values directly: true

[Multiline Values]
chorus: I’m a lumberjack, and I’m okay
  I sleep all night and I work all day

[No Values]
key_without_value
empty string value here =
```
[You can use comments]

# like this
; or this

# By default only in an empty line.
# Inline comments can be harmful because they prevent users
# from using the delimiting characters as parts of values.
# That being said, this can be customized.

[Sections Can Be Indented]

can_values_be_as_well = True
does_that_meanAnything_special = False
purpose = formatting for readability
multiline_values = are
handled just fine as
long as they are indented
deeper than the first line
of a value
# Did I mention we can indent comments, too?

14.2.5 Interpolation of values

On top of the core functionality, ConfigParser supports interpolation. This means values can be preprocessed before returning them from get() calls.

class configparser.BasicInterpolation

The default implementation used by ConfigParser. It enables values to contain format strings which refer to other values in the same section, or values in the special default section ¹. Additional default values can be provided on initialization.

For example:

[Paths]
home_dir: /Users
my_dir: %(home_dir)s/lumberjack
my_pictures: %(my_dir)s/Pictures

In the example above, ConfigParser with interpolation set to BasicInterpolation() would resolve %(home_dir)s to the value of home_dir (/Users in this case). %(my_dir)s in effect would resolve to /Users/lumberjack. All interpolations are done on demand so keys used in the chain of references do not have to be specified in any specific order in the configuration file.

With interpolation set to None, the parser would simply return %(my_dir)s/Pictures as the value of my_pictures and %(home_dir)s/lumberjack as the value of my_dir.

class configparser.ExtendedInterpolation

An alternative handler for interpolation which implements a more advanced syntax, used for instance in zc.buildout. Extended interpolation is using ${section:option} to denote a value from a foreign section. Interpolation can span multiple levels. For convenience, if the section: part is omitted, interpolation defaults to the current section (and possibly the default values from the special section).

For example, the configuration specified above with basic interpolation, would look like this with extended interpolation:

[Paths]
home_dir: /Users
my_dir: $(home_dir)/lumberjack
my_pictures: $(my_dir)/Pictures

364 Chapter 14. File Formats
Values from other sections can be fetched as well:

```python
[Common]
home_dir: /Users
library_dir: /Library
system_dir: /System
macports_dir: /opt/local

[Frameworks]
Python: 3.2
path: ${Common:system_dir}/Library/Frameworks/

[Arthur]
nickname: Two Sheds
last_name: Jackson
my_dir: ${Common:home_dir}/twosheds
my_pictures: ${my_dir}/Pictures
python_dir: ${Frameworks:path}/Python/Versions/${Frameworks:Python}
```

## 14.2.6 Mapping Protocol Access

New in version 3.2. Mapping protocol access is a generic name for functionality that enables using custom objects as if they were dictionaries. In case of `configparser`, the mapping interface implementation is using the `parser['section']`['option'] notation.

`parser['section']` in particular returns a proxy for the section’s data in the parser. This means that the values are not copied but they are taken from the original parser on demand. What’s even more important is that when values are changed on a section proxy, they are actually mutated in the original parser.

`configparser` objects behave as close to actual dictionaries as possible. The mapping interface is complete and adheres to the `MutableMapping` ABC. However, there are a few differences that should be taken into account:

- By default, all keys in sections are accessible in a case-insensitive manner. E.g.
  ```python
  for option in parser['section']
  ```
  yields only `optionxform`ed option key names. This means lowercased keys by default. At the same time, for a section that holds the key `a`, both expressions return `True`:
  ```python
  "a" in parser['section']
  "A" in parser['section']
  ```

- All sections include `DEFAULTSECT` values as well which means that `.clear()` on a section may not leave the section visibly empty. This is because default values cannot be deleted from the section (because technically they are not there). If they are overridden in the section, deleting causes the default value to be visible again. Trying to delete a default value causes a `KeyError`.

- `DEFAULTSECT` cannot be removed from the parser:
  ```python
  trying to delete it raises ValueError,
  parser.clear() leaves it intact,
  parser.popitem() never returns it.
  ```

- `parser.get(section, option, **kwargs)` - the second argument is not a fallback value. Note however that the section-level `get()` methods are compatible both with the mapping protocol and the classic `configparser` API.

- `parser.items()` is compatible with the mapping protocol (returns a list of `section_name, section_proxy` pairs including the `DEFAULTSECT`). However, this method can also be invoked with arguments: `parser.items(section, raw, vars)`. The latter call returns a list of `option, value` pairs for a specified `section`, with all interpolations expanded (unless `raw=True` is provided).
The mapping protocol is implemented on top of the existing legacy API so that subclasses overriding the original interface still should have mappings working as expected.

14.2.7 Customizing Parser Behaviour

There are nearly as many INI format variants as there are applications using it. `configparser` goes a long way to provide support for the largest sensible set of INI styles available. The default functionality is mainly dictated by historical background and it’s very likely that you will want to customize some of the features.

The most common way to change the way a specific config parser works is to use the `__init__()` options:

- **defaults**, default value: None
  
  This option accepts a dictionary of key-value pairs which will be initially put in the `DEFAULT` section. This makes for an elegant way to support concise configuration files that don’t specify values which are the same as the documented default.
  
  Hint: if you want to specify default values for a specific section, use `read_dict()` before you read the actual file.

- **dict_type**, default value: `collections.OrderedDict`
  
  This option has a major impact on how the mapping protocol will behave and how the written configuration files look. With the default ordered dictionary, every section is stored in the order they were added to the parser. Same goes for options within sections.
  
  An alternative dictionary type can be used for example to sort sections and options on write-back. You can also use a regular dictionary for performance reasons.
  
  Please note: there are ways to add a set of key-value pairs in a single operation. When you use a regular dictionary in those operations, the order of the keys may be random. For example:

```python
>>> parser = configparser.ConfigParser()
>>> parser.read_dict({'section1': {'key1': 'value1',
... 'key2': 'value2',
... 'key3': 'value3'},
... 'section2': {'keyA': 'valueA',
... 'keyB': 'valueB',
... 'keyC': 'valueC'},
... 'section3': {'foo': 'x',
... 'bar': 'y',
... 'baz': 'z'}}

>>> parser.sections()
['section3', 'section2', 'section1']

>>> [option for option in parser['section3']]
['baz', 'foo', 'bar']
```

In these operations you need to use an ordered dictionary as well:

```python
>>> from collections import OrderedDict
>>> parser = configparser.ConfigParser()
>>> parser.read_dict(
... OrderedDict((
... ('s1',
... OrderedDict((
... ('1', '2'),
... ('3', '4'),
... ('5', '6'))
... ))
... ),
... ('s2',
... OrderedDict({
... ... })
... ))
```

Chapter 14. File Formats
allow_no_value, default value: False

Some configuration files are known to include settings without values, but which otherwise conform to the syntax supported by configparser. The allow_no_value parameter to the constructor can be used to indicate that such values should be accepted:

```python
>>> import configparser
>>> sample_config = ""
... [mysqld]
...  user = mysql
...  pid-file = /var/run/mysqld/mysqld.pid
...  skip-external-locking
...  old_passwords = 1
...  skip-bdb
...  # we don’t need ACID today
...  skip-innodb
... ""
>>> config = configparser.ConfigParser(allow_no_value=True)
>>> config.read_string(sample_config)

>>> # Settings with values are treated as before:
>>> config["mysqld"]["user"]
'mysql'

>>> # Settings without values provide None:
>>> config["mysqld"]["skip-bdb"]

>>> # Settings which aren’t specified still raise an error:
>>> config["mysqld"]["does-not-exist"]
Traceback (most recent call last):
 ...
    KeyError: 'does-not-exist'
```

Delimiters are substrings that delimit keys from values within a section. The first occurrence of a delimiting substring on a line is considered a delimiter. This means values (but not keys) can contain the delimiters. See also the space_around_delimiters argument to ConfigParser.write().

- comment_prefixes, default value: ("#", ";")

Comment prefixes are strings that indicate the start of a valid comment within a config file. comment_prefixes are used only on otherwise empty lines (optionally indented) whereas inline_comment_prefixes can be used after every valid value (e.g. section names, options and empty lines as well). By default inline comments are disabled and "#" and ";" are used as prefixes for whole
line comments. Changed in version 3.2: In previous versions of `configparser` behaviour matched `comment_prefixes=(
    '#', ';')` and `inline_comment_prefixes=(';', ). Please note that config parsers don’t support escaping of comment prefixes so using `inline_comment_prefixes` may prevent users from specifying option values with characters used as comment prefixes. When in doubt, avoid setting `inline_comment_prefixes`. In any circumstances, the only way of storing comment prefix characters at the beginning of a line in multilime values is to interpolate the prefix, for example:

```python
>>> from configparser import ConfigParser, ExtendedInterpolation
>>> parser = ConfigParser(interpolation=ExtendedInterpolation())
>>> # the default BasicInterpolation could be used as well
>>> parser.read_string('"
... [DEFAULT]
... hash = #
... [hashes]
... shebang =
... ${hash}!/usr/bin/env python
... ${hash} -*- coding: utf-8 -*-
... extensions =
... enabled_extension
... another_extension
... disabled_by_comment
... yet_another_extension
... interpolation not necessary = if # is not at line start
... even in multilime values = line #1
... line #2
... line #3
... ""
>>> print(parser['hashes']['shebang'])
#!/usr/bin/env python
# -*- coding: utf-8 -*-
>>> print(parser['hashes']['extensions'])
enabled_extension
another_extension
yet_another_extension
>>> print(parser['hashes']['interpolation not necessary'])
if # is not at line start
>>> print(parser['hashes']['even in multiline values'])
line #1
line #2
line #3
```

- **strict**, default value: True

  When set to **True**, the parser will not allow for any section or option duplicates while reading from a single source (using `read_file()`, `read_string()` or `read_dict()`). It is recommended to use strict parsers in new applications. Changed in version 3.2: In previous versions of `configparser` behaviour matched `strict=False`.

- **empty_lines_in_values**, default value: True

  In config parsers, values can span multiple lines as long as they are indented more than the key that holds them. By default parsers also let empty lines to be parts of values. At the same time, keys can be arbitrarily indented themselves to improve readability. In consequence, when configuration files get big and complex, it is easy for the user to lose track of the file structure. Take for instance:
key = multiline
   value with a gotcha

this = is still a part of the multiline value of 'key'

This can be especially problematic for the user to see if she’s using a proportional font to edit the file. That is why when your application does not need values with empty lines, you should consider disallowing them. This will make empty lines split keys every time. In the example above, it would produce two keys, key and this.

• *default_section*, default value: `configparser.DEFAULTSECT` (that is: "DEFAULT")

   The convention of allowing a special section of default values for other sections or interpolation purposes is a powerful concept of this library, letting users create complex declarative configurations. This section is normally called "DEFAULT" but this can be customized to point to any other valid section name. Some typical values include: "general" or "common". The name provided is used for recognizing default sections when reading from any source and is used when writing configuration back to a file. Its current value can be retrieved using the `parser_instance.default_section` attribute and may be modified at runtime (i.e. to convert files from one format to another).

• *interpolation*, default value: `configparser.BasicInterpolation`

   Interpolation behaviour may be customized by providing a custom handler through the `interpolation` argument. None can be used to turn off interpolation completely, `ExtendedInterpolation()` provides a more advanced variant inspired by `zc.buildout`. More on the subject in the dedicated documentation section. `RawConfigParser` has a default value of None.

   More advanced customization may be achieved by overriding default values of these parser attributes. The defaults are defined on the classes, so they may be overriden by subclasses or by attribute assignment.

   `configparser.BOOLEAN_STATES`

   By default when using `getboolean()`, config parsers consider the following values True: ‘1’, ‘yes’, ‘true’, ‘on’ and the following values False: ‘0’, ‘no’, ‘false’, ‘off’. You can override this by specifying a custom dictionary of strings and their Boolean outcomes. For example:

   ```python
   >>> custom = configparser.ConfigParser()
   >>> custom[‘section1’] = {‘funky’: ‘nope’}
   >>> custom[‘section1’].getboolean(‘funky’)  # This raises a ValueError
   Traceback (most recent call last):
   ... ValueError: Not a boolean: nope
   >>> custom.BOOLEAN_STATES = {‘sure’: True, ‘nope’: False}
   >>> custom[‘section1’].getboolean(‘funky’)  # Now it works
   False
   ```

   Other typical Boolean pairs include accept/reject or enabled/disabled.

   `configparser.optionxform( option )`

   This method transforms option names on every read, get, or set operation. The default converts the name to lowercase. This also means that when a configuration file gets written, all keys will be lowercase. Override this method if that’s unsuitable. For example:

   ```python
   >>> config = ""
   ... [Section1]
   ... Key = Value
   ...
   ... [Section2]
   ... AnotherKey = Value
   ... ""
   >>> typical = configparser.ConfigParser()
   >>> typical.read_string(config)
   >>> list(typical[‘Section1’].keys())
   ```
['key']

```python
>>> list(typical['Section2'].keys())
['anotherkey']
```

```python
>>> custom = configparser.RawConfigParser()
>>> custom.optionxform = lambda option: option
>>> custom.read_string(config)
>>> list(custom['Section1'].keys())
['Key']
```

```python
>>> list(custom['Section2'].keys())
['AnotherKey']
```

```python
configparser.SECTCRE
A compiled regular expression used to parse section headers. The default matches \[section\] to the name "section". Whitespace is considered part of the section name, thus \[ larch \] will be read as a section of name " larch ". Override this attribute if that’s unsuitable. For example:

```python
>>> config = ""
... [Section 1]
... option = value
...
... [ Section 2 ]
... another = val
... ""
```  
```python
>>> typical = ConfigParser()
>>> typical.read_string(config)
>>> typical.sections()
['Section 1', 'Section 2 ']
```

```python
>>> custom = ConfigParser()
>>> custom.SECTCRE = re.compile(r"\[ *(?P<header>[^\]]+?) *\]"
>>> custom.read_string(config)
>>> custom.sections()
['Section 1', 'Section 2']
```

**Note:** While ConfigParser objects also use an OPTCRE attribute for recognizing option lines, it’s not recommended to override it because that would interfere with constructor options allow_no_value and delimiters.

### 14.2.8 Legacy API Examples

Mainly because of backwards compatibility concerns, configparser provides also a legacy API with explicit get/set methods. While there are valid use cases for the methods outlined below, mapping protocol access is preferred for new projects. The legacy API is at times more advanced, low-level and downright counterintuitive.

An example of writing to a configuration file:

```python
import configparser

config = configparser.RawConfigParser()

# Please note that using RawConfigParser’s set functions, you can assign
# non-string values to keys internally, but will receive an error when
# attempting to write to a file or when you get it in non-raw mode. Setting
# values using the mapping protocol or ConfigParser’s set() does not allow
# such assignments to take place.
config.add_section('Section1')
config.set('Section1', 'an_int', '15')
config.set('Section1', 'a_bool', 'true')
```
config.set('Section1', 'a_float', '3.1415')
config.set('Section1', 'baz', 'fun')
config.set('Section1', 'bar', 'Python')
config.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# Writing our configuration file to 'example.cfg'
with open('example.cfg', 'w') as configfile:
    config.write(configfile)

An example of reading the configuration file again:

```python
import configparser

config = configparser.RawConfigParser()
cfg = config.read('example.cfg')
```

# getfloat() raises an exception if the value is not a float
# getint() and getboolean() also do this for their respective types
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print(a_float + an_int)

# Notice that the next output does not interpolate '%(bar)s' or '%(baz)s'.
# This is because we are using a RawConfigParser().
if config.getboolean('Section1', 'a_bool'):
    print(config.get('Section1', 'foo'))

To get interpolation, use ConfigParser:

```python
import configparser

cfg = configparser.ConfigParser()
cfg.read('example.cfg')
```

# Set the optional *raw* argument of get() to True if you wish to disable
# interpolation in a single get operation.
print(cfg.get('Section1', 'foo', raw=False)) # -> "Python is fun!"
print(cfg.get('Section1', 'foo', raw=True))  # -> "%(bar)s is %(baz)s!"

# The optional *vars* argument is a dict with members that will take
# precedence in interpolation.
print(cfg.get('Section1', 'foo', vars={'bar': 'Documentation',
                                       'baz': 'evil'}))

# The optional *fallback* argument can be used to provide a fallback value
print(cfg.get('Section1', 'foo'))
    # -> "Python is fun!"
print(cfg.get('Section1', 'foo', fallback='Monty is not.'))
    # -> "Python is fun!"
print(cfg.get('Section1', 'monster', fallback='No such things as monsters.'))
    # -> "No such things as monsters."

# A bare print(cfg.get('Section1', 'monster')) would raise NoOptionError
# but we can also use:
print(cfg.get('Section1', 'monster', fallback=None))
    # -> None

Default values are available in both types of ConfigParsers. They are used in interpolation if an option used is not
defined elsewhere.

```python
import configparser

# New instance with 'bar' and 'baz' defaulting to 'Life' and 'hard' each
config = configparser.ConfigParser({'bar': 'Life', 'baz': 'hard'})
config.read('example.cfg')

print(config.get('Section1', 'foo'))  # -> "Python is fun!"
config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print(config.get('Section1', 'foo'))  # -> "Life is hard!"
```

### 14.2.9 ConfigParser Objects

```python
class configparser.ConfigParser(  
defaults=None,  
dict_type=collections.OrderedDict,  
allow_no_value=False,  
delimiters=('=', ':'),  
comment_prefixes=('#', ';'),  
inline_comment_prefixes=None,  
strict=True,  
empty_lines_in_values=True,  
default_section=configparser.DEFAULTSECT,  
interpolation=BasicInterpolation())
```

The main configuration parser. When `defaults` is given, it is initialized into the dictionary of intrinsic defaults. When `dict_type` is given, it will be used to create the dictionary objects for the list of sections, for the options within a section, and for the default values.

When `delimiters` is given, it is used as the set of substrings that divide keys from values. When `comment_prefixes` is given, it will be used as the set of substrings that prefix comments in otherwise empty lines. Comments can be indented. When `inline_comment_prefixes` is given, it will be used as the set of substrings that prefix comments in non-empty lines.

When `strict` is `True` (the default), the parser won’t allow for any section or option duplicates while reading from a single source (file, string or dictionary), raising `DuplicateSectionError` or `DuplicateOptionError`. When `empty_lines_in_values` is `False` (default: `True`), each empty line marks the end of an option. Otherwise, internal empty lines of a multiline option are kept as part of the value. When `allow_no_value` is `True` (default: `False`), options without values are accepted; the value held for these is `None` and they are serialized without the trailing delimiter.

When `default_section` is given, it specifies the name for the special section holding default values for other sections and interpolation purposes (normally named "DEFAULT"). This value can be retrieved and changed on runtime using the `default_section` instance attribute.

Interpolation behaviour may be customized by providing a custom handler through the `interpolation` argument. `None` can be used to turn off interpolation completely, `ExtendedInterpolation()` provides a more advanced variant inspired by `zc.buildout`. More on the subject in the dedicated documentation section.

All option names used in interpolation will be passed through the `optionxform()` method just like any other option name reference. For example, using the default implementation of `optionxform()` (which converts option names to lower case), the values `foo % (bar)s` and `foo % (BAR)s` are equivalent. Changed in version 3.1: The default `dict_type` is `collections.OrderedDict`. Changed in version 3.2: `allow_no_value`, `delimiters`, `comment_prefixes`, `strict`, `empty_lines_in_values`, `default_section` and `interpolation` were added.

```python
defaults()  
    Return a dictionary containing the instance-wide defaults.

sections()  
    Return a list of the sections available; the default section is not included in the list.

add_section(section)  
    Add a section named `section` to the instance. If a section by the given name already exists,  
    `DuplicateSectionError` is raised. If the default section name is passed, `ValueError` is
raised. The name of the section must be a string; if not, `TypeError` is raised. Changed in version 3.2: Non-string section names raise `TypeError`.

`has_section(section)`
Indicates whether the named `section` is present in the configuration. The default section is not acknowledged.

`options(section)`
Return a list of options available in the specified `section`.

`has_option(section, option)`
If the given `section` exists, and contains the given `option`, return `True`; otherwise return `False`. If the specified `section` is `None` or an empty string, DEFAULT is assumed.

`read(filenames, encoding=None)`
Attempt to read and parse a list of filenames, returning a list of filenames which were successfully parsed. If `filenames` is a string, it is treated as a single filename. If a file named in `filenames` cannot be opened, that file will be ignored. This is designed so that you can specify a list of potential configuration file locations (for example, the current directory, the user’s home directory, and some system-wide directory), and all existing configuration files in the list will be read. If none of the named files exist, the `ConfigParser` instance will contain an empty dataset. An application which requires initial values to be loaded from a file should load the required file or files using `read_file()` before calling `read()` for any optional files:

```python
import configparser, os

config = configparser.ConfigParser()
config.read_file(open('defaults.cfg'))
config.read(["site.cfg", os.path.expanduser("~/.myapp.cfg")],
            encoding='cp1250')
```

New in version 3.2: The `encoding` parameter. Previously, all files were read using the default encoding for `open()`.

`read_file(f, source=None)`
Read and parse configuration data from `f` which must be an iterable yielding Unicode strings (for example files opened in text mode).

Optional argument `source` specifies the name of the file being read. If not given and `f` has a `name` attribute, that is used for `source`; the default is `'<????>'`. New in version 3.2: Replaces `readfp()`.

`read_string(string, source='<string>')`
Parse configuration data from a string.

Optional argument `source` specifies a context-specific name of the string passed. If not given, `'<string>'` is used. This should commonly be a filesystem path or a URL. New in version 3.2.

`read_dict(dictionary, source='<dict>')` Load configuration from any object that provides a dict-like `items()` method. Keys are section names, values are dictionaries with keys and values that should be present in the section. If the used dictionary type preserves order, sections and their keys will be added in order. Values are automatically converted to strings.

Optional argument `source` specifies a context-specific name of the dictionary passed. If not given, `<dict>` is used.

This method can be used to copy state between parsers. New in version 3.2.

`get(section, option, *, raw=False, vars=None[, fallback])` Get an option value for the named `section`. If `vars` is provided, it must be a dictionary. The `option` is looked up in `vars` (if provided), `section`, and in `DEFAULTSECT` in that order. If the key is not found and `fallback` is provided, it is used as a fallback value. `None` can be provided as a `fallback` value.
All the ‘%’ interpolations are expanded in the return values, unless the raw argument is true. Values for interpolation keys are looked up in the same manner as the option. Changed in version 3.2: Arguments raw, vars and fallback are keyword only to protect users from trying to use the third argument as the fallback fallback (especially when using the mapping protocol).

```python
getint(section, option, *, raw=False, vars=None, fallback)
```
A convenience method which coerces the option in the specified section to an integer. See get() for explanation of raw, vars and fallback.

```python
getfloat(section, option, *, raw=False, vars=None, fallback)
```
A convenience method which coerces the option in the specified section to a floating point number. See get() for explanation of raw, vars and fallback.

```python
getboolean(section, option, *, raw=False, vars=None, fallback)
```
A convenience method which coerces the option in the specified section to a Boolean value. Note that the accepted values for the option are ‘1’, ‘yes’, ‘true’, and ‘on’, which cause this method to return True, and ‘0’, ‘no’, ‘false’, and ‘off’, which cause it to return False. These string values are checked in a case-insensitive manner. Any other value will cause it to raise ValueError. See get() for explanation of raw, vars and fallback.

```python
items(raw=False, vars=None)
```
When section is not given, return a list of section_name, section_proxy pairs, including DEFAULT-SECT.

Otherwise, return a list of name, value pairs for the options in the given section. Optional arguments have the same meaning as for the get() method. Changed in version 3.2: Items present in vars no longer appear in the result. The previous behaviour mixed actual parser options with variables provided for interpolation.

```python
set(section, option, value)
```
If the given section exists, set the given option to the specified value; otherwise raise NoSectionError. option and value must be strings; if not, TypeError is raised.

```python
write(fileobject, space_around_delimiters=True)
```
Write a representation of the configuration to the specified file object, which must be opened in text mode (accepting strings). This representation can be parsed by a future read() call. If space_around_delimiters is true, delimiters between keys and values are surrounded by spaces.

```python
remove_option(section, option)
```
Remove the specified option from the specified section. If the section does not exist, raise NoSectionError. If the option existed to be removed, return True; otherwise return False.

```python
remove_section(section)
```
Remove the specified section from the configuration. If the section in fact existed, return True. Otherwise return False.

```python
optionxform(option)
```
Transforms the option name option as found in an input file or as passed in by client code to the form that should be used in the internal structures. The default implementation returns a lower-case version of option; subclasses may override this or client code can set an attribute of this name on instances to affect this behavior.

You don’t need to subclass the parser to use this method, you can also set it on an instance, to a function that takes a string argument and returns a string. Setting it to str, for example, would make option names case sensitive:

```python
cfgparser = ConfigParser()
cfgparser.optionxform = str
```
Note that when reading configuration files, whitespace around the option names is stripped before optionxform() is called.
The Python Library Reference, Release 3.3.3

readfp (fp, filename=None)
Deprecated since version 3.2: Use read_file() instead. Changed in version 3.2: readfp() now iterates on f instead of calling f.readline(). For existing code calling readfp() with arguments which don’t support iteration, the following generator may be used as a wrapper around the file-like object:

```python
def readline_generator(f):
    line = f.readline()
    while line:
        yield line
    line = f.readline()
```

Instead of parser.readfp(f) use parser.read_file(readline_generator(f)).

configparser.MAX_INTERPOLATION_DEPTH
The maximum depth for recursive interpolation for get() when the raw parameter is false. This is relevant only when the default interpolation is used.

14.2.10 RawConfigParser Objects

class configparser.RawConfigParser (defaults=None, dict_type=collections.OrderedDict, allow_no_value=False, *, delimiters=('=', ':'), comment_prefixes=('#', ';'), inline_comment_prefixes=None, strict=True, empty_lines_in_values=True, default_section=configparser.DEFAULTSECT[, interpolation ])

Legacy variant of the ConfigParser with interpolation disabled by default and unsafe add_section and set methods.

**Note:** Consider using ConfigParser instead which checks types of the values to be stored internally. If you don’t want interpolation, you can use ConfigParser (interpolation=None).

add_section (section)
Add a section named section to the instance. If a section by the given name already exists, DuplicateSectionError is raised. If the default section name is passed, ValueError is raised.

Type of section is not checked which lets users create non-string named sections. This behaviour is unsupported and may cause internal errors.

set (section, option, value)
If the given section exists, set the given option to the specified value; otherwise raise NoSectionError. While it is possible to use RawConfigParser (or ConfigParser with raw parameters set to true) for internal storage of non-string values, full functionality (including interpolation and output to files) can only be achieved using string values.

This method lets users assign non-string values to keys internally. This behaviour is unsupported and will cause errors when attempting to write to a file or get it in non-raw mode. **Use the mapping protocol API** which does not allow such assignments to take place.

14.2.11 Exceptions

exception configparser.Error
Base class for all other configparser exceptions.

exception configparser.NoSectionError
Exception raised when a specified section is not found.
**14.3 netrc — netrc file processing**

The `netrc` class parses and encapsulates the netrc file format used by the Unix `ftp` program and other FTP clients.

```
A netrc instance or subclass instance encapsulates data from a netrc file. The initialization argument, if present, specifies the file to parse. If no argument is given, the file `.netrc` in the user's home directory will be read. Parse errors will raise NetrcParseError with diagnostic information including the file name, line number, and terminating token. If no argument is specified on a POSIX system, the presence of passwords in the `.netrc` file will raise a NetrcParseError if the file ownership or permissions are insecure (owned by a user other than the user running the process, or accessible for read or write by any other user). This implements security behavior equivalent to that of `ftp` and other programs that use `.netrc`. Changed in version 3.3.3: Added the POSIX permission check.
```

```
**exception netrc.NetrcParseError**
Exception raised by the `netrc` class when syntactical errors are encountered in source text. Instances of this exception provide three interesting attributes: `msg` is a textual explanation of the error, `filename` is the name of the source file, and `lineno` gives the line number on which the error was found.
```

A `netrc` instance has the following methods:

```
14.3.1 netrc Objects
```

**Source code:** Lib/netrc.py
netrc.authenticators(host)
Return a 3-tuple (login, account, password) of authenticators for host. If the netrc file did not contain an entry for the given host, return the tuple associated with the ‘default’ entry. If neither matching host nor default entry is available, return None.

netrc.__repr__()
Dump the class data as a string in the format of a netrc file. (This discards comments and may reorder the entries.)

Instances of netrc have public instance variables:

netrc.hosts
Dictionary mapping host names to (login, account, password) tuples. The ‘default’ entry, if any, is represented as a pseudo-host by that name.

netrc.macros
Dictionary mapping macro names to string lists.

Note: Passwords are limited to a subset of the ASCII character set. All ASCII punctuation is allowed in passwords, however, note that whitespace and non-printable characters are not allowed in passwords. This is a limitation of the way the .netrc file is parsed and may be removed in the future.

14.4 xdrlib — Encode and decode XDR data

Source code: Lib/xdrlib.py

The xdrlib module supports the External Data Representation Standard as described in RFC 1014, written by Sun Microsystems, Inc. June 1987. It supports most of the data types described in the RFC.

The xdrlib module defines two classes, one for packing variables into XDR representation, and another for unpacking from XDR representation. There are also two exception classes.

class xdrlib.Packer
Packer is the class for packing data into XDR representation. The Packer class is instantiated with no arguments.

class xdrlib.Unpacker(data)
Unpacker is the complementary class which unpacks XDR data values from a string buffer. The input buffer is given as data.

See Also:
RFC 1014 - XDR: External Data Representation Standard This RFC defined the encoding of data which was XDR at the time this module was originally written. It has apparently been obsoleted by RFC 1832.

RFC 1832 - XDR: External Data Representation Standard Newer RFC that provides a revised definition of XDR.

14.4.1 Packer Objects

Packer instances have the following methods:

Packer.get_buffer()
Returns the current pack buffer as a string.

Packer.reset()
Resets the pack buffer to the empty string.
In general, you can pack any of the most common XDR data types by calling the appropriate `pack_type()` method. Each method takes a single argument, the value to pack. The following simple data type packing methods are supported: `pack_uint()`, `pack_int()`, `pack_enum()`, `pack_bool()`, `pack_uhyper()`, and `pack_hyper()`.

**Packer**. `pack_float(value)`
- Packs the single-precision floating point number `value`.

**Packer**. `pack_double(value)`
- Packs the double-precision floating point number `value`.

The following methods support packing strings, bytes, and opaque data:

**Packer**. `pack_fstring(n, s)`
- Packs a fixed length string, `s`. `n` is the length of the string but it is not packed into the data buffer. The string is padded with null bytes if necessary to guaranteed 4 byte alignment.

**Packer**. `pack_fopaque(n, data)`
- Packs a fixed length opaque data stream, similarly to `pack_fstring()`.

**Packer**. `pack_string(s)`
- Packs a variable length string, `s`. The length of the string is first packed as an unsigned integer, then the string data is packed with `pack_fstring()`.

**Packer**. `pack_opaque(data)`
- Packs a variable length opaque data string, similarly to `pack_string()`.

**Packer**. `pack_bytes(bytes)`
- Packs a variable length byte stream, similarly to `pack_string()`.

The following methods support packing arrays and lists:

**Packer**. `pack_list(list, pack_item)`
- Packs a list of homogeneous items. This method is useful for lists with an indeterminate size; i.e. the size is not available until the entire list has been walked. For each item in the list, an unsigned integer 1 is packed first, followed by the data value from the list. `pack_item` is the function that is called to pack the individual item. At the end of the list, an unsigned integer 0 is packed.

For example, to pack a list of integers, the code might appear like this:

```python
import xdrlib
p = xdrlib.Packer()
p.pack_list([1, 2, 3], p.pack_int)
```

**Packer**. `pack_farray(n, array, pack_item)`
- Packs a fixed length list (`array`) of homogeneous items. `n` is the length of the list; it is not packed into the buffer, but a `ValueError` exception is raised if `len(array)` is not equal to `n`. As above, `pack_item` is the function used to pack each element.

**Packer**. `pack_array(list, pack_item)`
- Packs a variable length list of homogeneous items. First, the length of the list is packed as an unsigned integer, then each element is packed as in `pack_farray()` above.

### 14.4.2 Unpacker Objects

The `Unpacker` class offers the following methods:

**Unpacker**. `reset(data)`
- Resets the string buffer with the given `data`.

**Unpacker**. `get_position()`
- Returns the current unpack position in the data buffer.
Unpacker.set_position(position)
Sets the data buffer unpack position to position. You should be careful about using get_position() and set_position().

Unpacker.get_buffer()
Returns the current unpack data buffer as a string.

Unpacker.done()
Indicates unpack completion. Raises an Error exception if all of the data has not been unpacked.

In addition, every data type that can be packed with a Packer, can be unpacked with an Unpacker. Unpacking methods are of the form unpack_type(), and take no arguments. They return the unpacked object.

Unpacker.unpack_float()
Unpacks a single-precision floating point number.

Unpacker.unpack_double()
Unpacks a double-precision floating point number, similarly to unpack_float().

In addition, the following methods unpack strings, bytes, and opaque data:

Unpacker.unpack_fstring(n)
Unpacks and returns a fixed length string. n is the number of characters expected. Padding with null bytes to guaranteed 4 byte alignment is assumed.

Unpacker.unpack_fopaque(n)
Unpacks and returns a fixed length opaque data stream, similarly to unpack_fstring().

Unpacker.unpack_string()
Unpacks and returns a variable length string. The length of the string is first unpacked as an unsigned integer, then the string data is unpacked with unpack_fstring().

Unpacker.unpackOpaque()
Unpacks and returns a variable length opaque data string, similarly to unpack_string().

Unpacker.unpack_bytes()
Unpacks and returns a variable length byte stream, similarly to unpack_string().

The following methods support unpacking arrays and lists:

Unpacker.unpack_list(unpack_item)
Unpacks and returns a list of homogeneous items. The list is unpacked one element at a time by first unpacking an unsigned integer flag. If the flag is 1, then the item is unpacked and appended to the list. A flag of 0 indicates the end of the list. unpack_item is the function that is called to unpack the items.

Unpacker.unpack_farray(n, unpack_item)
Unpacks and returns (as a list) a fixed length array of homogeneous items. n is number of list elements to expect in the buffer. As above, unpack_item is the function used to unpack each element.

Unpacker.unpack_array(unpack_item)
Unpacks and returns a variable length list of homogeneous items. First, the length of the list is unpacked as an unsigned integer, then each element is unpacked as in unpack_farray() above.

14.4.3 Exceptions

Exceptions in this module are coded as class instances:

exception xdrlib.Error
The base exception class. Error has a single public attribute msg containing the description of the error.

exception xdrlib.ConversionError
Class derived from Error. Contains no additional instance variables.

Here is an example of how you would catch one of these exceptions:
import xdrlib
p = xdrlib.Packer()
try:
    p.pack_double(8.01)
except xdrlib.ConversionError as instance:
    print('packing the double failed:', instance.msg)

14.5 plistlib — Generate and parse Mac OS X .plist files

Source code: Lib/plistlib.py

This module provides an interface for reading and writing the “property list” XML files used mainly by Mac OS X.

The property list (.plist) file format is a simple XML pickle supporting basic object types, like dictionaries, lists, numbers and strings. Usually the top level object is a dictionary.

To write out and to parse a plist file, use the writePlist() and readPlist() functions.

To work with plist data in bytes objects, use writePlistToBytes() and readPlistFromBytes().

Values can be strings, integers, floats, booleans, tuples, lists, dictionaries (but only with string keys), Data or datetime.datetime objects. String values (including dictionary keys) have to be unicode strings – they will be written out as UTF-8.

The <data> plist type is supported through the Data class. This is a thin wrapper around a Python bytes object. Use Data if your strings contain control characters.

See Also:

PList manual page Apple’s documentation of the file format.

This module defines the following functions:

plistlib.readPlist(pathOrFile)
    Read a plist file. pathOrFile may either be a file name or a (readable and binary) file object. Return the unpacked root object (which usually is a dictionary).

    The XML data is parsed using the Expat parser from xml.parsers.expat – see its documentation for possible exceptions on ill-formed XML. Unknown elements will simply be ignored by the plist parser.

plistlib.writePlist(rootObject, pathOrFile)
    Write rootObject to a plist file. pathOrFile may either be a file name or a (writable and binary) file object.

    A TypeError will be raised if the object is of an unsupported type or a container that contains objects of unsupported types.

plistlib.readPlistFromBytes(data)
    Read a plist data from a bytes object. Return the root object.

plistlib.writePlistToBytes(rootObject)
    Return rootObject as a plist-formatted bytes object.

The following class is available:

class plistlib.Data(data)
    Return a “data” wrapper object around the bytes object data. This is used in functions converting from/to plists to represent the <data> type available in plists.

    It has one attribute, data, that can be used to retrieve the Python bytes object stored in it.
14.5.1 Examples

Generating a plist:

```python
pl = dict(
    aString = "Doodah",
    aList = ["A", "B", 12, 32.1, [1, 2, 3]],
    aFloat = 0.1,
    anInt = 728,
    aDict = dict(
        anotherString = "<hello & hi there!>",
        aThirdString = "M\xe4ssig, Ma\xdf",
        aTrueValue = True,
        aFalseValue = False,
    ),
    someData = Data(b"<binary gunk>"),
    someMoreData = Data(b"<lots of binary gunk>" * 10),
    aDate = datetime.datetime.fromtimestamp(time.mktime(time.gmtime())),
)
writePlist(pl, fileName)
```

Parsing a plist:

```python
pl = readPlist(pathOrFile)
print(pl["aKey"])
```
CHAPTER
FIFTEEN

CRYPTOGRAPHIC SERVICES

The modules described in this chapter implement various algorithms of a cryptographic nature. They are available at the discretion of the installation. On Unix systems, the crypt module may also be available. Here’s an overview:

15.1 hashlib — Secure hashes and message digests

Source code: Lib/hashlib.py

This module implements a common interface to many different secure hash and message digest algorithms. Included are the FIPS secure hash algorithms SHA1, SHA224, SHA256, SHA384, and SHA512 (defined in FIPS 180-2) as well as RSA's MD5 algorithm (defined in Internet RFC 1321). The terms “secure hash” and “message digest” are interchangeable. Older algorithms were called message digests. The modern term is secure hash.

Note: If you want the adler32 or crc32 hash functions, they are available in the zlib module.

Warning: Some algorithms have known hash collision weaknesses, refer to the “See also” section at the end.

There is one constructor method named for each type of hash. All return a hash object with the same simple interface. For example: use sha1() to create a SHA1 hash object. You can now feed this object with bytes-like objects (normally bytes) using the update() method. At any point you can ask it for the digest of the concatenation of the data fed to it so far using the digest() or hexdigest() methods.

Note: For better multithreading performance, the Python GIL is released for data larger than 2047 bytes at object creation or on update.

Note: Feeding string objects into update() is not supported, as hashes work on bytes, not on characters.

Constructors for hash algorithms that are always present in this module are md5(), sha1(), sha224(), sha256(), sha384(), and sha512(). Additional algorithms may also be available depending upon the OpenSSL library that Python uses on your platform.

For example, to obtain the digest of the byte string b’Nobody inspects the spammish repetition’:

```python
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
```

383
>>> m.digest_size
16
>>> m.block_size
64

More condensed:

```python
>>> hashlib.sha224(b"Nobody inspects the spammish repetition").hexdigest()
'a4337bc45a8fc544c03f52dc550cd6e1e87021bc896588bd79e901e2'
```

hashlib.
new (name [, data ])

Is a generic constructor that takes the string name of the desired algorithm as its first parameter. It also exists to allow access to the above listed hashes as well as any other algorithms that your OpenSSL library may offer. The named constructors are much faster than new() and should be preferred.

Using new() with an algorithm provided by OpenSSL:

```python
>>> h = hashlib.new('ripemd160')
>>> h.update(b"Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894ecc'
```

Hashlib provides the following constant attributes:

hashlib.
algorithms_guaranteed
Contains the names of the hash algorithms guaranteed to be supported by this module on all platforms. New in version 3.2.

hashlib.
algorithms_available
Contains the names of the hash algorithms that are available in the running Python interpreter. These names will be recognized when passed to new().algorithms_guaranteed will always be a subset. Duplicate algorithms with different name formats may appear in this set (thanks to OpenSSL). New in version 3.2.

The following values are provided as constant attributes of the hash objects returned by the constructors:

hash.
digest_size
The size of the resulting hash in bytes.

hash.
block_size
The internal block size of the hash algorithm in bytes.

A hash object has the following methods:

hash.
update (arg)
Update the hash object with the object arg, which must be interpretable as a buffer of bytes. Repeated calls are equivalent to a single call with the concatenation of all the arguments: m.update(a); m.update(b) is equivalent to m.update(a+b). Changed in version 3.1: The Python GIL is released to allow other threads to run while hash updates on data larger than 2047 bytes is taking place when using hash algorithms supplied by OpenSSL.

hash.
digest ()
Return the digest of the data passed to the update() method so far. This is a bytes object of size digest_size which may contain bytes in the whole range from 0 to 255.

hash.
hexdigest ()
Like digest() except the digest is returned as a string object of double length, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

hash.
copy ()
Return a copy (“clone”) of the hash object. This can be used to efficiently compute the digests of data sharing a common initial substring.

See Also:

Module hmac A module to generate message authentication codes using hashes.
Module **base64**  Another way to encode binary hashes for non-binary environments.


http://en.wikipedia.org/wiki/Cryptographic_hash_function#Cryptographic_hash_algorithms  Wikipedia article with information on which algorithms have known issues and what that means regarding their use.

### 15.2 hmac — Keyed-Hashing for Message Authentication

**Source code:** Lib/hmac.py

This module implements the HMAC algorithm as described by RFC 2104.

```python
hmac.new(key, msg=None, digestmod=None)
```

Return a new hmac object. `key` is a bytes object giving the secret key. If `msg` is present, the method call `update(msg)` is made. `digestmod` is the digest constructor or module for the HMAC object to use. It defaults to the `hashlib.md5` constructor.

An HMAC object has the following methods:

**HMAC.update(msg)**

Update the hmac object with the bytes object `msg`. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a + b)`.

**HMAC.digest()**

Return the digest of the bytes passed to the `update()` method so far. This bytes object will be the same length as the `digest_size` of the digest given to the constructor. It may contain non-ASCII bytes, including NUL bytes.

**Warning:** When comparing the output of `digest()` to an externally-supplied digest during a verification routine, it is recommended to use the `compare_digest()` function instead of the `==` operator to reduce the vulnerability to timing attacks.

**HMAC.hexdigest()**

Like `digest()` except the digest is returned as a string twice the length containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

**Warning:** When comparing the output of `hexdigest()` to an externally-supplied digest during a verification routine, it is recommended to use the `compare_digest()` function instead of the `==` operator to reduce the vulnerability to timing attacks.

**HMAC.copy()**

Return a copy ("clone") of the hmac object. This can be used to efficiently compute the digests of strings that share a common initial substring.

This module also provides the following helper function:

```python
hmac.compare_digest(a, b)
```

Return `a == b`. This function uses an approach designed to prevent timing analysis by avoiding content-based short circuiting behaviour, making it appropriate for cryptography. `a` and `b` must both be of the same type: either `str` (ASCII only, as e.g. returned by `hmac.hexdigest()`), or a `bytes-like object`.

**Note:** If `a` and `b` are of different lengths, or if an error occurs, a timing attack could theoretically reveal information about the types and lengths of `a` and `b`—but not their values.

New in version 3.3.
See Also:

Module `hashlib` The Python module providing secure hash functions.

Hardcore cypherpunks will probably find the cryptographic modules written by A.M. Kuchling of further interest; the package contains modules for various encryption algorithms, most notably AES. These modules are not distributed with Python but available separately. See the URL http://www.pycrypto.org for more information.
CHAPTER SIXTEEN

GENERIC OPERATING SYSTEM SERVICES

The modules described in this chapter provide interfaces to operating system features that are available on (almost) all operating systems, such as files and a clock. The interfaces are generally modeled after the Unix or C interfaces, but they are available on most other systems as well. Here’s an overview:

16.1 os — Miscellaneous operating system interfaces

This module provides a portable way of using operating system dependent functionality. If you just want to read or write a file see open(), if you want to manipulate paths, see the os.path module, and if you want to read all the lines in all the files on the command line see the fileinput module. For creating temporary files and directories see the tempfile module, and for high-level file and directory handling see the shutil module.

Notes on the availability of these functions:

- The design of all built-in operating system dependent modules of Python is such that as long as the same functionality is available, it uses the same interface; for example, the function os.stat(path) returns stat information about path in the same format (which happens to have originated with the POSIX interface).
- Extensions peculiar to a particular operating system are also available through the os module, but using them is of course a threat to portability.
- All functions accepting path or file names accept both bytes and string objects, and result in an object of the same type, if a path or file name is returned.
- An “Availability: Unix” note means that this function is commonly found on Unix systems. It does not make any claims about its existence on a specific operating system.
- If not separately noted, all functions that claim “Availability: Unix” are supported on Mac OS X, which builds on a Unix core.

Note: All functions in this module raise OSError in the case of invalid or inaccessible file names and paths, or other arguments that have the correct type, but are not accepted by the operating system.

exception os.error
An alias for the built-in OSError exception.

os.name
The name of the operating system dependent module imported. The following names have currently been registered: ‘posix’, ‘nt’, ‘mac’, ‘os2’, ‘ce’, ‘java’.

See Also:
- sys.platform has a finer granularity. os.uname() gives system-dependent version information.
- The platform module provides detailed checks for the system’s identity.
16.1.1 File Names, Command Line Arguments, and Environment Variables

In Python, file names, command line arguments, and environment variables are represented using the string type. On some systems, decoding these strings to and from bytes is necessary before passing them to the operating system. Python uses the file system encoding to perform this conversion (see sys.getfilesystemencoding()). Changed in version 3.1: On some systems, conversion using the file system encoding may fail. In this case, Python uses the surrogateescape encoding error handler, which means that undecodable bytes are replaced by a Unicode character U+DCxx on decoding, and these are again translated to the original byte on encoding. The file system encoding must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions may raise UnicodeErrors.

16.1.2 Process Parameters

These functions and data items provide information and operate on the current process and user.

\textbf{os.ctermid()}

Return the filename corresponding to the controlling terminal of the process.

Availability: Unix.

\textbf{os.environ}

A \texttt{mapping} object representing the string environment. For example, \texttt{environ[’HOME’]} is the pathname of your home directory (on some platforms), and is equivalent to \texttt{getenv("HOME")} in C.

This mapping is captured the first time the \texttt{os} module is imported, typically during Python startup as part of processing site.py. Changes to the environment made after this time are not reflected in \texttt{os.environ}, except for changes made by modifying \texttt{os.environ} directly.

If the platform supports the \texttt{putenv()} function, this mapping may be used to modify the environment as well as query the environment. \texttt{putenv()} will be called automatically when the mapping is modified.

On Unix, keys and values use \texttt{sys.getfilesystemencoding()} and ‘surrogateescape’ error handler. Use \texttt{environb} if you would like to use a different encoding.

\textbf{Note:} Calling \texttt{putenv()} directly does not change \texttt{os.environ}, so it’s better to modify \texttt{os.environ}.

\textbf{Note:} On some platforms, including FreeBSD and Mac OS X, setting \texttt{environ} may cause memory leaks. Refer to the system documentation for \texttt{putenv()}.

If \texttt{putenv()} is not provided, a modified copy of this mapping may be passed to the appropriate process-creation functions to cause child processes to use a modified environment.

If the platform supports the \texttt{unsetenv()} function, you can delete items in this mapping to unset environment variables. \texttt{unsetenv()} will be called automatically when an item is deleted from \texttt{os.environ}, and when one of the \texttt{pop()} or \texttt{clear()} methods is called.

\textbf{os.environb}

Bytes version of \texttt{environ}: a \texttt{mapping} object representing the environment as byte strings. \texttt{environ} and \texttt{environb} are synchronized (modify \texttt{environb} updates \texttt{environ}, and vice versa).

\texttt{environb} is only available if \texttt{supports_bytes_environ} is True. New in version 3.2.

\textbf{os.chdir (path)}
\textbf{os.fchdir (fd)}
\textbf{os.getcwd ()}

These functions are described in \textit{Files and Directories}.

\textbf{os.fsencode (filename)}

Encode \texttt{filename} to the filesystem encoding with ‘surrogateescape’ error handler, or ‘strict’ on Windows; return bytes unchanged.
fsdecode() is the reverse function. New in version 3.2.

os.fsdecode(filename)
Decode filename from the filesystem encoding with 'surrogateescape' error handler, or 'strict' on Windows; return str unchanged.

fsencode() is the reverse function. New in version 3.2.

os.getenv(key, default=None)
Return the value of the environment variable key if it exists, or default if it doesn't. key, default and the result are str.

On Unix, keys and values are decoded with sys.getfilesystemencoding() and 'surrogateescape' error handler. Use os.getenvb() if you would like to use a different encoding.

Availability: most flavors of Unix, Windows.

os.getenvb(key, default=None)
Return the value of the environment variable key if it exists, or default if it doesn't. key, default and the result are bytes.

Availability: most flavors of Unix. New in version 3.2.

os.get_exec_path(env=None)
Returns the list of directories that will be searched for a named executable, similar to a shell, when launching a process. env, when specified, should be an environment variable dictionary to lookup the PATH in. By default, when env is None, environ is used. New in version 3.2.

os.getegid()
Return the effective group id of the current process. This corresponds to the "set id" bit on the file being executed in the current process.

Availability: Unix.

os.geteuid()
Return the current process’s effective user id.

Availability: Unix.

os.getgid()
Return the real group id of the current process.

Availability: Unix.

os.getgrouplist(user, group)
Return list of group ids that user belongs to. If group is not in the list, it is included; typically, group is specified as the group ID field from the password record for user.

Availability: Unix. New in version 3.3.

os.getgroups()
Return list of supplemental group ids associated with the current process.

Availability: Unix.

Note: On Mac OS X, getgroups() behavior differs somewhat from other Unix platforms. If the Python interpreter was built with a deployment target of 10.5 or earlier, getgroups() returns the list of effective group ids associated with the current user process; this list is limited to a system-defined number of entries, typically 16, and may be modified by calls to setgroups() if suitably privileged. If built with a deployment target greater than 10.5, getgroups() returns the current group access list for the user associated with the effective user id of the process; the group access list may change over the lifetime of the process, it is not affected by calls to setgroups(), and its length is not limited to 16. The deployment target value, MACOSX_DEPLOYMENT_TARGET, can be obtained with sysconfig.get_config_var().
os.getlogin()
Return the name of the user logged in on the controlling terminal of the process. For most purposes, it is more useful to use the environment variables
LOGNAME or USERNAME to find out who the user is, or pwd.getpwuid(os.getuid())[0] to get the login name of the currently effective user id.
Availability: Unix, Windows.

os.getpgid(pid)
Return the process group id of the process with process id pid. If pid is 0, the process group id of the current process is returned.
Availability: Unix.

os.getpgrp()
Return the id of the current process group.
Availability: Unix.

os.getpid()
Return the current process id.
Availability: Unix, Windows.

os.getppid()
Return the parent’s process id. When the parent process has exited, on Unix the id returned is the one of the init process (1), on Windows it is still the same id, which may be already reused by another process.

os.getpriority(which, who)
Get program scheduling priority. The value which is one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER, and who is interpreted relative to which (a process identifier for PRIO_PROCESS, process group identifier for PRIO_PGRP, and a user ID for PRIO_USER). A zero value for who denotes (respectively) the calling process, the process group of the calling process, or the real user ID of the calling process.
Availability: Unix. New in version 3.3.

os.PRIOR_PROCESS
os.PRIOR_PGRP
os.PRIOR_USER
Parameters for the getpriority() and setpriority() functions.
Availability: Unix. New in version 3.3.

os.getresuid()
Return a tuple (ruid, euid, suid) denoting the current process’s real, effective, and saved user ids.

os.getresgid()
Return a tuple (rgid, egid, sgid) denoting the current process’s real, effective, and saved group ids.

os.getuid()
Return the current process’s user id.
Availability: Unix.

os.initgroups(username, gid)
Call the system initgroups() to initialize the group access list with all of the groups of which the specified username is a member, plus the specified group id.

os.putenv(key, value)
Set the environment variable named key to the string value. Such changes to the environment affect sub-processes started with os.system(), popen() or fork() and execv().
Availability: most flavors of Unix, Windows.

**Note:** On some platforms, including FreeBSD and Mac OS X, setting `environ` may cause memory leaks. Refer to the system documentation for `putenv`.

When `putenv()` is supported, assignments to items in `os.environ` are automatically translated into corresponding calls to `putenv()`, however, calls to `putenv()` don’t update `os.environ`, so it is actually preferable to assign to items of `os.environ`.

`os.setegid(egid)`
Set the current process’s effective group id.

Availability: Unix.

`os.seteuid(euid)`
Set the current process’s effective user id.

Availability: Unix.

`os.setgid(gid)`
Set the current process’ group id.

Availability: Unix.

`os.setgroups(groups)`
Set the list of supplemental group ids associated with the current process to `groups`. `groups` must be a sequence, and each element must be an integer identifying a group. This operation is typically available only to the superuser.

Availability: Unix.

**Note:** On Mac OS X, the length of `groups` may not exceed the system-defined maximum number of effective group ids, typically 16. See the documentation for `getgroups()` for cases where it may not return the same group list set by calling `setgroups()`.

`os.setpgrp()`
Call the system call `setpgrp()` or `setpgrp(0, 0)` depending on which version is implemented (if any). See the Unix manual for the semantics.

Availability: Unix.

`os.setpgid(pid, pgrp)`
Call the system call `setpgid()` to set the process group id of the process with id `pid` to the process group with id `pgrp`. See the Unix manual for the semantics.

Availability: Unix.

`os.setpriority(which, who, priority)`
Set program scheduling priority. The value `which` is one of `PRIO_PROCESS`, `PRIO_PGRP`, or `PRIO_USER`, and `who` is interpreted relative to `which` (a process identifier for `PRIO_PROCESS`, process group identifier for `PRIO_PGRP`, and a user ID for `PRIO_USER`). A zero value for `who` denotes (respectively) the calling process, the process group of the calling process, or the real user ID of the calling process. `priority` is a value in the range -20 to 19. The default priority is 0; lower priorities cause more favorable scheduling.

Availability: Unix New in version 3.3.

`os.setregid(rgid, egid)`
Set the current process’s real and effective group ids.

Availability: Unix.

`os.setresgid(rgid, egid, sgid)`
Set the current process’s real, effective, and saved group ids.

`os.setresuid(ruid, euid, suid)`
Set the current process’s real, effective, and saved user ids.


`os.setreuid(ruid, euid)`
Set the current process’s real and effective user ids.

Availability: Unix.

`os.getsid(pid)`
Call the system call `getsid()`. See the Unix manual for the semantics.

Availability: Unix.

`os.setsid()`
Call the system call `setsid()`. See the Unix manual for the semantics.

Availability: Unix.

`os.setuid(uid)`
Set the current process’s user id.

Availability: Unix.

`os.strerror(code)`
Return the error message corresponding to the error code in `code`. On platforms where `strerror()` returns `NULL` when given an unknown error number, `ValueError` is raised.

Availability: Unix, Windows.

`os.supports_bytes_environ`  
True if the native OS type of the environment is bytes (e.g., False on Windows). New in version 3.2.

`os.umask(mask)`
Set the current numeric umask and return the previous umask.

Availability: Unix, Windows.

`os.uname()`
Returns information identifying the current operating system. The return value is an object with five attributes:

* `sysname` - operating system name
* `nodename` - name of machine on network (implementation-defined)
* `release` - operating system release
* `version` - operating system version
* `machine` - hardware identifier

For backwards compatibility, this object is also iterable, behaving like a five-tuple containing `sysname`, `nodename`, `release`, `version`, and `machine` in that order.

Some systems truncate `nodename` to 8 characters or to the leading component; a better way to get the hostname is `socket.gethostname()` or even `socket.gethostbyaddr(socket.gethostname())`.

Availability: recent flavors of Unix. Changed in version 3.3: Return type changed from a tuple to a tuple-like object with named attributes.

`os.unsetenv(key)`
Unset (delete) the environment variable named `key`. Such changes to the environment affect subprocesses started with `os.system()`, `popen()` or `fork()` and `execv()`.

392 Chapter 16. Generic Operating System Services
When `unsetenv()` is supported, deletion of items in `os.environ` is automatically translated into a corresponding call to `unsetenv()`. However, calls to `unsetenv()` don’t update `os.environ`, so it is actually preferable to delete items of `os.environ`.

Availability: most flavors of Unix, Windows.

## 16.1.3 File Object Creation

This function creates new *file objects*. (See also `open()` for opening file descriptors.)

```python
os.fdopen(fd, *args, **kwargs)
```

Return an open file object connected to the file descriptor `fd`. This is an alias of the `open()` built-in function and accepts the same arguments. The only difference is that the first argument of `fdopen()` must always be an integer.

## 16.1.4 File Descriptor Operations

These functions operate on I/O streams referenced using file descriptors.

File descriptors are small integers corresponding to a file that has been opened by the current process. For example, standard input is usually file descriptor 0, standard output is 1, and standard error is 2. Further files opened by a process will then be assigned 3, 4, 5, and so forth. The name “file descriptor” is slightly deceptive; on Unix platforms, sockets and pipes are also referenced by file descriptors.

The `fileno()` method can be used to obtain the file descriptor associated with a *file object* when required. Note that using the file descriptor directly will bypass the file object methods, ignoring aspects such as internal buffering of data.

```python
os.close(fd)
```

Close file descriptor `fd`.

Availability: Unix, Windows.

**Note:** This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `pipe()`. To close a “file object” returned by the built-in function `open()` or by `popen()` or `fdopen()`, use its `close()` method.

```python
os.closerange(fd_low, fd_high)
```

Close all file descriptors from `fd_low` (inclusive) to `fd_high` (exclusive), ignoring errors. Equivalent to (but much faster than):

```python
for fd in range(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

Availability: Unix, Windows.

```python
os.device_encoding(fd)
```

Return a string describing the encoding of the device associated with `fd` if it is connected to a terminal; else return `None`.

```python
os.dup(fd)
```

Return a duplicate of file descriptor `fd`.

Availability: Unix, Windows.

```python
os.dup2(fd, fd2)
```

Duplicate file descriptor `fd` to `fd2`, closing the latter first if necessary.
Availability: Unix, Windows.

`os.fchmod(fd, mode)`
Change the mode of the file given by `fd` to the numeric `mode`. See the docs for `chmod()` for possible values of `mode`. As of Python 3.3, this is equivalent to `os.chmod(fd, mode)`.

Availability: Unix.

`os.fchown(fd, uid, gid)`
Change the owner and group id of the file given by `fd` to the numeric `uid` and `gid`. To leave one of the ids unchanged, set it to `-1`. See `chown()`. As of Python 3.3, this is equivalent to `os.chown(fd, uid, gid)`.

Availability: Unix.

`os.fdatasync(fd)`
Force write of file with filedescriptor `fd` to disk. Does not force update of metadata.

Availability: Unix.

Note: This function is not available on MacOS.

`os.fpathconf(fd, name)`
Return system configuration information relevant to an open file. `name` specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the `pathconf_names` dictionary. For configuration variables not included in that mapping, passing an integer for `name` is also accepted.

If `name` is a string and is not known, `ValueError` is raised. If a specific value for `name` is not supported by the host system, even if it is included in `pathconf_names`, an `OSError` is raised with `errno.EINVAL` for the error number.

As of Python 3.3, this is equivalent to `os.pathconf(fd, name)`.

Availability: Unix.

`os.fstat(fd)`
Return status for file descriptor `fd`, like `stat()`. As of Python 3.3, this is equivalent to `os.stat(fd)`.

Availability: Unix, Windows.

`os.fstatvfs(fd)`
Return information about the filesystem containing the file associated with file descriptor `fd`, like `statvfs()`. As of Python 3.3, this is equivalent to `os.statvfs(fd)`.

Availability: Unix.

`os.fsync(fd)`
Force write of file with filedescriptor `fd` to disk. On Unix, this calls the native `fsync()` function; on Windows, the MS _commit() function.

If you’re starting with a buffered Python file object `f`, first do `f.flush()`, and then do `os.fsync(f.fileno())`, to ensure that all internal buffers associated with `f` are written to disk.

Availability: Unix, Windows.

`os.ftruncate(fd, length)`
Truncate the file corresponding to file descriptor `fd`, so that it is at most `length` bytes in size. As of Python 3.3, this is equivalent to `os.truncate(fd, length)`.

Availability: Unix.

`os.isatty(fd)`
Return True if the file descriptor `fd` is open and connected to a tty(-like) device, else False.
**os.lockf** (*fd, cmd, len*)

Apply, test or remove a POSIX lock on an open file descriptor. *fd* is an open file descriptor. *cmd* specifies the command to use - one of *F_LOCK*, *F_TLOCK*, *F_ULOCK* or *F_TEST*. *len* specifies the section of the file to lock.

Availability: Unix. New in version 3.3.

- **os.F_LOCK**
- **os.F_TLOCK**
- **os.F_ULOCK**
- **os.F_TEST**

Flags that specify what action lockf() will take.

Availability: Unix. New in version 3.3.

**os.lseek** (*fd, pos, how*)

Set the current position of file descriptor *fd* to position *pos*, modified by *how*: SEEK_SET or 0 to set the position relative to the beginning of the file; SEEK_CUR or 1 to set it relative to the current position; SEEK_END or 2 to set it relative to the end of the file. Return the new cursor position in bytes, starting from the beginning.

Availability: Unix, Windows.

- **os.SEEK_SET**
- **os.SEEK_CUR**
- **os.SEEK_END**

Parameters to the lseek() function. Their values are 0, 1, and 2, respectively.

Availability: Unix, Windows. New in version 3.3: Some operating systems could support additional values, like os.SEEK_HOLE or os.SEEK_DATA.

**os.open** (*file, flags, mode=0o777, *, dir_fd=None*)

Open the file *file* and set various flags according to *flags* and possibly its mode according to *mode*. When computing *mode*, the current umask value is first masked out. Return the file descriptor for the newly opened file.

For a description of the flag and mode values, see the C run-time documentation; flag constants (like O_RDONLY and O_WRONLY) are defined in this module too (see open() flag constants). In particular, on Windows adding O_BINARY is needed to open files in binary mode.

This function can support paths relative to directory descriptors.

Availability: Unix, Windows.

**Note:** This function is intended for low-level I/O. For normal usage, use the built-in function open(), which returns a file object with read() and write() methods (and many more). To wrap a file descriptor in a file object, use fdopen().

New in version 3.3: The dir_fd argument.

**os.openpty()**

Open a new pseudo-terminal pair. Return a pair of file descriptors (master, slave) for the pty and the tty, respectively. For a (slightly) more portable approach, use the pty module.

Availability: some flavors of Unix.

**os.pipe()**

Create a pipe. Return a pair of file descriptors (*r, w*) usable for reading and writing, respectively.

Availability: Unix, Windows.

**os.pipe2**(flags)

Create a pipe with flags set atomically. flags can be constructed by ORing together one or more of these values: O_NONBLOCK, O_CLOEXEC. Return a pair of file descriptors (*r, w*) usable for reading and writing, respectively.
Availability: some flavors of Unix. New in version 3.3.

```python
os.posix_fallocate(fd, offset, len)
```
Ensures that enough disk space is allocated for the file specified by `fd` starting from `offset` and continuing for `len` bytes.

Availability: Unix. New in version 3.3.

```python
os.posix_fadvise(fd, offset, len, advice)
```
Announces an intention to access data in a specific pattern thus allowing the kernel to make optimizations. The advice applies to the region of the file specified by `fd` starting at `offset` and continuing for `len` bytes. `advice` is one of `POSIX_FADV_NORMAL`, `POSIX_FADV_SEQUENTIAL`, `POSIX_FADV_RANDOM`, `POSIX_FADV_NOREUSE`, `POSIX_FADV_WILLNEED` or `POSIX_FADV_DONTNEED`.

Availability: Unix. New in version 3.3.

```python
os.POSIX_FADV_NORMAL
os.POSIX_FADV_SEQUENTIAL
os.POSIX_FADV_RANDOM
os.POSIX_FADV_NOREUSE
os.POSIX_FADV_WILLNEED
os.POSIX_FADV_DONTNEED
```
Flags that can be used in `advice` in `posix_fadvise()` that specify the access pattern that is likely to be used.

Availability: Unix. New in version 3.3.

```python
os.pread(fd, buffersize, offset)
```
Read from a file descriptor, `fd`, at a position of `offset`. It will read up to `buffersize` number of bytes. The file offset remains unchanged.

Availability: Unix. New in version 3.3.

```python
os.pwrite(fd, string, offset)
```
Write `string` to a file descriptor, `fd`, from `offset`, leaving the file offset unchanged.

Availability: Unix. New in version 3.3.

```python
os.read(fd, n)
```
Read at most `n` bytes from file descriptor `fd`. Return a bytestring containing the bytes read. If the end of the file referred to by `fd` has been reached, an empty bytes object is returned.

Availability: Unix, Windows.

#### Note:
This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `pipe()`. To read a “file object” returned by the built-in function `open()` or by `popen()` or `fdopen()`, or `sys.stdin`, use its `read()` or `readline()` methods.

```python
os.sendfile(out, in, offset, nbytes)
```
Copy `nbytes` bytes from file descriptor `in` to file descriptor `out` starting at `offset`. Return the number of bytes sent. When EOF is reached return 0.

The first function notation is supported by all platforms that define `sendfile()`.

On Linux, if `offset` is given as None, the bytes are read from the current position of `in` and the position of `in` is updated.

The second case may be used on Mac OS X and FreeBSD where `headers` and `trailers` are arbitrary sequences of buffers that are written before and after the data from `in` is written. It returns the same as the first case.

On Mac OS X and FreeBSD, a value of 0 for `nbytes` specifies to send until the end of `in` is reached.

All platforms support sockets as `out` file descriptor, and some platforms allow other types (e.g. regular file, pipe) as well.
Availability: Unix. New in version 3.3.

`os.SF_NODISKIO`

`os.SF_MNOWAIT`

`os.SF_SYNC`

Parameters to the `sendfile()` function, if the implementation supports them.

Availability: Unix. New in version 3.3.

`os.readv(fd, buffers)`

Read from a file descriptor into a number of writable buffers. `buffers` is an arbitrary sequence of writable buffers. Returns the total number of bytes read.

Availability: Unix. New in version 3.3.

`os.tcgetpgrp(fd)`

Return the process group associated with the terminal given by `fd` (an open file descriptor as returned by `os.open()`).

Availability: Unix.

`os.tcsetpgrp(fd, pg)`

Set the process group associated with the terminal given by `fd` (an open file descriptor as returned by `os.open()`) to `pg`.

Availability: Unix.

`os.ttyname(fd)`

Return a string which specifies the terminal device associated with file descriptor `fd`. If `fd` is not associated with a terminal device, an exception is raised.

Availability: Unix.

`os.write(fd, str)`

Write the bytestring in `str` to file descriptor `fd`. Return the number of bytes actually written.

Availability: Unix, Windows.

**Note:** This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `pipe()`. To write a “file object” returned by the built-in function `open()` or by `popen()` or `fdopen()`, or `sys.stdout` or `sys.stderr`, use its `write()` method.

`os.writev(fd, buffers)`

Write the contents of `buffers` to file descriptor `fd`, where `buffers` is an arbitrary sequence of buffers. Returns the total number of bytes written.

Availability: Unix. New in version 3.3.

### `open()` Flag Constants

The following constants are options for the `flags` parameter to the `open()` function. They can be combined using the bitwise OR operator `|`. Some of them are not available on all platforms. For descriptions of their availability and use, consult the `open(2)` manual page on Unix or the MSDN on Windows.

`os.O_RDONLY`

`os.O_WRONLY`

`os.O_RDWR`

`os.O_APPEND`

`os.O_CREAT`

`os.O_EXCL`

`os.O_TRUNC`

These constants are available on Unix and Windows.

`os.O_DSYNC`
os.O_RSYNC
os.O_SYNC
os.O_NDELAY
os.O_NONBLOCK
os.O_NOCTTY
os.O_SHLOCK
os.O_EXLOCK
os.O_CLOEXEC

These constants are only available on Unix. Changed in version 3.3: Add O_CLOEXEC constant.

os.O_BINARY
os.O_NOINHERIT
os.O_SHORT_LIVED
os.O_TEMPORARY
os.O_RANDOM
os.O_SEQUENTIAL
os.O_TEXT

These constants are only available on Windows.

os.O_ASYNC
os.O_DIRECT
os.O_DIRECTORY
os.O_NOFOLLOW
os.O_NOATIME

These constants are GNU extensions and not present if they are not defined by the C library.

os.RTLD_LAZY
os.RTLD_NOW
os.RTLD_GLOBAL
os.RTLD_LOCAL
os.RTLD_NODELETE
os.RTLD_NOLOAD
os.RTLD_DEEPBIND

See the Unix manual page dlopen(3). New in version 3.3.

Querying the size of a terminal

New in version 3.3.

os.get_terminal_size(fd=STDOUT_FILENO)

Return the size of the terminal window as (columns, lines), tuple of type terminal_size.

The optional argument fd (default STDOUT_FILENO, or standard output) specifies which file descriptor should be queried.

If the file descriptor is not connected to a terminal, an OSError is raised.

shutil.get_terminal_size() is the high-level function which should normally be used, os.get_terminal_size is the low-level implementation.

Availability: Unix, Windows.

class os.terminal_size
A subclass of tuple, holding (columns, lines) of the terminal window size.

columns
   Width of the terminal window in characters.

lines
   Height of the terminal window in characters.
16.1.5 Files and Directories

On some Unix platforms, many of these functions support one or more of these features:

- **specifying a file descriptor:** For some functions, the *path* argument can be not only a string giving a path name, but also a file descriptor. The function will then operate on the file referred to by the descriptor. (For POSIX systems, Python will call the *f...* version of the function.)

  You can check whether or not *path* can be specified as a file descriptor on your platform using `os.supports_fd`. If it is unavailable, using it will raise a `NotImplementedError`.

  If the function also supports *dir_fd* or *follow_symlinks* arguments, it is an error to specify one of those when supplying *path* as a file descriptor.

- **paths relative to directory descriptors:** If *dir_fd* is not `None`, it should be a file descriptor referring to a directory, and the path to operate on should be relative; *path* will then be relative to that directory. If the path is absolute, *dir_fd* is ignored. (For POSIX systems, Python will call the *l...at* or *f...at* version of the function.)

  You can check whether or not *dir_fd* is supported on your platform using `os.supports_dir_fd`. If it is unavailable, using it will raise a `NotImplementedError`.

- **not following symlinks:** If *follow_symlinks* is `False`, and the last element of the path to operate on is a symbolic link, the function will operate on the symbolic link itself instead of the file the link points to. (For POSIX systems, Python will call the *l...at* version of the function.)

  You can check whether or not *follow_symlinks* is supported on your platform using `os.supports_follow_symlinks`. If it is unavailable, using it will raise a `NotImplementedError`.

`os.access(path, mode, *, dir_fd=None, effective_ids=False, follow_symlinks=True)`

Use the real uid/gid to test for access to *path*. Note that most operations will use the effective uid/gid, therefore this routine can be used in a suid/sgid environment to test if the invoking user has the specified access to *path*. *mode* should be `F_OK` to test the existence of *path*, or it can be the inclusive OR of one or more of `R_OK`, `W_OK`, and `X_OK` to test permissions. Return `True` if access is allowed, `False` if not. See the Unix man page `access(2)` for more information.

This function can support specifying *paths relative to directory descriptors* and *not following symlinks*.

If *effective_ids* is `True`, `access()` will perform its access checks using the effective uid/gid instead of the real uid/gid. *effective_ids* may not be supported on your platform; you can check whether or not it is available using `os.supports_effective_ids`. If it is unavailable, using it will raise a `NotImplementedError`.

Availability: Unix, Windows.

**Note:** Using `access()` to check if a user is authorized to e.g. open a file before actually doing so using `open()` creates a security hole, because the user might exploit the short time interval between checking and opening the file to manipulate it. It’s preferable to use EAFP techniques. For example:

```python
if os.access("myfile", os.R_OK):
    with open("myfile") as fp:
        return fp.read()
return "some default data"
```

is better written as:

```python
try:
    fp = open("myfile")
except PermissionError:
    return "some default data"
else:
    return os.read(fp)
```
with fp:
    return fp.read()

Note: I/O operations may fail even when access() indicates that they would succeed, particularly for operations on network filesystems which may have permissions semantics beyond the usual POSIX permission-bit model.

Changed in version 3.3: Added the dir_fd, effective_ids, and follow_symlinks parameters.

os.F_OK
os.R_OK
os.W_OK
os.X_OK

Values to pass as the mode parameter of access() to test the existence, readability, writability and executability of path, respectively.

os.chdir(path)

Change the current working directory to path.

This function can support specifying a file descriptor. The descriptor must refer to an opened directory, not an open file.

Availability: Unix, Windows. New in version 3.3: Added support for specifying path as a file descriptor on some platforms.

os.chflags(path, flags, *, follow_symlinks=True)

Set the flags of path to the numeric flags. flags may take a combination (bitwise OR) of the following values (as defined in the stat module):

• stat.UF_NODUMP
• stat.UF_IMMUTABLE
• stat.UF_APPEND
• stat.UF_OPAQUE
• stat.UF_NOUNLINK
• stat.UF_COMPRESSED
• stat.UF_HIDDEN
• stat.SF_ARCHIVED
• stat.SF_IMMUTABLE
• stat.SF_APPEND
• stat.SF_NOUNLINK
• stat.SF_SNAPSHOT

This function can support not following symlinks.


os.chmod(path, mode, *, dir_fd=None, follow_symlinks=True)

Change the mode of path to the numeric mode. mode may take one of the following values (as defined in the stat module) or bitwise ORed combinations of them:

• stat.S_ISUID
• stat.S_ISGID
• stat.S_ENFMT
• stat.S_ISVTX
• `stat.S_IREAD`
• `stat.S_IWRITE`
• `stat.S_IEXEC`
• `stat.S_IRWXU`
• `stat.S_IRUSR`
• `stat.S_IWUSR`
• `stat.S_IXUSR`
• `stat.S_IRWXG`
• `stat.S_IRGRP`
• `stat.S_IWGRP`
• `stat.S_IXGRP`
• `stat.S_IRWXO`
• `stat.S_IROTH`
• `stat.S_IWOTH`
• `stat.S_IXOTH`

This function can support specifying a file descriptor, paths relative to directory descriptors and not following symlinks.

Availability: Unix, Windows.

Note: Although Windows supports `chmod()`, you can only set the file’s read-only flag with it (via the `stat.S_IWRITE` and `stat.S_IREAD` constants or a corresponding integer value). All other bits are ignored.

New in version 3.3: Added support for specifying path as an open file descriptor, and the `dir_fd` and `follow_symlinks` arguments.

```python
os.chown(path, uid, gid, *, dir_fd=None, follow_symlinks=True)
```
Change the owner and group id of `path` to the numeric `uid` and `gid`. To leave one of the ids unchanged, set it to -1.

This function can support specifying a file descriptor, paths relative to directory descriptors and not following symlinks.

See `shutil.chown()` for a higher-level function that accepts names in addition to numeric ids.


```python
os.chroot(path)
```
Change the root directory of the current process to `path`.

Availability: Unix.

```python
os.fchdir(fd)
```
Change the current working directory to the directory represented by the file descriptor `fd`. The descriptor must refer to an opened directory, not an open file. As of Python 3.3, this is equivalent to `os.chdir(fd)`.

Availability: Unix.

```python
os.getcwd()
```
Return a string representing the current working directory.

Availability: Unix, Windows.
The Python Library Reference, Release 3.3.3

```

os.getcwdb()
  Return a bytestring representing the current working directory.
  Availability: Unix, Windows.

os.lchflags(path, flags)
  Set the flags of path to the numeric flags, like chflags(), but do not follow symbolic links. As of Python 3.3, this is equivalent to os.chflags(path, flags, follow_symlinks=False).
  Availability: Unix.

os.lchmod(path, mode)
  Change the mode of path to the numeric mode. If path is a symlink, this affects the symlink rather than the target. See the docs for chmod() for possible values of mode. As of Python 3.3, this is equivalent to os.chmod(path, mode, follow_symlinks=False).
  Availability: Unix.

os.lchown(path, uid, gid)
  Change the owner and group id of path to the numeric uid and gid. This function will not follow symbolic links. As of Python 3.3, this is equivalent to os.chown(path, uid, gid, follow_symlinks=False).
  Availability: Unix.

os.link(src, dst, *, src_dir_fd=None, dst_dir_fd=None, follow_symlinks=True)
  Create a hard link pointing to src named dst.
  This function can support specifying src_dir_fd and/or dst_dir_fd to supply paths relative to directory descriptors, and not following symlinks.

os.listdir(path='')
  Return a list containing the names of the entries in the directory given by path. The list is in arbitrary order, and does not include the special entries '.' and '..' even if they are present in the directory.
  path may be either of type str or of type bytes. If path is of type bytes, the filenames returned will also be of type bytes; in all other circumstances, they will be of type str.
  This function can also support specifying a file descriptor; the file descriptor must refer to a directory.

  Note: To encode str filenames to bytes, use fsencode().


os.lstat(path, *, dir_fd=None)
  Perform the equivalent of an lstat() system call on the given path. Similar to stat(), but does not follow symbolic links. On platforms that do not support symbolic links, this is an alias for stat(). As of Python 3.3, this is equivalent to os.stat(path, dir_fd=dir_fd, follow_symlinks=False).
  This function can also support paths relative to directory descriptors. Changed in version 3.2: Added support for Windows 6.0 (Vista) symbolic links. Changed in version 3.3: Added the dir_fd parameter.

os.mkdir(path, mode=0o777, *, dir_fd=None)
  Create a directory named path with numeric mode mode.
  On some systems, mode is ignored. Where it is used, the current umask value is first masked out. If the directory already exists, OSError is raised.
  This function can also support paths relative to directory descriptors.
  It is also possible to create temporary directories; see the tempfile module’s tempfile.mkdtemp() function.

402 Chapter 16. Generic Operating System Services
```

```python
os.makedirs(path, mode=0o777, exist_ok=False)
```

Recursive directory creation function. Like `mkdir()`, but makes all intermediate-level directories needed to contain the leaf directory.

The default mode is `0o777` (octal). On some systems, mode is ignored. Where it is used, the current umask value is first masked out.

If `exist_ok` is `False` (the default), an `OSError` is raised if the target directory already exists. If `exist_ok` is `True` an `OSError` is still raised if the umask-masked mode is different from the existing mode, on systems where the mode is used. `OSError` will also be raised if the directory creation fails.

**Note:** `makedirs()` will become confused if the path elements to create include `pardir` (e.g. `".."` on UNIX systems).

This function handles UNC paths correctly. New in version 3.2: The `exist_ok` parameter.

```python
os.mkfifo(path, mode=0o666, *, dir_fd=None)
```

Create a FIFO (a named pipe) named `path` with numeric mode `mode`. The current umask value is first masked out from the mode.

This function can also support paths relative to directory descriptors.

FIFOs are pipes that can be accessed like regular files. FIFOs exist until they are deleted (for example with `os.unlink()`). Generally, FIFOs are used as rendezvous between “client” and “server” type processes: the server opens the FIFO for reading, and the client opens it for writing. Note that `mkfifo()` doesn’t open the FIFO — it just creates the rendezvous point.


```python
os.mknod(filename, mode=0o600, device=0, *, dir_fd=None)
```

Create a filesystem node (file, device special file or named pipe) named `filename`. `mode` specifies both the permissions to use and the type of node to be created, being combined (bitwise OR) with one of `stat.S_IFREG`, `stat.S_IFCHR`, `stat.S_IFBLK`, and `stat.S_IFIFO` (those constants are available in `stat`). For `stat.S_IFCHR` and `stat.S_IFBLK`, `device` defines the newly created device special file (probably using `os.makedev()`), otherwise it is ignored.

This function can also support paths relative to directory descriptors. New in version 3.3: The `dir_fd` argument.

```python
os.major(device)
```

Extract the device major number from a raw device number (usually the `st_dev` or `st_rdev` field from `stat`).

```python
os.minor(device)
```

Extract the device minor number from a raw device number (usually the `st_dev` or `st_rdev` field from `stat`).

```python
os.makedev(major, minor)
```

Compose a raw device number from the major and minor device numbers.

```python
os.pathconf(path, name)
```

Return system configuration information relevant to a named file. `name` specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the `pathconf_names` dictionary. For configuration variables not included in that mapping, passing an integer for `name` is also accepted.

If `name` is a string and is not known, `ValueError` is raised. If a specific value for `name` is not supported by the host system, even if it is included in `pathconf_names`, an `OSError` is raised with `errno.EINVAL` for the error number.

This function can support specifying a file descriptor.
Availability: Unix.

**os.pathconf_names**
Dictionary mapping names accepted by `pathconf()` and `fpathconf()` to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

Availability: Unix.

**os.readlink** *(path, *, dir_fd=None)*
Return a string representing the path to which the symbolic link points. The result may be either an absolute or relative pathname; if it is relative, it may be converted to an absolute pathname using `os.path.join(os.path.dirname(path), result)`.

If the `path` is a string object, the result will also be a string object, and the call may raise an `UnicodeDecodeError`. If the `path` is a bytes object, the result will be a bytes object.

This function can also support paths relative to directory descriptors.


**os.remove** *(path, *, dir_fd=None)*
Remove (delete) the file `path`. If `path` is a directory, `OSError` is raised. Use `rmdir()` to remove directories.

This function can support paths relative to directory descriptors.

On Windows, attempting to remove a file that is in use causes an exception to be raised; on Unix, the directory entry is removed but the storage allocated to the file is not made available until the original file is no longer in use.

This function is identical to `unlink()`.


**os.removedirs** *(path)*
Remove directories recursively. Works like `rmdir()` except that, if the leaf directory is successfully removed, `removedirs()` tries to successively remove every parent directory mentioned in `path` until an error is raised (which is ignored, because it generally means that a parent directory is not empty). For example, `os.removedirs('foo/bar/baz')` will first remove the directory `‘foo/bar/baz’`, and then remove `‘foo/bar’` and `‘foo’` if they are empty. Raises `OSError` if the leaf directory could not be successfully removed.

**os.rename** *(src, dst, *, src_dir_fd=None, dst_dir_fd=None)*
Rename the file or directory `src` to `dst`. If `dst` is a directory, `OSError` will be raised. On Unix, if `dst` exists and is a file, it will be replaced silently if the user has permission. The operation may fail on some Unix flavors if `src` and `dst` are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement). On Windows, if `dst` already exists, `OSError` will be raised even if it is a file.

This function can support specifying `src_dir_fd` and/or `dst_dir_fd` to supply paths relative to directory descriptors.

If you want cross-platform overwriting of the destination, use `replace()`.


**os.renames** *(old, new)*
Recursive directory or file renaming function. Works like `rename()`, except creation of any intermediate directories needed to make the new pathname good is attempted first. After the rename, directories corresponding to rightmost path segments of the old name will be pruned away using `removedirs()`.

**Note:** This function can fail with the new directory structure made if you lack permissions needed to remove the leaf directory or file.
os.replace(src, dst, *, src_dir_fd=None, dst_dir_fd=None)
Rename the file or directory src to dst. If dst is a directory, OSError will be raised. If dst exists and is a file, it will be replaced silently if the user has permission. The operation may fail if src and dst are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement).

This function can support specifying src_dir_fd and/or dst_dir_fd to supply paths relative to directory descriptors.


os.rmdir(path, *, dir_fd=None)
Remove (delete) the directory path. Only works when the directory is empty, otherwise, OSError is raised. In order to remove whole directory trees, shutil.rmtree() can be used.

This function can support paths relative to directory descriptors.


os.stat(path, *, dir_fd=None, follow_symlinks=True)
Perform the equivalent of a stat() system call on the given path. path may be specified as either a string or as an open file descriptor. (This function normally follows symlinks; to stat a symlink add the argument follow_symlinks=False, or use lstat().)

The return value is an object whose attributes correspond roughly to the members of the stat structure, namely:

- st_mode - protection bits,
- st_ino - inode number,
- st_dev - device,
- st_nlink - number of hard links,
- st_uid - user id of owner,
- st_gid - group id of owner,
- st_size - size of file, in bytes,
- st_atime - time of most recent access expressed in seconds,
- st_mtime - time of most recent content modification expressed in seconds,
- st_ctime - platform dependent; time of most recent metadata change on Unix, or the time of creation on Windows, expressed in seconds
- st_atime_ns - time of most recent access expressed in nanoseconds as an integer,
- st_mtime_ns - time of most recent content modification expressed in nanoseconds as an integer,
- st_ctime_ns - platform dependent; time of most recent metadata change on Unix, or the time of creation on Windows, expressed in nanoseconds as an integer

On some Unix systems (such as Linux), the following attributes may also be available:

- st_blocks - number of 512-byte blocks allocated for file
- st_blksize - filesystem blocksize for efficient file system I/O
- st_rdev - type of device if an inode device
- st_flags - user defined flags for file

On other Unix systems (such as FreeBSD), the following attributes may be available (but may be only filled out if root tries to use them):

- st_gen - file generation number
- st_birthtime - time of file creation

On Mac OS systems, the following attributes may also be available:
• st_rsize
• st_creator
• st_type

Note: The exact meaning and resolution of the st_atime, st_mtime, and st_ctime attributes depend on the operating system and the file system. For example, on Windows systems using the FAT or FAT32 file systems, st_mtime has 2-second resolution, and st_atime has only 1-day resolution. See your operating system documentation for details. Similarly, although st_atime_ns, st_mtime_ns, and st_ctime_ns are always expressed in nanoseconds, many systems do not provide nanosecond precision. On systems that do provide nanosecond precision, the floating-point object used to store st_atime, st_mtime, and st_ctime cannot preserve all of it, and as such will be slightly inexact. If you need the exact timestamps you should always use st_atime_ns, st_mtime_ns, and st_ctime_ns.

For backward compatibility, the return value of stat() is also accessible as a tuple of at least 10 integers giving the most important (and portable) members of the stat structure, in the order st_mode, st_ino, st_dev, st_nlink, st_uid, st_gid, st_size, st_atime, st_mtime, st_ctime. More items may be added at the end by some implementations.

This function can support specifying a file descriptor and not following symlinks.

The standard module stat defines functions and constants that are useful for extracting information from a stat structure. (On Windows, some items are filled with dummy values.)

Example:

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
posix.stat_result(st_mode=33188, st_ino=7876932, st_dev=234881026,
st_nlink=1, st_uid=501, st_gid=501, st_size=264, st_atime=1297230295,
st_mtime=1297230027, st_ctime=1297230027)
>>> statinfo.st_size
264
```


os.stat_float_times([newvalue])
Determine whether stat_result represents time stamps as float objects. If newvalue is True, future calls to stat() return floats, if it is False, future calls return ints. If newvalue is omitted, return the current setting.

For compatibility with older Python versions, accessing stat_result as a tuple always returns integers.

Python now returns float values by default. Applications which do not work correctly with floating point time stamps can use this function to restore the old behaviour.

The resolution of the timestamps (that is the smallest possible fraction) depends on the system. Some systems only support second resolution; on these systems, the fraction will always be zero.

It is recommended that this setting is only changed at program startup time in the __main__ module; libraries should never change this setting. If an application uses a library that works incorrectly if floating point time stamps are processed, this application should turn the feature off until the library has been corrected. Deprecated since version 3.3.

os.statvfs(path)
Perform a statvfs() system call on the given path. The return value is an object whose attributes describe the filesystem on the given path, and correspond to the members of the statvfs structure, namely: f_bsize, f_frsize, f_blocks, f_bfree, f_bavail, f_files, f_ffree, f_favail, f_flag, f_namemax.
Two module-level constants are defined for the \texttt{f\_flag} attribute’s bit-flags: if \texttt{ST\_RDONLY} is set, the filesystem is mounted read-only, and if \texttt{ST\_NOSUID} is set, the semantics of setuid/setgid bits are disabled or not supported.

This function can support specifying a file descriptor. Changed in version 3.2: The \texttt{ST\_RDONLY} and \texttt{ST\_NOSUID} constants were added. Availability: Unix. New in version 3.3: Added support for specifying an open file descriptor for \texttt{path}.

\texttt{os.supports\_dir\_fd}

A \texttt{Set} object indicating which functions in the \texttt{os} module permit use of their \texttt{dir\_fd} parameter. Different platforms provide different functionality, and an option that might work on one might be unsupported on another. For consistency’s sakes, functions that support \texttt{dir\_fd} always allow specifying the parameter, but will raise an exception if the functionality is not actually available.

To check whether a particular function permits use of its \texttt{dir\_fd} parameter, use the \texttt{in} operator on \texttt{supports\_dir\_fd}. As an example, this expression determines whether the \texttt{dir\_fd} parameter of \texttt{os.stat()} is locally available:

\begin{verbatim}
os.stat in os.supports_dir_fd
\end{verbatim}

Currently \texttt{dir\_fd} parameters only work on Unix platforms; none of them work on Windows. New in version 3.3.

\texttt{os.supports\_effective\_ids}

A \texttt{Set} object indicating which functions in the \texttt{os} module permit use of the \texttt{effective\_ids} parameter for \texttt{os.access()}. If the local platform supports it, the collection will contain \texttt{os.access()}, otherwise it will be empty.

To check whether you can use the \texttt{effective\_ids} parameter for \texttt{os.access()}, use the \texttt{in} operator on \texttt{supports\_dir\_fd}, like so:

\begin{verbatim}
os.access in os.supports_effective_ids
\end{verbatim}

Currently \texttt{effective\_ids} only works on Unix platforms; it does not work on Windows. New in version 3.3.

\texttt{os.supports\_fd}

A \texttt{Set} object indicating which functions in the \texttt{os} module permit specifying their \texttt{path} parameter as an open file descriptor. Different platforms provide different functionality, and an option that might work on one might be unsupported on another. For consistency’s sakes, functions that support \texttt{fd} always allow specifying the parameter, but will raise an exception if the functionality is not actually available.

To check whether a particular function permits specifying an open file descriptor for its \texttt{path} parameter, use the \texttt{in} operator on \texttt{supports\_fd}. As an example, this expression determines whether \texttt{os.chdir()} accepts open file descriptors when called on your local platform:

\begin{verbatim}
os.chdir in os.supports_fd
\end{verbatim}

New in version 3.3.

\texttt{os.supports\_follow\_symlinks}

A \texttt{Set} object indicating which functions in the \texttt{os} module permit use of their \texttt{follow\_symlinks} parameter. Different platforms provide different functionality, and an option that might work on one might be unsupported on another. For consistency’s sakes, functions that support \texttt{follow\_symlinks} always allow specifying the parameter, but will raise an exception if the functionality is not actually available.

To check whether a particular function permits use of its \texttt{follow\_symlinks} parameter, use the \texttt{in} operator on \texttt{supports\_follow\_symlinks}. As an example, this expression determines whether the \texttt{follow\_symlinks} parameter of \texttt{os.stat()} is locally available:

\begin{verbatim}
os.stat in os.supports_follow_symlinks
\end{verbatim}

New in version 3.3.
Create a symbolic link pointing to `source` named `link_name`.

On Windows, a symlink represents either a file or a directory, and does not morph to the target dynamically. If the target is present, the type of the symlink will be created to match. Otherwise, the symlink will be created as a directory if `target_is_directory` is `True` or a file symlink (the default) otherwise. On non-Windows platforms, `target_is_directory` is ignored.

Symbolic link support was introduced in Windows 6.0 (Vista). `symlink()` will raise a `NotImplementedError` on Windows versions earlier than 6.0.

This function can support **paths relative to directory descriptors**.

**Note:** On Windows, the `SeCreateSymbolicLinkPrivilege` is required in order to successfully create symlinks. This privilege is not typically granted to regular users but is available to accounts which can escalate privileges to the administrator level. Either obtaining the privilege or running your application as an administrator are ways to successfully create symlinks.

`OSError` is raised when the function is called by an unprivileged user.

Availability: Unix, Windows. Changed in version 3.2: Added support for Windows 6.0 (Vista) symbolic links.New in version 3.3: Added the `dir_fd` argument, and now allow `target_is_directory` on non-Windows platforms.

**os.sync()**

Force write of everything to disk.

Availability: Unix. New in version 3.3.

**os.truncate(path, length)**

Truncate the file corresponding to `path`, so that it is at most `length` bytes in size.

This function can support **specifying a file descriptor**.

Availability: Unix. New in version 3.3.

**os.unlink(path, *, dir_fd=None)**

Remove (delete) the file `path`. This function is identical to `remove()`; the `unlink` name is its traditional Unix name. Please see the documentation for `remove()` for further information.


**os.utime(path, times=None, *, ns=None, dir_fd=None, follow_symlinks=True)**

Set the access and modified times of the file specified by `path`.

`utime()` takes two optional parameters, `times` and `ns`. These specify the times set on `path` and are used as follows:

- **If `ns` is not `None`,** it must be a 2-tuple of the form `(atime_ns, mtime_ns)` where each member is an int expressing nanoseconds.
- **If `times` is not `None`,** it must be a 2-tuple of the form `(atime, mtime)` where each member is an int or float expressing seconds.
- **If `times` and `ns` are both `None`,** this is equivalent to specifying `ns=(atime_ns, mtime_ns)` where both times are the current time.

It is an error to specify tuples for both `times` and `ns`.

Whether a directory can be given for `path` depends on whether the operating system implements directories as files (for example, Windows does not). Note that the exact times you set here may not be returned by a subsequent `stat()` call, depending on the resolution with which your operating system records access and modification times; see `stat()`. The best way to preserve exact times is to use the `st_atime_ns` and `st_mtime_ns` fields from the `os.stat()` result object with the `ns` parameter to `utime`.

This function can support **specifying a file descriptor**, **paths relative to directory descriptors** and **not following symlinks**.

```python
os.walk(top, topdown=True, onerror=None, followlinks=False)
```

Generate the file names in a directory tree by walking the tree either top-down or bottom-up. For each directory in the tree rooted at directory `top` (including `top` itself), it yields a 3-tuple `(dirpath, dirnames, filenames)`.

- `dirpath` is a string, the path to the directory.
- `dirnames` is a list of the names of the subdirectories in `dirpath` (excluding `.` and `..`).
- `filenames` is a list of the names of the non-directory files in `dirpath`. Note that the names in the lists contain no path components. To get a full path (which begins with `top`) to a file or directory in `dirpath`, do `os.path.join(dirpath, name)`.

If optional argument `topdown` is `True` or not specified, the triple for a directory is generated before the triples for any of its subdirectories (directories are generated top-down). If `topdown` is `False`, the triple for a directory is generated after the triples for all of its subdirectories (directories are generated bottom-up).

When `topdown` is `True`, the caller can modify the `dirnames` list in-place (perhaps using `del` or slice assignment), and `walk()` will only recurse into the subdirectories whose names remain in `dirnames`; this can be used to prune the search, impose a specific order of visiting, or even to inform `walk()` about directories the caller creates or renames before it resumes `walk()` again. Modifying `dirnames` when `topdown` is `False` is ineffective, because in bottom-up mode the directories in `dirnames` are generated before `dirpath` itself is generated.

By default, errors from the `listdir()` call are ignored. If optional argument `onerror` is specified, it should be a function; it will be called with one argument, an `OSError` instance. It can report the error to continue with the walk, or raise the exception to abort the walk. Note that the filename is available as the `filename` attribute of the exception object.

By default, `walk()` will not walk down into symbolic links that resolve to directories. Set `followlinks` to `True` to visit directories pointed to by symlinks, on systems that support them.

---

**Note:** Be aware that setting `followlinks` to `True` can lead to infinite recursion if a link points to a parent directory of itself. `walk()` does not keep track of the directories it visited already.

---

**Note:** If you pass a relative pathname, don’t change the current working directory between resumptions of `walk()`. `walk()` never changes the current directory, and assumes that its caller doesn’t either.

---

This example displays the number of bytes taken by non-directory files in each directory under the starting directory, except that it doesn’t look under any CVS subdirectory:

```python
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print(root, "consumes", end=" ")
    print(sum(getsize(join(root, name)) for name in files), end=" ")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

In the next example, walking the tree bottom-up is essential: `rmdir()` doesn’t allow deleting a directory before the directory is empty:

```python
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION: This is dangerous! For example, if top == "/", it
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
```

---

16.1. os — Miscellaneous operating system interfaces
for name in files:
    os.remove(os.path.join(root, name))
for name in dirs:
    os.rmdir(os.path.join(root, name))

os.fwalk(top='.', topdown=True, onerror=None, *, follow_symlinks=False, dir_fd=None)

This behaves exactly like walk(), except that it yields a 4-tuple (dirpath, dirnames, filenames, dirfd), and it supports dir_fd.
dirpath, dirnames and filenames are identical to walk() output, and dirfd is a file descriptor referring to the directory dirpath.

This function always supports paths relative to directory descriptors and not following symlinks. Note however that, unlike other functions, the fwalk() default value for follow_symlinks is False.

Note: Since fwalk() yields file descriptors, those are only valid until the next iteration step, so you should duplicate them (e.g. with dup()) if you want to keep them longer.

This example displays the number of bytes taken by non-directory files in each directory under the starting directory, except that it doesn’t look under any CVS subdirectory:

import os
for root, dirs, files, rootfd in os.fwalk('python/Lib/email'):
    print(root, "consumes", end="")
    print(sum([os.stat(name, dir_fd=rootfd).st_size for name in files]), end="")
    print("bytes in", len(files), "non-directory files")
if ’CVS’ in dirs:
    dirs.remove(’CVS’)  # don’t visit CVS directories

In the next example, walking the tree bottom-up is essential: rmdir() doesn’t allow deleting a directory before the directory is empty:

# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION: This is dangerous! For example, if top == ’/’, it
# could delete all your disk files.
import os
for root, dirs, files, rootfd in os.fwalk(top, topdown=False):
    for name in files:
        os.unlink(name, dir_fd=rootfd)
    for name in dirs:
        os.rmdir(name, dir_fd=rootfd)

Availability: Unix. New in version 3.3.

Linux extended attributes

New in version 3.3. These functions are all available on Linux only.

os.getxattr(path, attribute, *, follow_symlinks=True)

Return the value of the extended filesystem attribute attribute for path. attribute can be bytes or str. If it is str, it is encoded with the filesystem encoding.

This function can support specifying a file descriptor and not following symlinks.

os.listxattr(path=None, *, follow_symlinks=True)

Return a list of the extended filesystem attributes on path. The attributes in the list are represented as strings decoded with the filesystem encoding. If path is None, listxattr() will examine the current directory.
This function can support specifying a file descriptor and not following symlinks.

```python
os.removeattr(path, attribute, *, follow_symlinks=True)
```

Removes the extended filesystem attribute `attribute` from `path`. `attribute` should be bytes or str. If it is a string, it is encoded with the filesystem encoding.

This function can support specifying a file descriptor and not following symlinks.

```python
os.setxattr(path, attribute, value, flags=0, *, follow_symlinks=True)
```

Set the extended filesystem attribute `attribute` on `path` to `value`. `attribute` must be a bytes or str with no embedded NULs. If it is a str, it is encoded with the filesystem encoding. `flags` may be `XATTR_REPLACE` or `XATTR_CREATE`. If `XATTR_REPLACE` is given and the attribute does not exist, `EEXISTS` will be raised. If `XATTR_CREATE` is given and the attribute already exists, the attribute will not be created and `ENODATA` will be raised.

This function can support specifying a file descriptor and not following symlinks.

**Note:** A bug in Linux kernel versions less than 2.6.39 caused the flags argument to be ignored on some filesystems.

```python
os.XATTR_SIZE_MAX
```

The maximum size the value of an extended attribute can be. Currently, this is 64 KiB on Linux.

```python
os.XATTR_CREATE
```

This is a possible value for the flags argument in `setxattr()`. It indicates the operation must create an attribute.

```python
os.XATTR_REPLACE
```

This is a possible value for the flags argument in `setxattr()`. It indicates the operation must replace an existing attribute.

### 16.1.6 Process Management

These functions may be used to create and manage processes.

The various `exec*` functions take a list of arguments for the new program loaded into the process. In each case, the first of these arguments is passed to the new program as its own name rather than as an argument a user may have typed on a command line. For the C programmer, this is the `argv[0]` passed to a program’s `main()`. For example, `os.execv('/bin/echo', ['foo', 'bar'])` will only print `bar` on standard output; `foo` will seem to be ignored.

```python
os.abort()
```

Generate a SIGABRT signal to the current process. On Unix, the default behavior is to produce a core dump; on Windows, the process immediately returns an exit code of 3. Be aware that calling this function will not call the Python signal handler registered for SIGABRT with `signal.signal()`.

**Availability:** Unix, Windows.

```python
os.execl(path, arg0, arg1, ...)
os.execl(path, arg0, arg1, ..., env)
os.execlp(file, arg0, arg1, ...)
os.execlipse(file, arg0, arg1, ..., env)
os.execv(path, args)
os.execve(path, args, env)
os.execvp(file, args)
os.execvpe(file, args, env)
```

These functions all execute a new program, replacing the current process; they do not return. On Unix, the new executable is loaded into the current process, and will have the same process id as the caller. Errors will be reported as `OSError` exceptions.
The current process is replaced immediately. Open file objects and descriptors are not flushed, so if there may be data buffered on these open files, you should flush them using `sys.stdout.flush()` or `os.fsync()` before calling an `exec*` function.

The “l” and “v” variants of the `exec*` functions differ in how command-line arguments are passed. The “l” variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the `execl*()` functions. The “v” variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the `args` parameter. In either case, the arguments to the child process should start with the name of the command being run, but this is not enforced.

The variants which include a “p” near the end (`execlp()`, `execlpe()`, `execvp()`, and `execvpe()`) will use the `PATH` environment variable to locate the program file. When the environment is being replaced (using one of the `exec*e` variants, discussed in the next paragraph), the new environment is used as the source of the `PATH` variable. The other variants, `execl()`, `execl()` (note that these all end in “cl”), `execv()`, and `execve()`, will not use the `PATH` variable to locate the executable; `path` must contain an appropriate absolute or relative path.

For `execl()`, `execlp()`, `execvp()`, and `execvpe()` (note that these all end in “e”), the `env` parameter must be a mapping which is used to define the environment variables for the new process (these are used instead of the current process’ environment); the functions `execl()`, `execlp()`, `execv()`, and `execve()` all cause the new process to inherit the environment of the current process.

For `execve()` on some platforms, `path` may also be specified as an open file descriptor. This functionality may not be supported on your platform; you can check whether or not it is available using `os.supports_fd`. If it is unavailable, using it will raise a `NotImplementedError`.


`os._exit(n)`
Exit the process with status `n`, without calling cleanup handlers, flushing stdio buffers, etc.
Availability: Unix, Windows.

**Note:** The standard way to exit is `sys.exit(n)`. `_exit()` should normally only be used in the child process after a `fork()`.

The following exit codes are defined and can be used with `_exit()`, although they are not required. These are typically used for system programs written in Python, such as a mail server’s external command delivery program.

**Note:** Some of these may not be available on all Unix platforms, since there is some variation. These constants are defined where they are defined by the underlying platform.

`os.EX_OK`
Exit code that means no error occurred.
Availability: Unix.

`os.EX_USAGE`
Exit code that means the command was used incorrectly, such as when the wrong number of arguments are given.
Availability: Unix.

`os.EX_DATAERR`
Exit code that means the input data was incorrect.
Availability: Unix.

`os.EX_NOINPUT`
Exit code that means an input file did not exist or was not readable.
Availability: Unix.

**os.EX_NOUSER**
Exit code that means a specified user did not exist.
Availability: Unix.

**os.EX_NOHOST**
Exit code that means a specified host did not exist.
Availability: Unix.

**os.EX_UNAVAILABLE**
Exit code that means that a required service is unavailable.
Availability: Unix.

**os.EX_SOFTWARE**
Exit code that means an internal software error was detected.
Availability: Unix.

**os.EX_OSERR**
Exit code that means an operating system error was detected, such as the inability to fork or create a pipe.
Availability: Unix.

**os.EX_OSFIL**
Exit code that means some system file did not exist, could not be opened, or had some other kind of error.
Availability: Unix.

**os.EX_CANTCREAT**
Exit code that means a user specified output file could not be created.
Availability: Unix.

**os.EX_IOERR**
Exit code that means that an error occurred while doing I/O on some file.
Availability: Unix.

**os.EX_TEMPFAIL**
Exit code that means a temporary failure occurred. This indicates something that may not really be an error, such as a network connection that couldn’t be made during a retryable operation.
Availability: Unix.

**os.EX_PROTOCOL**
Exit code that means that a protocol exchange was illegal, invalid, or not understood.
Availability: Unix.

**os.EX_NOPERM**
Exit code that means that there were insufficient permissions to perform the operation (but not intended for file system problems).
Availability: Unix.

**os.EX_CONFIG**
Exit code that means that some kind of configuration error occurred.
Availability: Unix.

**os.EX_NOTFOUND**
Exit code that means something like “an entry was not found”.
Availability: Unix.
**os.fork()**

Fork a child process. Return 0 in the child and the child’s process id in the parent. If an error occurs OSError is raised.

Note that some platforms including FreeBSD <= 6.3, Cygwin and OS/2 EMX have known issues when using fork() from a thread.

Availability: Unix.

**os.forkpty()**

Fork a child process, using a new pseudo-terminal as the child’s controlling terminal. Return a pair of (pid, fd), where pid is 0 in the child, the new child’s process id in the parent, and fd is the file descriptor of the master end of the pseudo-terminal. For a more portable approach, use the pty module. If an error occurs OSError is raised.

Availability: some flavors of Unix.

**os.kill(pid, sig)**

Send signal sig to the process pid. Constants for the specific signals available on the host platform are defined in the signal module.

Windows: The `signal.CTRL_C_EVENT` and `signal.CTRL_BREAK_EVENT` signals are special signals which can only be sent to console processes which share a common console window, e.g., some subprocesses. Any other value for sig will cause the process to be unconditionally killed by the TerminateProcess API, and the exit code will be set to sig. The Windows version of `kill()` additionally takes process handles to be killed.

See also `signal.pthread_kill()`. New in version 3.2: Windows support.

**os.killpg(pgid, sig)**

Send the signal sig to the process group pgid.

Availability: Unix.

**os.nice(increment)**

Add increment to the process’s “niceness”. Return the new niceness.

Availability: Unix.

**os.plock(op)**

Lock program segments into memory. The value of op (defined in `<sys/lock.h>`) determines which segments are locked.

Availability: Unix.

**os.popen(...)**

Run child processes, returning opened pipes for communications. These functions are described in section File Object Creation.

**os.spawnl(mode, path, ...)**

**os.spawnle(mode, path, ..., env)**

**os.spawnlp(mode, file, ...)**

**os.spawnlpe(mode, file, ..., env)**

**os.spawnv(mode, path, args)**

**os.spawnve(mode, path, args, env)**

**os.spawnvp(mode, file, args)**

**os.spawnvpe(mode, file, args, env)**

Execute the program path in a new process.

(Note that the subprocess module provides more powerful facilities for spawning new processes and retrieving their results; using that module is preferable to using these functions. Check especially the Replacing Older Functions with the subprocess Module section.)

If mode is P_NOWAIT, this function returns the process id of the new process; if mode is P_WAIT, returns the process’s exit code if it exits normally, or -signal, where signal is the signal that killed the process. On Windows, the process id will actually be the process handle, so can be used with the `waitpid()` function.
The “l” and “v” variants of the spawn* functions differ in how command-line arguments are passed. The “l” variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the spawnl* functions. The “v” variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the args parameter. In either case, the arguments to the child process must start with the name of the command being run.

The variants which include a second “p” near the end (spawnlp(), spawnlpe(), spawnvp(), and spawnvpe()) will use the PATH environment variable to locate the program file. When the environment is being replaced (using one of the spawn*e variants, discussed in the next paragraph), the new environment is used as the source of the PATH variable. The other variants, spawnl(), spawnle(), spawnv(), and spawnve(), will not use the PATH variable to locate the executable; path must contain an appropriate absolute or relative path.

For spawnle(), spawnlpe(), spawnve(), and spawnve() (note that these all end in “e”), the env parameter must be a mapping which is used to define the environment variables for the new process (they are used instead of the current process’ environment); the functions spawnl(), spawnlpe(), spawnvp(), and spawnvpe() all cause the new process to inherit the environment of the current process. Note that keys and values in the env dictionary must be strings; invalid keys or values will cause the function to fail, with a return value of 127.

As an example, the following calls to spawnlp() and spawnvpe() are equivalent:

```python
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')
```
```python
L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

Availability: Unix, Windows. spawnlp(), spawnlpe(), spawnvp() and spawnvpe() are not available on Windows. spawnle() and spawnve() are not thread-safe on Windows; we advise you to use the subprocess module instead.

### os.P_NOWAIT
### os.P_NOWAITO
Possible values for the mode parameter to the spawn* family of functions. If either of these values is given, the spawn*() functions will return as soon as the new process has been created, with the process id as the return value.

Availability: Unix, Windows.

### os.P_WAIT
Possible value for the mode parameter to the spawn* family of functions. If this is given as mode, the spawn*() functions will not return until the new process has run to completion and will return the exit code of the process the run is successful, or -signal if a signal kills the process.

Availability: Unix, Windows.

### os.P_DETACH
### os.P_OVERLAY
Possible values for the mode parameter to the spawn* family of functions. These are less portable than those listed above. P_DETACH is similar to P_NOWAIT, but the new process is detached from the console of the calling process. If P_OVERLAY is used, the current process will be replaced; the spawn* function will not return.

Availability: Windows.

### os.startfile(path[, operation ])
Start a file with its associated application.
When *operation* is not specified or ‘open’, this acts like double-clicking the file in Windows Explorer, or giving the file name as an argument to the *start* command from the interactive command shell: the file is opened with whatever application (if any) its extension is associated.

When another *operation* is given, it must be a “command verb” that specifies what should be done with the file. Common verbs documented by Microsoft are ‘print’ and ‘edit’ (to be used on files) as well as ‘explore’ and ‘find’ (to be used on directories).  

*startfile()* returns as soon as the associated application is launched. There is no option to wait for the application to close, and no way to retrieve the application’s exit status. The *path* parameter is relative to the current directory. If you want to use an absolute path, make sure the first character is not a slash (’/’); the underlying Win32 *ShellExecute()* function doesn’t work if it is. Use the *os.path.normpath()* function to ensure that the path is properly encoded for Win32.

Availability: Windows.

*os.*<br>`system (command)`<br>Execute the command (a string) in a subshell. This is implemented by calling the Standard C function *system()*, and has the same limitations. Changes to *sys.stdin*, etc. are not reflected in the environment of the executed command. If *command* generates any output, it will be sent to the interpreter standard output stream.

On Unix, the return value is the exit status of the process encoded in the format specified for *wait()* . Note that POSIX does not specify the meaning of the return value of the C *system()* function, so the return value of the Python function is system-dependent.

On Windows, the return value is that returned by the system shell after running *command*. The shell is given by the Windows environment variable

COMSPEC: it is usually *cmd.exe*, which returns the exit status of the command run; on systems using a non-native shell, consult your shell documentation.

The subprocess module provides more powerful facilities for spawning new processes and retrieving their results; using that module is preferable to using this function. See the Replacing Older Functions with the subprocess Module section in the subprocess documentation for some helpful recipes.

Availability: Unix, Windows.

*os.*<br>`times()`<br>Returns the current global process times. The return value is an object with five attributes:

•*user* - user time

•*system* - system time

•*children_user* - user time of all child processes

•*children_system* - system time of all child processes

•*elapsed* - elapsed real time since a fixed point in the past

For backwards compatibility, this object also behaves like a five-tuple containing *user, system, children_user, children_system, and elapsed* in that order.

See the Unix manual page times(2) or the corresponding Windows Platform API documentation. On Windows, only *user* and *system* are known; the other attributes are zero. On OS/2, only *elapsed* is known; the other attributes are zero.

Availability: Unix, Windows. Changed in version 3.3: Return type changed from a tuple to a tuple-like object with named attributes.

*os.*<br>`wait()`<br>Wait for completion of a child process, and return a tuple containing its pid and exit status indication: a 16-bit number, whose low byte is the signal number that killed the process, and whose high byte is the exit status (if the signal number is zero); the high bit of the low byte is set if a core file was produced.

Availability: Unix.
os.waitid(idtype, id, options)

Wait for the completion of one or more child processes. idtype can be P_PID, P_PGID or P_ALL. id specifies the pid to wait on. options is constructed from the ORing of one or more of WEXITED, WSTOPPED or WCONTINUED and additionally may be ORed with WNOHANG or WNOWAIT. The return value is an object representing the data contained in the siginfo_t structure, namely: si_pid, si_uid, si_signo, si_status, si_code or None if WNOHANG is specified and there are no children in a waitable state.

Availability: Unix. New in version 3.3.

os.P_PID
os.P_PGID
os.P_ALL

These are the possible values for idtype in waitid(). They affect how id is interpreted.

Availability: Unix. New in version 3.3.

os.WEXITED
os.WSTOPPED
os.WNOWAIT

Flags that can be used in options in waitid() that specify what child signal to wait for.

Availability: Unix. New in version 3.3.

os.CLD_EXITED
os.CLD_DUMPED
os.CLD_TRAPPED
os.CLD_CONTINUED

These are the possible values for si_code in the result returned by waitid().

Availability: Unix. New in version 3.3.

os.waitpid(pid, options)

The details of this function differ on Unix and Windows.

On Unix: Wait for completion of a child process given by process id pid, and return a tuple containing its process id and exit status indication (encoded as for wait()). The semantics of the call are affected by the value of the integer options, which should be 0 for normal operation.

If pid is greater than 0, waitpid() requests status information for that specific process. If pid is 0, the request is for the status of any child in the process group of the current process. If pid is -1, the request pertains to any child of the current process. If pid is less than -1, status is requested for any process in the process group -pid (the absolute value of pid).

An OSError is raised with the value of errno when the syscall returns -1.

On Windows: Wait for completion of a process given by process handle pid, and return a tuple containing pid, and its exit status shifted left by 8 bits (shifting makes cross-platform use of the function easier). A pid less than or equal to 0 has no special meaning on Windows, and raises an exception. The value of integer options has no effect. pid can refer to any process whose id is known, not necessarily a child process. The spawn* functions called with P_NOWAIT return suitable process handles.

os.wait3(options)

Similar to waitpid(), except no process id argument is given and a 3-element tuple containing the child’s process id, exit status indication, and resource usage information is returned. Refer to resource.getrusage() for details on resource usage information. The option argument is the same as that provided to waitpid() and wait4().

Availability: Unix.

os.wait4(pid, options)

Similar to waitpid(), except a 3-element tuple, containing the child’s process id, exit status indication, and resource usage information is returned. Refer to resource.getrusage() for details on resource usage information. The arguments to wait4() are the same as those provided to waitpid().

Availability: Unix.
The option for `waitpid()` to return immediately if no child process status is available immediately. The function returns `(0, 0)` in this case.

Availability: Unix.

This option causes child processes to be reported if they have been continued from a job control stop since their status was last reported.

Availability: some Unix systems.

This option causes child processes to be reported if they have been stopped but their current state has not been reported since they were stopped.

Availability: Unix.

The following functions take a process status code as returned by `system()`, `wait()`, or `waitpid()` as a parameter. They may be used to determine the disposition of a process.

- `os.WCOREDUMP(status)`
  - Return `True` if a core dump was generated for the process, otherwise return `False`.
  - Availability: Unix.

- `os.WIFCONTINUED(status)`
  - Return `True` if the process has been continued from a job control stop, otherwise return `False`.
  - Availability: Unix.

- `os.WIFSTOPPED(status)`
  - Return `True` if the process has been stopped, otherwise return `False`.
  - Availability: Unix.

- `os.WIFSIGNALED(status)`
  - Return `True` if the process exited due to a signal, otherwise return `False`.
  - Availability: Unix.

- `os.WIFEXITED(status)`
  - Return `True` if the process exited using the `exit(2)` system call, otherwise return `False`.
  - Availability: Unix.

- `os.WEXITSTATUS(status)`
  - If `WIFEXITED(status)` is true, return the integer parameter to the `exit(2)` system call. Otherwise, the return value is meaningless.
  - Availability: Unix.

- `os.WSTOPSIG(status)`
  - Return the signal which caused the process to stop.
  - Availability: Unix.

- `os.WTERMSIG(status)`
  - Return the signal which caused the process to exit.
  - Availability: Unix.

### 16.1.7 Interface to the scheduler

These functions control how a process is allocated CPU time by the operating system. They are only available on some Unix platforms. For more detailed information, consult your Unix manpages. New in version 3.3. The following scheduling policies are exposed if they are supported by the operating system.
The default scheduling policy.

Scheduling policy for CPU-intensive processes that tries to preserve interactivity on the rest of the computer.

Scheduling policy for extremely low priority background tasks.

Scheduling policy for sporadic server programs.

A First In First Out scheduling policy.

A round-robin scheduling policy.

This flag can OR’ed with any other scheduling policy. When a process with this flag set forks, its child’s scheduling policy and priority are reset to the default.

class os.sched_param(sched_priority)

This class represents tunable scheduling parameters used in sched_setparam(), sched_setscheduler(), and sched_getparam(). It is immutable.

At the moment, there is only one possible parameter:

sched_priority

The scheduling priority for a scheduling policy.

Get the minimum priority value for policy. policy is one of the scheduling policy constants above.

Get the maximum priority value for policy. policy is one of the scheduling policy constants above.

Set the scheduling policy for the process with PID pid. A pid of 0 means the calling process. policy is one of the scheduling policy constants above. param is a sched_param instance.

Return the scheduling policy for the process with PID pid. A pid of 0 means the calling process. The result is one of the scheduling policy constants above.

Set a scheduling parameters for the process with PID pid. A pid of 0 means the calling process. param is a sched_param instance.

Return the scheduling parameters as a sched_param instance for the process with PID pid. A pid of 0 means the calling process.

Return the round-robin quantum in seconds for the process with PID pid. A pid of 0 means the calling process.

Voluntarily relinquish the CPU.

Restrict the process with PID pid (or the current process if zero) to a set of CPUs. mask is an iterable of integers representing the set of CPUs to which the process should be restricted.

Return the set of CPUs the process with PID pid (or the current process if zero) is restricted to.

See Also:
multiprocessing.cpu_count() returns the number of CPUs in the system.

16.1.8 Miscellaneous System Information

os.confstr(name)
Return string-valued system configuration values. name specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given as the keys of the confstr_names dictionary. For configuration variables not included in that mapping, passing an integer for name is also accepted.

If the configuration value specified by name isn’t defined, None is returned.

If name is a string and is not known, ValueError is raised. If a specific value for name is not supported by the host system, even if it is included in confstr_names, an OSError is raised with errno.EINVAL for the error number.

Availability: Unix.

os.confstr_names
Dictionary mapping names accepted by confstr() to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

Availability: Unix.

os.getloadavg()
Return the number of processes in the system run queue averaged over the last 1, 5, and 15 minutes or raises OSError if the load average was unobtainable.

Availability: Unix.

os.sysconf(name)
Return integer-valued system configuration values. If the configuration value specified by name isn’t defined, -1 is returned. The comments regarding the name parameter for confstr() apply here as well; the dictionary that provides information on the known names is given by sysconf_names.

Availability: Unix.

os.sysconf_names
Dictionary mapping names accepted by sysconf() to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

Availability: Unix.

The following data values are used to support path manipulation operations. These are defined for all platforms.

Higher-level operations on pathnames are defined in the os.path module.

os.curdir
The constant string used by the operating system to refer to the current directory. This is ’.’ for Windows and POSIX. Also available via os.path.

os.pardir
The constant string used by the operating system to refer to the parent directory. This is ’..’ for Windows and POSIX. Also available via os.path.

os.sep
The character used by the operating system to separate pathname components. This is ’/’ for POSIX and ’\’ for Windows. Note that knowing this is not sufficient to be able to parse or concatenate pathnames — use os.path.split() and os.path.join() — but it is occasionally useful. Also available via os.path.

os.altsep
An alternative character used by the operating system to separate pathname components, or None if only one separator character exists. This is set to ’/’ on Windows systems where sep is a backslash. Also available via os.path.
The character which separates the base filename from the extension; for example, the ‘.’ in os.py. Also available via os.path.

The character conventionally used by the operating system to separate search path components (as in PATH), such as ‘:’ for POSIX or ‘;’ for Windows. Also available via os.path.

The default search path used by exec*p* and spawn*p* if the environment doesn’t have a ‘PATH’ key. Also available via os.path.

The string used to separate (or, rather, terminate) lines on the current platform. This may be a single character, such as ‘\n’ for POSIX, or multiple characters, for example, ‘\r\n’ for Windows. Do not use os.linesep as a line terminator when writing files opened in text mode (the default); use a single ‘\n’ instead, on all platforms.

The file path of the null device. For example: ‘/dev/null’ for POSIX, ‘nul’ for Windows. Also available via os.path.

16.1.9 Miscellaneous Functions

Return a string of \( n \) random bytes suitable for cryptographic use.

This function returns random bytes from an OS-specific randomness source. The returned data should be unpredictable enough for cryptographic applications, though its exact quality depends on the OS implementation. On a Unix-like system this will query /dev/urandom, and on Windows it will use CryptGenRandom(). If a randomness source is not found, NotImplementedError will be raised.

For an easy-to-use interface to the random number generator provided by your platform, please see random.SystemRandom.

16.2 io — Core tools for working with streams

16.2.1 Overview

The io module provides Python’s main facilities for dealing with various types of I/O. There are three main types of I/O: text I/O, binary I/O and raw I/O. These are generic categories, and various backing stores can be used for each of them. A concrete object belonging to any of these categories is called a file object. Other common terms are stream and file-like object.

Independently of its category, each concrete stream object will also have various capabilities: it can be read-only, write-only, or read-write. It can also allow arbitrary random access (seeking forwards or backwards to any location), or only sequential access (for example in the case of a socket or pipe).

All streams are careful about the type of data you give to them. For example giving a str object to the write() method of a binary stream will raise a TypeError. So will giving a bytes object to the write() method of a text stream. Changed in version 3.3: Operations that used to raise IOError now raise OSError, since IOError is now an alias of OSError.

Text I/O

Text I/O expects and produces str objects. This means that whenever the backing store is natively made of bytes (such as in the case of a file), encoding and decoding of data is made transparently as well as optional translation of platform-specific newline characters.
The easiest way to create a text stream is with `open()`, optionally specifying an encoding:

```python
f = open("myfile.txt", "r", encoding="utf-8")
```

In-memory text streams are also available as `StringIO` objects:

```python
f = io.StringIO("some initial text data")
```

The text stream API is described in detail in the documentation of `TextIOBase`.

### Binary I/O

Binary I/O (also called *buffered I/O*) expects and produces `bytes` objects. No encoding, decoding, or newline translation is performed. This category of streams can be used for all kinds of non-text data, and also when manual control over the handling of text data is desired.

The easiest way to create a binary stream is with `open()` with `'b'` in the mode string:

```python
f = open("myfile.jpg", "rb")
```

In-memory binary streams are also available as `BytesIO` objects:

```python
f = io.BytesIO(b"some initial binary data: \x00\x01")
```

The binary stream API is described in detail in the docs of `BufferedIOBase`.

Other library modules may provide additional ways to create text or binary streams. See `socket.socket.makefile()` for example.

### Raw I/O

Raw I/O (also called *unbuffered I/O*) is generally used as a low-level building-block for binary and text streams; it is rarely useful to directly manipulate a raw stream from user code. Nevertheless, you can create a raw stream by opening a file in binary mode with buffering disabled:

```python
f = open("myfile.jpg", "rb", buffering=0)
```

The raw stream API is described in detail in the docs of `RawIOBase`.

### 16.2.2 High-level Module Interface

**`io.DEFAULT_BUFFER_SIZE`**

An int containing the default buffer size used by the module’s buffered I/O classes. `open()` uses the file’s `blksize` (as obtained by `os.stat()`) if possible.

```python
io.open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)
```

This is an alias for the builtin `open()` function.

**Exception `io.BlockingIOError`**

This is a compatibility alias for the builtin `BlockingIOError` exception.

**Exception `io.UnsupportedOperation`**

An exception inheriting `OSError` and `ValueError` that is raised when an unsupported operation is called on a stream.

### In-memory streams

It is also possible to use a `str` or `bytes`-like object as a file for both reading and writing. For strings `StringIO` can be used like a file opened in text mode. `BytesIO` can be used like a file opened in binary mode. Both provide full read-write capabilities with random access.

See Also:
**sys** contains the standard IO streams: **sys.stdin**, **sys.stdout**, and **sys.stderr**.

### 16.2.3 Class hierarchy

The implementation of I/O streams is organized as a hierarchy of classes. First abstract base classes (ABCs), which are used to specify the various categories of streams, then concrete classes providing the standard stream implementations.

**Note:** The abstract base classes also provide default implementations of some methods in order to help implementation of concrete stream classes. For example, **BufferedIOBase** provides unoptimized implementations of **readinto()** and **readline()**.

At the top of the I/O hierarchy is the abstract base class **IOBase**. It defines the basic interface to a stream. Note, however, that there is no separation between reading and writing to streams; implementations are allowed to raise **UnsupportedOperation** if they do not support a given operation.

The **RawIOBase** ABC extends **IOBase**. It deals with the reading and writing of bytes to a stream. **FileIO** subclasses **RawIOBase** to provide an interface to files in the machine’s file system.

The **BufferedIOBase** ABC deals with buffering on a raw byte stream (**RawIOBase**). Its subclasses, **BufferedReader**, **BufferedReader**, and **BufferedReader** buffer streams that are readable, writable, and both readable and writable. **BufferedReader** provides a buffered interface to random access streams. Another **BufferedIOBase** subclass, **BytesIO**, is a stream of in-memory bytes.

The **TextIOBase** ABC, another subclass of **IOBase**, deals with streams whose bytes represent text, and handles encoding and decoding to and from strings. **TextIOWrapper**, which extends it, is a buffered text interface to a buffered raw stream (**BufferedIOBase**). Finally, **StringIO** is an in-memory stream for text.

Argument names are not part of the specification, and only the arguments of **open()** are intended to be used as keyword arguments.

The following table summarizes the ABCs provided by the **io** module:

<table>
<thead>
<tr>
<th>ABC</th>
<th>Inherits</th>
<th>Stub Methods</th>
<th>Mixin Methods and Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOBase</td>
<td></td>
<td>fileno, seek, and truncate</td>
<td>close, closed, <strong>enter</strong>, <strong>exit</strong>, flush, isatty, <strong>iter</strong>, <strong>next</strong>, readable, readline, readlines, seekable, tell, writable, and writelines</td>
</tr>
<tr>
<td>RawIOBase</td>
<td>IOBase</td>
<td>readinto and write</td>
<td>Inherited <strong>IOBase</strong> methods, read, and readall</td>
</tr>
<tr>
<td>BufferedIOBase</td>
<td>IOBase</td>
<td>detach, read, readI, and write</td>
<td>Inherited <strong>IOBase</strong> methods, readinto</td>
</tr>
<tr>
<td>TextIOBase</td>
<td>IOBase</td>
<td>detach, read, readline, and write</td>
<td>Inherited <strong>IOBase</strong> methods, encoding, errors, and newlines</td>
</tr>
</tbody>
</table>

### I/O Base Classes

**class** **io.IOBase**

The abstract base class for all I/O classes, acting on streams of bytes. There is no public constructor.

This class provides empty abstract implementations for many methods that derived classes can override selectively; the default implementations represent a file that cannot be read, written or sought.
Even though `IOBase` does not declare `read()`, `readinto()`, or `write()` because their signatures will vary, implementations and clients should consider those methods part of the interface. Also, implementations may raise a `ValueError` (or `UnsupportedOperation`) when operations they do not support are called.

The basic type used for binary data read from or written to a file is `bytes`. `bytearrays` are accepted too, and in some cases (such as `readinto()`) required. Text I/O classes work with `str` data.

Note that calling any method (even inquiries) on a closed stream is undefined. Implementations may raise `ValueError` in this case.

`IOBase` (and its subclasses) supports the iterator protocol, meaning that an `IOBase` object can be iterated over yielding the lines in a stream. Lines are defined slightly differently depending on whether the stream is a binary stream (yielding bytes), or a text stream (yielding character strings). See `readline()` below.

`IOBase` is also a context manager and therefore supports the `with` statement. In this example, `file` is closed after the `with` statement’s suite is finished—even if an exception occurs:

```python
with open('spam.txt', 'w') as file:
    file.write('Spam and eggs!')
```

`IOBase` provides these data attributes and methods:

- `close()`
  - Flush and close this stream. This method has no effect if the file is already closed. Once the file is closed, any operation on the file (e.g. reading or writing) will raise a `ValueError`.
  - As a convenience, it is allowed to call this method more than once; only the first call, however, will have an effect.

- `closed`
  - True if the stream is closed.

- `fileno()`
  - Return the underlying file descriptor (an integer) of the stream if it exists. An `OSError` is raised if the IO object does not use a file descriptor.

- `flush()`
  - Flush the write buffers of the stream if applicable. This does nothing for read-only and non-blocking streams.

- `isatty()`
  - Return `True` if the stream is interactive (i.e., connected to a terminal/tty device).

- `readable()`
  - Return `True` if the stream can be read from. If False, `read()` will raise `OSError`.

- `readline(limit=-1)`
  - Read and return one line from the stream. If `limit` is specified, at most `limit` bytes will be read.
  - The line terminator is always `b'\n'` for binary files; for text files, the `newlines` argument to `open()` can be used to select the line terminator(s) recognized.

- `readlines(hint=-1)`
  - Read and return a list of lines from the stream. `hint` can be specified to control the number of lines read: no more lines will be read if the total size (in bytes/characters) of all lines so far exceeds `hint`.
  - Note that it’s already possible to iterate on file objects using `for line in file: ...` without calling `file.readlines()`.

- `seek(offset, whence=SEEK_SET)`
  - Change the stream position to the given byte `offset`. `offset` is interpreted relative to the position indicated by `whence`. Values for `whence` are:
    - `SEEK_SET` or 0 – start of the stream (the default); `offset` should be zero or positive
    - `SEEK_CUR` or 1 – current stream position; `offset` may be negative
SEEK_END or 2 – end of the stream; offset is usually negative

Return the new absolute position. New in version 3.1: The SEEK_* constants. New in version 3.3: Some operating systems could support additional values, like os.SEEK_HOLE or os.SEEK_DATA. The valid values for a file could depend on it being open in text or binary mode.

seekable()
Return True if the stream supports random access. If False, seek(), tell() and truncate() will raise OSError.

tell()
Return the current stream position.

truncate(size=None)
Resize the stream to the given size in bytes (or the current position if size is not specified). The current stream position isn’t changed. This resizing can extend or reduce the current file size. In case of extension, the contents of the new file area depend on the platform (on most systems, additional bytes are zero-filled, on Windows they’re undetermined). The new file size is returned.

writable()
Return True if the stream supports writing. If False, write() and truncate() will raise OSError.

writelines(lines)
Write a list of lines to the stream. Line separators are not added, so it is usual for each of the lines provided to have a line separator at the end.

class io.RawIOBase
Base class for raw binary I/O. It inherits IOBase. There is no public constructor.

Raw binary I/O typically provides low-level access to an underlying OS device or API, and does not try to encapsulate it in high-level primitives (this is left to Buffered I/O and Text I/O, described later in this page).

In addition to the attributes and methods from IOBase, RawIOBase provides the following methods:

read(n=-1)
Read up to n bytes from the object and return them. As a convenience, if n is unspecified or -1, readall() is called. Otherwise, only one system call is ever made. Fewer than n bytes may be returned if the operating system call returns fewer than n bytes.

If 0 bytes are returned, and n was not 0, this indicates end of file. If the object is in non-blocking mode and no bytes are available, None is returned.

readall()
Read and return all the bytes from the stream until EOF, using multiple calls to the stream if necessary.

readinto(b)
Read up to len(b) bytes into bytearray b and return the number of bytes read. If the object is in non-blocking mode and no bytes are available, None is returned.

write(b)
Write the given bytes or bytearray object, b, to the underlying raw stream and return the number of bytes written. This can be less than len(b), depending on specifics of the underlying raw stream, and especially if it is in non-blocking mode. None is returned if the raw stream is set not to block and no single byte could be readily written to it.

class io.BufferedIOBase
Base class for binary streams that support some kind of buffering. It inherits IOBase. There is no public constructor.

The main difference with RawIOBase is that methods read(), readinto() and write() will try (respectively) to read as much input as requested or to consume all given output, at the expense of making perhaps more than one system call.

In addition, those methods can raise BlockingIOError if the underlying raw stream is in non-blocking mode and cannot take or give enough data; unlike their RawIOBase counterparts, they will never return None.
Besides, the `read()` method does not have a default implementation that defers to `readinto()`.

A typical `BufferedIOBase` implementation should not inherit from a `RawIOBase` implementation, but wrap one, like `BufferedWriter` and `BufferedReader` do.

`BufferedIOBase` provides or overrides these methods and attribute in addition to those from `IOBase`:

- **raw**
  The underlying raw stream (a `RawIOBase` instance) that `BufferedIOBase` deals with. This is not part of the `BufferedIOBase` API and may not exist on some implementations.

- **detach()**
  Separate the underlying raw stream from the buffer and return it.
  After the raw stream has been detached, the buffer is in an unusable state.
  Some buffers, like `BytesIO`, do not have the concept of a single raw stream to return from this method. They raise `UnsupportedOperationException`. New in version 3.1.

- **read(n=-1)**
  Read and return up to `n` bytes. If the argument is omitted, `None`, or negative, data is read and returned until EOF is reached. An empty `bytes` object is returned if the stream is already at EOF.
  If the argument is positive, and the underlying raw stream is not interactive, multiple raw reads may be issued to satisfy the byte count (unless EOF is reached first). But for interactive raw streams, at most one raw read will be issued, and a short result does not imply that EOF is imminent.
  A `BlockingIOError` is raised if the underlying raw stream is in non-blocking mode, and has no data available at the moment.

- **read1(n=-1)**
  Read and return up to `n` bytes, with at most one call to the underlying raw stream's `read()` method. This can be useful if you are implementing your own buffering on top of a `BufferedIOBase` object.

- **readinto(b)**
  Read up to `len(b)` bytes into `bytearray b` and return the number of bytes read.
  Like `read()`, multiple reads may be issued to the underlying raw stream, unless the latter is interactive.
  A `BlockingIOError` is raised if the underlying raw stream is in non-blocking mode, and has no data available at the moment.

- **write(b)**
  Write the given `bytes` or `bytearray` object, `b` and return the number of bytes written (never less than `len(b)`, since if the write fails an `OSError` will be raised). Depending on the actual implementation, these bytes may be readily written to the underlying stream, or held in a buffer for performance and latency reasons.
  When in non-blocking mode, a `BlockingIOError` is raised if the data needed to be written to the raw stream but it couldn’t accept all the data without blocking.

### Raw File I/O

**class** `io.FileIO(name, mode='r', closefd=True, opener=None)`

`FileIO` represents an OS-level file containing bytes data. It implements the `RawIOBase` interface (and therefore the `IOBase` interface, too).

The `name` can be one of two things:

- a character string or `bytes` object representing the path to the file which will be opened;
- an integer representing the number of an existing OS-level file descriptor to which the resulting `FileIO` object will give access.
The mode can be ‘r’, ‘w’, ‘x’ or ‘a’ for reading (default), writing, exclusive creation or appending. The file will be created if it doesn’t exist when opened for writing or appending; it will be truncated when opened for writing. FileExistsError will be raised if it already exists when opened for creating. Opening a file for creating implies writing, so this mode behaves in a similar way to ‘w’. Add a ‘+’ to the mode to allow simultaneous reading and writing.

The read() (when called with a positive argument), readinto() and write() methods on this class will only make one system call.

A custom opener can be used by passing a callable as opener. The underlying file descriptor for the file object is then obtained by calling opener with (name, flags). opener must return an open file descriptor (passing os.open as opener results in functionality similar to passing None).

See the open() built-in function for examples on using the opener parameter. Changed in version 3.3: The opener parameter was added. The ‘x’ mode was added. In addition to the attributes and methods from IOBase and RawIOBase, FileIO provides the following data attributes:

mode
   The mode as given in the constructor.

name
   The file name. This is the file descriptor of the file when no name is given in the constructor.

Buffered Streams

Buffered I/O streams provide a higher-level interface to an I/O device than raw I/O does.

class io.BytesIO([initial_bytes])
   A stream implementation using an in-memory bytes buffer. It inherits BufferedIOBase.

   The argument initial_bytes contains optional initial bytes data.

   BytesIO provides or overrides these methods in addition to those from BufferedIOBase and IOBase:

   getbuffer()
      Return a readable and writable view over the contents of the buffer without copying them. Also, mutating the view will transparently update the contents of the buffer:

      >>> b = io.BytesIO(b"abcdef")
      >>> view = b.getbuffer()
      >>> view[2:4] = b"56"
      >>> b.getvalue()
      b'ab56ef'

      Note: As long as the view exists, the BytesIO object cannot be resized.

      New in version 3.2.

   getvalue()
      Return bytes containing the entire contents of the buffer.

   read1()
      In BytesIO, this is the same as read().

class io.BufferedReader (raw, buffer_size=DEFAULT_BUFFER_SIZE)
   A buffer providing higher-level access to a readable, sequential RawIOBase object. It inherits BufferedIOBase. When reading data from this object, a larger amount of data may be requested from the underlying raw stream, and kept in an internal buffer. The buffered data can then be returned directly on subsequent reads.

   The constructor creates a BufferedReader for the given readable raw stream and buffer_size. If buffer_size is omitted, DEFAULT_BUFFER_SIZE is used.
BufferedReader provides or overrides these methods in addition to those from BufferedIOBase and IOBase:

peek([n])
Return bytes from the stream without advancing the position. At most one single read on the raw stream is done to satisfy the call. The number of bytes returned may be less or more than requested.

read([n])
Read and return n bytes, or if n is not given or negative, until EOF or if the read call would block in non-blocking mode.

read1(n)
Read and return up to n bytes with only one call on the raw stream. If at least one byte is buffered, only buffered bytes are returned. Otherwise, one raw stream read call is made.

class io.BufferedWriter (raw, buffer_size=DEFAULT_BUFFER_SIZE)
A buffer providing higher-level access to a writeable, sequential RawIOBase object. It inherits BufferedIOBase. When writing to this object, data is normally placed into an internal buffer. The buffer will be written out to the underlying RawIOBase object under various conditions, including:

• when the buffer gets too small for all pending data;
• when flush() is called;
• when a seek() is requested (for BufferedReader objects);
• when the BufferedWriter object is closed or destroyed.

The constructor creates a BufferedWriter for the given writeable raw stream. If the buffer_size is not given, it defaults to DEFAULT_BUFFER_SIZE.

BufferedWriter provides or overrides these methods in addition to those from BufferedIOBase and IOBase:

flush()
Force bytes held in the buffer into the raw stream. A BlockingIOError should be raised if the raw stream blocks.

write(b)
Write the bytes or bytearray object, b and return the number of bytes written. When in non-blocking mode, a BlockingIOError is raised if the buffer needs to be written out but the raw stream blocks.

class io.BufferedRandom (raw, buffer_size=DEFAULT_BUFFER_SIZE)
A buffered interface to random access streams. It inherits BufferedReader and BufferedWriter, and further supports seek() and tell() functionality.

The constructor creates a reader and writer for a seekable raw stream, given in the first argument. If the buffer_size is omitted it defaults to DEFAULT_BUFFER_SIZE.

BufferedRandom is capable of anything BufferedReader or BufferedWriter can do.

class io.BufferedRWPair (reader, writer, buffer_size=DEFAULT_BUFFER_SIZE)
A buffered I/O object combining two unidirectional RawIOBase objects – one readable, the other writeable – into a single bidirectional endpoint. It inherits BufferedIOBase.

reader and writer are RawIOBase objects that are readable and writeable respectively. If the buffer_size is omitted it defaults to DEFAULT_BUFFER_SIZE.

BufferedRWPair implements all of BufferedIOBase’s methods except for detach(), which raises UnsupportedOperation.

Warning: BufferedWriter does not attempt to synchronize accesses to its underlying raw streams. You should not pass it the same object as reader and writer; use BufferedReader instead.
Text I/O

class io.TextIOBase
Base class for text streams. This class provides a character and line based interface to stream I/O. There is no readinto() method because Python’s character strings are immutable. It inherits IOBase. There is no public constructor.

TextIOBase provides or overrides these data attributes and methods in addition to those from IOBase:

encoding
The name of the encoding used to decode the stream’s bytes into strings, and to encode strings into bytes.

errors
The error setting of the decoder or encoder.

newlines
A string, a tuple of strings, or None, indicating the newlines translated so far. Depending on the implementation and the initial constructor flags, this may not be available.

buffer
The underlying binary buffer (a BufferedIOBase instance) that TextIOBase deals with. This is not part of the TextIOBase API and may not exist in some implementations.

detach()
Separate the underlying binary buffer from the TextIOBase and return it.

After the underlying buffer has been detached, the TextIOBase is in an unusable state.

Some TextIOBase implementations, like StringIO, may not have the concept of an underlying buffer and calling this method will raise UnsupportedOperation. New in version 3.1.

read(n)
Read and return at most n characters from the stream as a single str. If n is negative or None, reads until EOF.

readline(limit=-1)
Read until newline or EOF and return a single str. If the stream is already at EOF, an empty string is returned.

If limit is specified, at most limit characters will be read.

seek(offset, whence=SEEK_SET)
Change the stream position to the given offset. Behaviour depends on the whence parameter:

• SEEK_SET or 0: seek from the start of the stream (the default); offset must either be a number returned by TextIOBase.tell(), or zero. Any other offset value produces undefined behaviour.

• SEEK_CUR or 1: “seek” to the current position; offset must be zero, which is a no-operation (all other values are unsupported).

• SEEK_END or 2: seek to the end of the stream; offset must be zero (all other values are unsupported).

Return the new absolute position as an opaque number. New in version 3.1: The SEEK_* constants.

tell()
Return the current stream position as an opaque number. The number does not usually represent a number of bytes in the underlying binary storage.

write(s)
Write the string s to the stream and return the number of characters written.

class io.TextIOWrapper(buffer, encoding=None, errors=None, newline=None, line_buffering=False, write_through=False)
A buffered text stream over a BufferedIOBase binary stream. It inherits TextIOBase.
encoding gives the name of the encoding that the stream will be decoded or encoded with. It defaults to `locale.getpreferredencoding(False)`.

errors is an optional string that specifies how encoding and decoding errors are to be handled. Pass ‘strict’ to raise a ValueError exception if there is an encoding error (the default of None has the same effect), or pass ‘ignore’ to ignore errors. (Note that ignoring encoding errors can lead to data loss.) ‘replace’ causes a replacement marker (such as ‘?’) to be inserted where there is malformed data. When writing, ‘xmlcharrefreplace’ (replace with the appropriate XML character reference) or ‘backslashreplace’ (replace with backslashed escape sequences) can be used. Any other error handling name that has been registered with codecs.register_error() is also valid.

newline controls how line endings are handled. It can be None,”, ‘\n’, ‘\r’, and ‘\r\n’. It works as follows:

•When reading input from the stream, if newline is None, universal newlines mode is enabled. Lines in the input can end in ‘\n’, ‘\r’, or ‘\r\n’, and these are translated into ‘\n’ before being returned to the caller. If it is ”, universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.

•When writing output to the stream, if newline is None, any ‘\n’ characters written are translated to the system default line separator, os.linesep. If newline is ” or ‘\n’, no translation takes place. If newline is any of the other legal values, any ‘\n’ characters written are translated to the given string.

If line_buffering is True, flush() is implied when a call to write contains a newline character.

If write_through is True, calls to write() are guaranteed not to be buffered: any data written on the TextIOWrapper object is immediately handled to its underlying binary buffer. Changed in version 3.3: The default encoding is now locale.getpreferredencoding(False) instead of locale.getpreferredencoding(). Don’t change temporary the locale encoding using locale.setlocale(), use the current locale encoding instead of the user preferred encoding. TextIOWrapper provides one attribute in addition to those of TextIOBase and its parents:

line_buffering
Whether line buffering is enabled.

class io.StringIO (initial_value='', newline=None)
An in-memory stream for text I/O.

The initial value of the buffer (an empty string by default) can be set by providing initial_value. The newline argument works like that of TextIOWrapper. The default is to do no newline translation.

StringIO provides this method in addition to those from TextIOBase and its parents:

getvalue()
Return a str containing the entire contents of the buffer at any time before the StringIO object’s close() method is called.

Example usage:

import io

output = io.StringIO()
output.write(‘First line.\n’)
print(‘Second line.’, file=output)

# Retrieve file contents -- this will be
# ‘First line.\nSecond line.\n’
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()

class io.IncrementalNewlineDecoder
A helper codec that decodes newlines for universal newlines mode. It inherits
codecs.IncrementalDecoder.

16.2.4 Performance

This section discusses the performance of the provided concrete I/O implementations.

Binary I/O

By reading and writing only large chunks of data even when the user asks for a single byte, buffered I/O hides
any inefficiency in calling and executing the operating system's unbuffered I/O routines. The gain depends on the
OS and the kind of I/O which is performed. For example, on some modern OSes such as Linux, unbuffered disk
I/O can be as fast as buffered I/O. The bottom line, however, is that buffered I/O offers predictable performance
regardless of the platform and the backing device. Therefore, it is almost always preferable to use buffered I/O
rather than unbuffered I/O for binary data.

Text I/O

Text I/O over a binary storage (such as a file) is significantly slower than binary I/O over the same storage,
because it requires conversions between unicode and binary data using a character codec. This can become
noticeable handling huge amounts of text data like large log files. Also, TextIOWrapper.tell() and
TextIOWrapper.seek() are both quite slow due to the reconstruction algorithm used.

StringIO, however, is a native in-memory unicode container and will exhibit similar speed to BytesIO.

Multi-threading

FileIO objects are thread-safe to the extent that the operating system calls (such as read(2) under Unix) they
wrap are thread-safe too.

Binary buffered objects (instances of BufferedReader, BufferedWriter, BufferedRandom and
BufferedReaderRWPair) protect their internal structures using a lock; it is therefore safe to call them from multi-
ple threads at once.

TextIOWrapper objects are not thread-safe.

Reentrancy

Binary buffered objects (instances of BufferedReader, BufferedWriter, BufferedRandom and
BufferedReaderRWPair) are not reentrant. While reentrant calls will not happen in normal situations, they can arise
from doing I/O in a signal handler. If a thread tries to re-enter a buffered object which it is already accessing, a
RuntimeError is raised. Note this doesn’t prohibit a different thread from entering the buffered object.

The above implicitly extends to text files, since the open() function will wrap a buffered object inside a
TextIOWrapper. This includes standard streams and therefore affects the built-in function print() as well.

16.3 time — Time access and conversions

This module provides various time-related functions. For related functionality, see also the datetime and
calendar modules.
Although this module is always available, not all functions are available on all platforms. Most of the functions defined in this module call platform C library functions with the same name. It may sometimes be helpful to consult the platform documentation, because the semantics of these functions varies among platforms.

An explanation of some terminology and conventions is in order.

- The epoch is the point where the time starts. On January 1st of that year, at 0 hours, the “time since the epoch” is zero. For Unix, the epoch is 1970. To find out what the epoch is, look at gmtime(0).

- The functions in this module may not handle dates and times before the epoch or far in the future. The cut-off point in the future is determined by the C library; for 32-bit systems, it is typically in 2038.

- Year 2000 (Y2K) issues: Python depends on the platform’s C library, which generally doesn’t have year 2000 issues, since all dates and times are represented internally as seconds since the epoch. Function strftime() can parse 2-digit years when given %Y format code. When 2-digit years are parsed, they are converted according to the POSIX and ISO C standards: values 69–99 are mapped to 1969–1999, and values 0–68 are mapped to 2000–2068.

- UTC is Coordinated Universal Time (formerly known as Greenwich Mean Time, or GMT). The acronym UTC is not a mistake but a compromise between English and French.

- DST is Daylight Saving Time, an adjustment of the timezone by (usually) one hour during part of the year. DST rules are magic (determined by local law) and can change from year to year. The C library has a table containing the local rules (often it is read from a system file for flexibility) and is the only source of True Wisdom in this respect.

- The precision of the various real-time functions may be less than suggested by the units in which their value or argument is expressed. E.g. on most Unix systems, the clock “ticks” only 50 or 100 times a second.

- On the other hand, the precision of time() and sleep() is better than their Unix equivalents: times are expressed as floating point numbers, time() returns the most accurate time available (using Unix gettimeofday() where available), and sleep() will accept a time with a nonzero fraction (Unix select() is used to implement this, where available).

- The time value as returned by gmtime(), localtime(), and strftime(), and accepted by asctime(), mktime() and strftime(), is a sequence of 9 integers. The return values of gmtime(), localtime(), and strftime() also offer attribute names for individual fields.

  See struct_time for a description of these objects. Changed in version 3.3: The struct_time type was extended to provide the tm_gmtoff and tm_zone attributes when platform supports corresponding struct tm members.

- Use the following functions to convert between time representations:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>seconds since the epoch</td>
<td>struct_time in UTC</td>
<td>gmtime()</td>
</tr>
<tr>
<td>seconds since the epoch</td>
<td>struct_time in local time</td>
<td>localtime()</td>
</tr>
<tr>
<td>struct_time in UTC</td>
<td>seconds since the epoch</td>
<td>calendar.timegm()</td>
</tr>
<tr>
<td>struct_time in local time</td>
<td>seconds since the epoch</td>
<td>mktime()</td>
</tr>
</tbody>
</table>

The module defines the following functions and data items:

time.altzone

The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if daylight is nonzero.

time.asctime([t])

Convert a tuple or struct_time representing a time as returned by gmtime() or localtime() to a string of the following form: ‘Sun Jun 20 23:21:05 1993’. If t is not provided, the current time as returned by localtime() is used. Locale information is not used by asctime().

Note: Unlike the C function of the same name, asctime() does not add a trailing newline.
time.clock()  
On Unix, return the current processor time as a floating point number expressed in seconds. The precision, 
and in fact the very definition of the meaning of "processor time", depends on that of the C function of the 
same name, but in any case, this is the function to use for benchmarking Python or timing algorithms. 

On Windows, this function returns wall-clock seconds elapsed since the first call to this function, as a 
floating point number, based on the Win32 function QueryPerformanceCounter(). The resolution 
is typically better than one microsecond. Deprecated since version 3.3: The behaviour of this function 
depends on the platform: use perf_counter() or process_time() instead, depending on your 
requirements, to have a well defined behaviour.

time.clock_getres(clk_id)  
Return the resolution (precision) of the specified clock clk_id. 
Availability: Unix. New in version 3.3.

time.clock_gettime(clk_id)  
Return the time of the specified clock clk_id. 
Availability: Unix. New in version 3.3.

time.clock_settime(clk_id, time)  
Set the time of the specified clock clk_id. 
Availability: Unix. New in version 3.3.

time.CLOCK_HIGHRES  
The Solaris OS has a CLOCK_HIGHRES timer that attempts to use an optimal hardware source, and may 
give close to nanosecond resolution. CLOCK_HIGHRES is the nonadjustable, high-resolution clock. 
Availability: Solaris. New in version 3.3.

time.CLOCK_MONOTONIC  
Clock that cannot be set and represents monotonic time since some unspecified starting point. 
Availability: Unix. New in version 3.3.

time.CLOCK_MONOTONIC_RAW  
Similar to CLOCK_MONOTONIC, but provides access to a raw hardware-based time that is not subject to 
NTP adjustments. 
Availability: Linux 2.6.28 or later. New in version 3.3.

time.CLOCK_PROCESS_CPUTIME_ID  
High-resolution per-process timer from the CPU. 
Availability: Unix. New in version 3.3.

time.CLOCK_REALTIME  
System-wide real-time clock. Setting this clock requires appropriate privileges. 
Availability: Unix. New in version 3.3.

time.CLOCK_THREAD_CPUTIME_ID  
Thread-specific CPU-time clock. 
Availability: Unix. New in version 3.3.

time.ctime([secs])  
Convert a time expressed in seconds since the epoch to a string representing local time. If secs is not 
provided or None, the current time as returned by time() is used. ctime(secs) is equivalent to 
astime(localtime(secs)). Locale information is not used by ctime().

time.daylight  
Nonzero if a DST timezone is defined.

time.get_clock_info(name)  
Get information on the specified clock as a namespace object. Supported clock names and the corresponding 
functions to read their value are:
• 'clock': time.clock()
• 'monotonic': time.monotonic()
• 'perf_counter': time.perf_counter()
• 'process_time': time.process_time()
• 'time': time.time()

The result has the following attributes:

• adjustable: True if the clock can be changed automatically (e.g. by a NTP daemon) or manually by
  the system administrator, False otherwise

• implementation: The name of the underlying C function used to get the clock value

• monotonic: True if the clock cannot go backward, False otherwise

• resolution: The resolution of the clock in seconds (float)

New in version 3.3.

time.gmtime([secs])
Convert a time expressed in seconds since the epoch to a struct_time in UTC in which the dst
flag is always zero. If secs is not provided or None, the current time as returned by time() is used.
Fractions of a second are ignored. See above for a description of the struct_time object. See
calendar.timegm() for the inverse of this function.

time.localtime([secs])
Like gmtime() but converts to local time. If secs is not provided or None, the current time as returned by
time() is used. The dst flag is set to 1 when DST applies to the given time.

time.mktime(t)
This is the inverse function of localtime(). Its argument is the struct_time or full 9-tuple (since the
dst flag is needed; use -1 as the dst flag if it is unknown) which expresses the time in local time, not UTC.
It returns a floating point number, for compatibility with time(). If the input value cannot be represented
as a valid time, either OverflowError or ValueError will be raised (which depends on whether the
invalid value is caught by Python or the underlying C libraries). The earliest date for which it can generate
a time is platform-dependent.

time.monotonic()
Return the value (in fractional seconds) of a monotonic clock, i.e. a clock that cannot go backwards. The
clock is not affected by system clock updates. The reference point of the returned value is undefined, so that
only the difference between the results of consecutive calls is valid.

On Windows versions older than Vista, monotonic() detects GetTickCount() integer overflow (32
bits, roll-over after 49.7 days). It increases an internal epoch (reference time) by \(2^{32}\) each time that an
overflow is detected. The epoch is stored in the process-local state and so the value of monotonic() may
be different in two Python processes running for more than 49 days. On more recent versions of Windows
and on other operating systems, monotonic() is system-wide.


time.perf_counter()
Return the value (in fractional seconds) of a performance counter, i.e. a clock with the highest available
resolution to measure a short duration. It does include time elapsed during sleep and is system-wide. The
reference point of the returned value is undefined, so that only the difference between the results of consec-
tutive calls is valid. New in version 3.3.

time.process_time()
Return the value (in fractional seconds) of the sum of the system and user CPU time of the current process.
It does not include time elapsed during sleep. It is process-wide by definition. The reference point of the
returned value is undefined, so that only the difference between the results of consecutive calls is valid. New
in version 3.3.

time.sleep(secs)
Suspend execution for the given number of seconds. The argument may be a floating point number to
indicate a more precise sleep time. The actual suspension time may be less than that requested because any caught signal will terminate the sleep() following execution of that signal’s catching routine. Also, the suspension time may be longer than requested by an arbitrary amount because of the scheduling of other activity in the system.

```
from time import strftime
```

Convert a tuple or struct_time representing a time as returned by gmtime() or localtime() to a string as specified by the format argument. If `t` is not provided, the current time as returned by localtime() is used. *format* must be a string. ValueError is raised if any field in `t` is outside of the allowed range.

0 is a legal argument for any position in the time tuple; if it is normally illegal the value is forced to a correct one.

The following argument directives can be embedded in the *format* string. They are shown without the optional field width and precision specification, and are replaced by the indicated characters in the strftime() result:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Locale’s abbreviated weekday name.</td>
<td></td>
</tr>
<tr>
<td>%A</td>
<td>Locale’s full weekday name.</td>
<td></td>
</tr>
<tr>
<td>%b</td>
<td>Locale’s abbreviated month name.</td>
<td></td>
</tr>
<tr>
<td>%B</td>
<td>Locale’s full month name.</td>
<td></td>
</tr>
<tr>
<td>%c</td>
<td>Locale’s appropriate date and time representation.</td>
<td></td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month as a decimal number [01,31].</td>
<td></td>
</tr>
<tr>
<td>%H</td>
<td>Hour (24-hour clock) as a decimal number [00,23].</td>
<td></td>
</tr>
<tr>
<td>%I</td>
<td>Hour (12-hour clock) as a decimal number [01,12].</td>
<td></td>
</tr>
<tr>
<td>%j</td>
<td>Day of the year as a decimal number [001,366].</td>
<td></td>
</tr>
<tr>
<td>%m</td>
<td>Month as a decimal number [01,12].</td>
<td></td>
</tr>
<tr>
<td>%M</td>
<td>Minute as a decimal number [00,59].</td>
<td></td>
</tr>
<tr>
<td>%p</td>
<td>Locale’s equivalent of either AM or PM.</td>
<td>(1)</td>
</tr>
<tr>
<td>%S</td>
<td>Second as a decimal number [00,61].</td>
<td>(2)</td>
</tr>
<tr>
<td>%U</td>
<td>Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.</td>
<td>(3)</td>
</tr>
<tr>
<td>%W</td>
<td>Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.</td>
<td>(3)</td>
</tr>
<tr>
<td>%x</td>
<td>Locale’s appropriate date representation.</td>
<td></td>
</tr>
<tr>
<td>%X</td>
<td>Locale’s appropriate time representation.</td>
<td></td>
</tr>
<tr>
<td>%y</td>
<td>Year without century as a decimal number [00,99].</td>
<td></td>
</tr>
<tr>
<td>%Y</td>
<td>Year with century as a decimal number.</td>
<td></td>
</tr>
<tr>
<td>%z</td>
<td>Time zone offset indicating a positive or negative time difference from UTC/GMT of the form +HHMM or -HHMM, where H represents decimal hour digits and M represents decimal minute digits [-23:59, +23:59].</td>
<td></td>
</tr>
<tr>
<td>%Z</td>
<td>Time zone name (no characters if no time zone exists).</td>
<td></td>
</tr>
<tr>
<td>%%</td>
<td>A literal ‘%’ character.</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. When used with the strftime() function, the %p directive only affects the output hour field if the %I directive is used to parse the hour.

2. The range really is 0 to 61; value 60 is valid in timestamps representing leap seconds and value 61 is supported for historical reasons.

3. When used with the strftime() function, %U and %W are only used in calculations when the day of the week and the year are specified.

Here is an example, a format for dates compatible with that specified in the
RFC 2822 Internet email standard. ¹

```python
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

Additional directives may be supported on certain platforms, but only the ones listed here have a meaning standardized by ANSI C. To see the full set of format codes supported on your platform, consult the `strftime(3)` documentation.

On some platforms, an optional field width and precision specification can immediately follow the initial `%` of a directive in the following order; this is also not portable. The field width is normally 2 except for `%j` where it is 3.

```python
time.strptime(string[, format])
```

Parse a string representing a time according to a format. The return value is a `struct_time` as returned by `gmtime()` or `localtime()`.

The `format` parameter uses the same directives as those used by `strftime()`; it defaults to "%a %b %d %H:%M:%S %Y" which matches the formatting returned by `ctime()`. If `string` cannot be parsed according to `format`, or if it has excess data after parsing, `ValueError` is raised. The default values used to fill in any missing data when more accurate values cannot be inferred are (1900, 1, 1, 0, 0, 0, 0, 1, -1). Both `string` and `format` must be strings.

For example:

```python
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")
(time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0, 
  tm_sec=0, tm_wday=3, tm_yday=335, tm_isdst=-1)
```

Support for the `%Z` directive is based on the values contained in `tzname` and whether `daylight` is true. Because of this, it is platform-specific except for recognizing UTC and GMT which are always known (and are considered to be non-daylight savings timezones).

Only the directives specified in the documentation are supported. Because `strftime()` is implemented per platform it can sometimes offer more directives than those listed. But `strptime()` is independent of any platform and thus does not necessarily support all directives available that are not documented as supported.

```python
class time.struct_time
```

The type of the time value sequence returned by `gmtime()`, `localtime()`, and `strptime()`. It is an object with a named tuple interface: values can be accessed by index and by attribute name. The following values are present:

<table>
<thead>
<tr>
<th>Index</th>
<th>Attribute</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>tm_year</code></td>
<td>(for example, 1993)</td>
</tr>
<tr>
<td>1</td>
<td><code>tm_mon</code></td>
<td>range [1, 12]</td>
</tr>
<tr>
<td>2</td>
<td><code>tm_mday</code></td>
<td>range [1, 31]</td>
</tr>
<tr>
<td>3</td>
<td><code>tm_hour</code></td>
<td>range [0, 23]</td>
</tr>
<tr>
<td>4</td>
<td><code>tm_min</code></td>
<td>range [0, 59]</td>
</tr>
<tr>
<td>5</td>
<td><code>tm_sec</code></td>
<td>range [0, 61]; see (2) in <code>strftime()</code> description</td>
</tr>
<tr>
<td>6</td>
<td><code>tm_wday</code></td>
<td>range [0, 6], Monday is 0</td>
</tr>
<tr>
<td>7</td>
<td><code>tm_yday</code></td>
<td>range [1, 366]</td>
</tr>
<tr>
<td>8</td>
<td><code>tm_isdst</code></td>
<td>0, 1 or -1; see below</td>
</tr>
<tr>
<td>N/A</td>
<td><code>tm_zone</code></td>
<td>abbreviation of timezone name</td>
</tr>
<tr>
<td>N/A</td>
<td><code>tm_gmtoff</code></td>
<td>offset east of UTC in seconds</td>
</tr>
</tbody>
</table>

¹ The use of `%z` is now deprecated, but the `%Z` escape that expands to the preferred hour/minute offset is not supported by all ANSI C libraries. Also, a strict reading of the original 1982 RFC 822 standard calls for a two-digit year (%y rather than %Y), but practice moved to 4-digit years long before the year 2000. After that, RFC 822 became obsolete and the 4-digit year has been first recommended by RFC 1123 and then mandated by RFC 2822.
Note that unlike the C structure, the month value is a range of [1, 12], not [0, 11]. A -1 argument as the daylight savings flag, passed to `mktime()` will usually result in the correct daylight savings state to be filled in.

When a tuple with an incorrect length is passed to a function expecting a `struct_time`, or having elements of the wrong type, a `TypeError` is raised.

Changed in version 3.3: `tm_gmtoff` and `tm_zone` attributes are available on platforms with C library supporting the corresponding fields in `struct tm`.

### time.time()

Return the time in seconds since the epoch as a floating point number. Note that even though the time is always returned as a floating point number, not all systems provide time with a better precision than 1 second. While this function normally returns non-decreasing values, it can return a lower value than a previous call if the system clock has been set back between the two calls.

### time.timezone

The offset of the local (non-DST) timezone, in seconds west of UTC (negative in most of Western Europe, positive in the US, zero in the UK).

### time.tzname

A tuple of two strings: the first is the name of the local non-DST timezone, the second is the name of the local DST timezone. If no DST timezone is defined, the second string should not be used.

### time.tzset()

Resets the time conversion rules used by the library routines. The environment variable `TZ` specifies how this is done.

Availability: Unix.

**Note:** Although in many cases, changing the `TZ` environment variable may affect the output of functions like `localtime()` without calling `tzset()`, this behavior should not be relied on.

The `TZ` environment variable should contain no whitespace.

The standard format of the `TZ` environment variable is (whitespace added for clarity):

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

Where the components are:

- **std and dst** Three or more alphanumerics giving the timezone abbreviations. These will be propagated into `time.tzname`
- **offset** The offset has the form: ± hh[:mm[:ss]]. This indicates the value added the local time to arrive at UTC. If preceded by a ‘-‘, the timezone is east of the Prime Meridian; otherwise, it is west. If no offset follows dst, summer time is assumed to be one hour ahead of standard time.
- **start[/time], end[/time]** Indicates when to change to and back from DST. The format of the start and end dates are one of the following:
  - ‘Jn’ The Julian day $n$ (1 $\leq n \leq 365$). Leap days are not counted, so in all years February 28 is day 59 and March 1 is day 60.
  - ‘n’ The zero-based Julian day (0 $\leq n \leq 365$). Leap days are counted, and it is possible to refer to February 29.
  - ‘Mm.n.d’ The $d$’th day (0 $\leq d \leq 6$) or week $n$ of month $m$ of the year (1 $\leq n \leq 5$, 1 $\leq m \leq 12$, where week 5 means “the last $d$ day in month $m$” which may occur in either the fourth or the fifth week). Week 1 is the first week in which the $d$’th day occurs. Day zero is Sunday.

`time` has the same format as `offset` except that no leading sign (‘-‘ or ‘+’) is allowed. The default, if time is not given, is 02:00:00.

16.3. time — Time access and conversions 437
os.environ[‘TZ’] = ‘EST+05EDT,M4.1.0,M10.5.0’
>>>
time.tzset()
>>>
time.strftime(‘%X %x %Z’)
‘02:07:36 05/08/03 EDT’
>>>
os.environ[‘TZ’] = ‘AEST-10AEDT-11,M10.5.0,M3.5.0’
>>>
time.tzset()
>>>
time.strftime(‘%X %x %Z’)
‘16:08:12 05/08/03 AEST’

On many Unix systems (including *BSD, Linux, Solaris, and Darwin), it is more convenient to use the system’s zoneinfo (tzfile(5)) database to specify the timezone rules. To do this, set the TZ environment variable to the path of the required timezone datafile, relative to the root of the systems ‘zoneinfo’ timezone database, usually located at /usr/share/zoneinfo. For example, ‘US/Eastern’, ‘Australia/Melbourne’, ‘Egypt’ or ‘Europe/Amsterdam’.

os.environ[‘TZ’] = ‘US/Eastern’
>>>
time.tzset()
>>>
time.tzname
(‘EST’, ‘EDT’)
>>>
os.environ[‘TZ’] = ‘Egypt’
>>>
time.tzset()
>>>
time.tzname
(‘EET’, ‘EEST’)

See Also:
Module datetime  More object-oriented interface to dates and times.
Module locale  Internationalization services. The locale setting affects the interpretation of many format specifiers in strftime() and strptime().
Module calendar  General calendar-related functions. timegm() is the inverse of gmtime() from this module.

16.4 argparse — Parser for command-line options, arguments and sub-commands

New in version 3.2. Source code: Lib/argparse.py

Tutorial
This page contains the API reference information. For a more gentle introduction to Python command-line parsing, have a look at the argparse tutorial.

The argparse module makes it easy to write user-friendly command-line interfaces. The program defines what arguments it requires, and argparse will figure out how to parse those out of sys.argv. The argparse module also automatically generates help and usage messages and issues errors when users give the program invalid arguments.

16.4.1 Example

The following code is a Python program that takes a list of integers and produces either the sum or the max:
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
    const=sum, default=max,
    help='sum the integers (default: find the max)')

args = parser.parse_args()
print(args.accumulate(args.integers))

Assuming the Python code above is saved into a file called prog.py, it can be run at the command line and provides useful help messages:

$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]
Process some integers.

positional arguments:
  N   an integer for the accumulator

optional arguments:
  -h, --help    show this help message and exit
  --sum         sum the integers (default: find the max)

When run with the appropriate arguments, it prints either the sum or the max of the command-line integers:

$ python prog.py 1 2 3 4
4

$ python prog.py 1 2 3 4 --sum
10

If invalid arguments are passed in, it will issue an error:

$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: `a`

The following sections walk you through this example.

Creating a parser

The first step in using the argparse is creating an ArgumentParser object:

```python
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

The ArgumentParser object will hold all the information necessary to parse the command line into Python data types.

Adding arguments

Filling an ArgumentParser with information about program arguments is done by making calls to the add_argument() method. Generally, these calls tell the ArgumentParser how to take the strings on the command line and turn them into objects. This information is stored and used when parse_args() is called. For example:

```python
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
    help='an integer for the accumulator')
>>> parser.add_argument('--sum', dest='accumulate', action='store_const',
    const=sum, default=max,
    help='sum the integers (default: find the max)')
```
... const=sum, default=max,
... help='sum the integers (default: find the max)’

Later, calling `parse_args()` will return an object with two attributes, `integers` and `accumulate`. The `integers` attribute will be a list of one or more ints, and the `accumulate` attribute will be either the `sum()` function, if `--sum` was specified at the command line, or the `max()` function if it was not.

### Parsing arguments

`ArgumentParser` parses arguments through the `parse_args()` method. This will inspect the command line, convert each argument to the appropriate type and then invoke the appropriate action. In most cases, this means a simple `Namespace` object will be built up from attributes parsed out of the command line:

```python
>>> parser.parse_args(['--sum', '7', '-1', '42'])
Namespace(accumulate=<built-in function sum>, integers=[7, -1, 42])
```

In a script, `parse_args()` will typically be called with no arguments, and the `ArgumentParser` will automatically determine the command-line arguments from `sys.argv`.

### 16.4.2 ArgumentParser objects

```python
class argparse.ArgumentParser (prog=None, usage=None, description=None, epilog=None, parents=[], formatter_class=argparse.HelpFormatter, prefix_chars='-', fromfile_prefix_chars=None, argument_default=None, conflict_handler='error', add_help=True)
```

Create a new `ArgumentParser` object. All parameters should be passed as keyword arguments. Each parameter has its own more detailed description below, but in short they are:

- **prog** - The name of the program (default: `sys.argv[0]`)
- **usage** - The string describing the program usage (default: generated from arguments added to parser)
- **description** - Text to display before the argument help (default: none)
- **epilog** - Text to display after the argument help (default: none)
- **parents** - A list of `ArgumentParser` objects whose arguments should also be included
- **formatter_class** - A class for customizing the help output
- **prefix_chars** - The set of characters that prefix optional arguments (default: ‘-‘)
- **fromfile_prefix_chars** - The set of characters that prefix files from which additional arguments should be read (default: `None`)
- **argument_default** - The global default value for arguments (default: `None`)
- **conflict_handler** - The strategy for resolving conflicting optionals (usually unnecessary)
- **add_help** - Add a -h/-help option to the parser (default: `True`)

The following sections describe how each of these are used.

### prog

By default, `ArgumentParser` objects uses `sys.argv[0]` to determine how to display the name of the program in help messages. This default is almost always desirable because it will make the help messages match how the program was invoked on the command line. For example, consider a file named `myprogram.py` with the following code:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```
The help for this program will display `myprogram.py` as the program name (regardless of where the program was invoked from):

```bash
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
  -h, --help    show this help message and exit
  --foo FOO     foo help

$ cd ..
$ python subdir\myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
  -h, --help    show this help message and exit
  --foo FOO     foo help
```

To change this default behavior, another value can be supplied using the `prog=` argument to `ArgumentParser`:

```python
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

optional arguments:
  -h, --help    show this help message and exit

Note that the program name, whether determined from `sys.argv[0]` or from the `prog=` argument, is available to help messages using the `%prog` format specifier.

```python
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

optional arguments:
  -h, --help    show this help message and exit
  --foo FOO     foo of the myprogram program
```

### usage

By default, `ArgumentParser` calculates the usage message from the arguments it contains:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [-h] [--foo [FOO]] bar [bar ...]

positional arguments:
  bar       bar help

optional arguments:
  -h, --help    show this help message and exit
  --foo [FOO]   foo help
```

The default message can be overridden with the `usage=` keyword argument:

```python
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%(prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
```
usage: PROG [options]

positional arguments:
    bar     bar help

optional arguments:
    -h, --help     show this help message and exit
    --foo [FOO]   foo help

The $(prog)s format specifier is available to fill in the program name in your usage messages.

description

Most calls to the ArgumentParser constructor will use the description= keyword argument. This argument gives a brief description of what the program does and how it works. In help messages, the description is displayed between the command-line usage string and the help messages for the various arguments:

```python
>>> parser = argparse.ArgumentParser(description='A foo that bars')
>>> parser.print_help()
usage: argparse.py [-h]
A foo that bars

optional arguments:
    -h, --help     show this help message and exit
```

By default, the description will be line-wrapped so that it fits within the given space. To change this behavior, see the formatter_class argument.

epi

Some programs like to display additional description of the program after the description of the arguments. Such text can be specified using the epilog= argument to ArgumentParser:

```python
>>> parser = argparse.ArgumentParser(...)  
...   description='A foo that bars',
...   epilog='And that’s how you’d foo a bar')
>>> parser.print_help()
usage: argparse.py [-h]
A foo that bars

optional arguments:
    -h, --help     show this help message and exit
```

And that’s how you’d foo a bar

As with the description argument, the epilog= text is by default line-wrapped, but this behavior can be adjusted with the formatter_class argument to ArgumentParser.

parents

Sometimes, several parsers share a common set of arguments. Rather than repeating the definitions of these arguments, a single parser with all the shared arguments and passed to parents= argument to ArgumentParser can be used. The parents= argument takes a list of ArgumentParser objects, collects all the positional and optional actions from them, and adds these actions to the ArgumentParser object being constructed:

```python
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)
```
foo_parser = argparse.ArgumentParser(parents=[parent_parser])
foo_parser.add_argument('foo')
foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

bar_parser = argparse.ArgumentParser(parents=[parent_parser])
bar_parser.add_argument('--bar')
bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)

Note that most parent parsers will specify add_help=False. Otherwise, the ArgumentParser will see two -h/--help options (one in the parent and one in the child) and raise an error.

**Note:** You must fully initialize the parsers before passing them via parents=. If you change the parent parsers after the child parser, those changes will not be reflected in the child.

**formatter_class**

ArgumentParser objects allow the help formatting to be customized by specifying an alternate formatting class. Currently, there are four such classes:

class argparse.RawDescriptionHelpFormatter
class argparse.RawTextHelpFormatter
class argparse.ArgumentDefaultsHelpFormatter
class argparse.MetavarTypeHelpFormatter

RawDescriptionHelpFormatter and RawTextHelpFormatter give more control over how textual descriptions are displayed. By default, ArgumentParser objects line-wrap the description and epilog texts in command-line help messages:

```python
parser = argparse.ArgumentParser(
    prog='PROG',
    description='''this description
    was indented weird
    but that is okay''',
    epilog='''
    likewise for this epilog whose whitespace will
    be cleaned up and whose words will be wrapped
    across a couple lines'''
)
parser.print_help()
```

usage: PROG [-h]

this description was indented weird but that is okay

optional arguments:
- h, --help show this help message and exit

likewise for this epilog whose whitespace will be cleaned up and whose words will be wrapped across a couple lines

Passing RawDescriptionHelpFormatter as formatter_class= indicates that description and epilog are already correctly formatted and should not be line-wraped:

```python
parser = argparse.ArgumentParser(
    prog='PROG',
    formatter_class=argparse.RawDescriptionHelpFormatter,
    description=textwrap.dedent('''
    Please do not mess up this text!
    ''')
)
```
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!

optional arguments:
- h, --help show this help message and exit

RawTextHelpFormatter maintains whitespace for all sorts of help text, including argument descriptions.

ArgumentDefaultsHelpFormatter automatically adds information about default values to each of the argument help messages:

```python
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
>>> parser.add_argument('--foo', type=int, default=42, help='FOO!')
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar [bar ...]]

positional arguments:
  bar BAR! (default: [1, 2, 3])

optional arguments:
  - h, --help show this help message and exit
    --foo FOO FOO! (default: 42)
```

MetavarTypeHelpFormatter uses the name of the type argument for each argument as the display name for its values (rather than using the dest as the regular formatter does):

```python
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.MetavarTypeHelpFormatter)
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', type=float)
>>> parser.print_help()
usage: PROG [-h] [--foo int] float

positional arguments:
  float

optional arguments:
  - h, --help show this help message and exit
    --foo int
```

prefix_chars

Most command-line options will use -- as the prefix, e.g. -f/--foo. Parsers that need to support different or additional prefix characters, e.g. for options like +f or /foo, may specify them using the prefix_chars argument to the ArgumentParser constructor:
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+-')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')

The prefix_chars= argument defaults to ‘-’. Supplying a set of characters that does not include – will cause -f/--foo options to be disallowed.

fromfile_prefix_chars

Sometimes, for example when dealing with a particularly long argument lists, it may make sense to keep the list of arguments in a file rather than typing it out at the command line. If the fromfile_prefix_chars= argument is given to the ArgumentParser constructor, then arguments that start with any of the specified characters will be treated as files, and will be replaced by the arguments they contain. For example:

>>> with open('args.txt', 'w') as fp:
...     fp.write('-f
     bar')

>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')

Arguments read from a file must by default be one per line (but see also convert_arg_line_to_args()) and are treated as if they were in the same place as the original file referencing argument on the command line. So in the example above, the expression ['-f', 'foo', '@args.txt'] is considered equivalent to the expression ['-f', 'foo', '-f', 'bar'].

The fromfile_prefix_chars= argument defaults to None, meaning that arguments will never be treated as file references.

argument_default

Generally, argument defaults are specified either by passing a default to add_argument() or by calling the set_defaults() methods with a specific set of name-value pairs. Sometimes however, it may be useful to specify a single parser-wide default for arguments. This can be accomplished by passing the argument_default= keyword argument to ArgumentParser. For example, to globally suppress attribute creation on parse_args() calls, we supply argument_default=argparse.SUPPRESS:

>>> parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar', nargs='?')
>>> parser.parse_args(['--foo', '1', 'BAR'])
Namespace(bar='BAR', foo='1')
>>> parser.parse_args([])
Namespace()

conflict_handler

ArgumentParser objects do not allow two actions with the same option string. By default, ArgumentParser objects raises an exception if an attempt is made to create an argument with an option string that is already in use:

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
  ...
ArgumentError: argument --foo: conflicting option string(s): --foo
Sometimes (e.g. when using parents) it may be useful to simply override any older arguments with the same option string. To get this behavior, the value 'resolve' can be supplied to the conflict_handler= argument of ArgumentParser:

```python
>>> parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]
```

optional arguments:
- `-h`, `--help` show this help message and exit
- `-f FOO` old foo help
- `--foo FOO` new foo help

Note that ArgumentParser objects only remove an action if all of its option strings are overridden. So, in the example above, the old `-f/--foo` action is retained as the `-f` action, because only the `--foo` option string was overridden.

add_help

By default, ArgumentParser objects add an option which simply displays the parser’s help message. For example, consider a file named myprogram.py containing the following code:

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

If `-h` or `--help` is supplied at the command line, the ArgumentParser help will be printed:

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]
```

optional arguments:
- `-h`, `--help` show this help message and exit
- `--foo FOO` foo help

Occasionally, it may be useful to disable the addition of this help option. This can be achieved by passing `False` as the add_help= argument to ArgumentParser:

```python
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [-foo FOO]
```

optional arguments:
- `--foo FOO` foo help

The help option is typically `-h/--help`. The exception to this is if the prefix_chars= is specified and does not include --, in which case `-h` and `--help` are not valid options. In this case, the first character in prefix_chars is used to prefix the help options:

```python
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]
```

optional arguments:
- `+h`, `++help` show this help message and exit
16.4.3 The add_argument() method

ArgumentParser.add_argument(name or flags..., [action], [nargs], [const], [default], [type], [choices], [required], [help], [metavar], [dest])

Define how a single command-line argument should be parsed. Each parameter has its own more detailed description below, but in short they are:

- **name or flags** - Either a name or a list of option strings, e.g. `foo` or `-f`, `--foo`.
- **action** - The basic type of action to be taken when this argument is encountered at the command line.
- **nargs** - The number of command-line arguments that should be consumed.
- **const** - A constant value required by some action and nargs selections.
- **default** - The value produced if the argument is absent from the command line.
- **type** - The type to which the command-line argument should be converted.
- **choices** - A container of the allowable values for the argument.
- **required** - Whether or not the command-line option may be omitted (optionals only).
- **help** - A brief description of what the argument does.
- **metavar** - A name for the argument in usage messages.
- **dest** - The name of the attribute to be added to the object returned by `parse_args()`.

The following sections describe how each of these are used.

**name or flags**

The `add_argument()` method must know whether an optional argument, like `-f` or `--foo`, or a positional argument, like a list of filenames, is expected. The first arguments passed to `add_argument()` must therefore be either a series of flags, or a simple argument name. For example, an optional argument could be created like:

```python
>>> parser.add_argument('-f', '--foo')
```

while a positional argument could be created like:

```python
>>> parser.add_argument('bar')
```

When `parse_args()` is called, optional arguments will be identified by the `-` prefix, and the remaining arguments will be assumed to be positional:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
Namespace(bar='BAR', foo='FOO')
usage: PROG [-h] [-f FOO] bar
PROG: error: too few arguments
```

**action**

ArgumentParser objects associate command-line arguments with actions. These actions can do just about anything with the command-line arguments associated with them, though most actions simply add an attribute to the object returned by `parse_args()`. The action keyword argument specifies how the command-line arguments should be handled. The supported actions are:

- **'store'** - This just stores the argument’s value. This is the default action. For example:
```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args('--foo 1'.split())
Namespace(foo='1')
```

- **'store_const'** - This stores the value specified by the `const` keyword argument. (Note that the `const` keyword argument defaults to the rather unhelpful `None`.) The 'store_const' action is most commonly used with optional arguments that specify some sort of flag. For example:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_const', const=42)
>>> parser.parse_args('--foo'.split())
Namespace(foo=42)
```

- **'store_true' and 'store_false'** - These store the values `True` and `False` respectively. These are special cases of 'store_const'. For example:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('--bar', action='store_false')
>>> parser.parse_args('--foo --bar'.split())
Namespace(bar=False, foo=True)
```

- **'append'** - This stores a list, and appends each argument value to the list. This is useful to allow an option to be specified multiple times. Example usage:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append')
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['1', '2'])
```

- **'append_const'** - This stores a list, and appends the value specified by the `const` keyword argument to the list. (Note that the `const` keyword argument defaults to `None`.) The 'append_const' action is typically useful when multiple arguments need to store constants to the same list. For example:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const', const=str)
>>> parser.add_argument('--int', dest='types', action='append_const', const=int)
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<class 'str'>, <class 'int'>])
```

- **'count'** - This counts the number of times a keyword argument occurs. For example, this is useful for increasing verbosity levels:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count')
>>> parser.parse_args('-vvv'.split())
Namespace(verbose=3)
```

- **'help'** - This prints a complete help message for all the options in the current parser and then exits. By default a help action is automatically added to the parser. See `ArgumentParser` for details of how the output is created.

- **'version'** - This expects a `version=` keyword argument in the `add_argument()` call, and prints version information and exits when invoked:

```python
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--version', action='version', version='%(prog)s 2.0')
>>> parser.parse_args(['--version'])
PROG 2.0
```

You can also specify an arbitrary action by passing an object that implements the Action API. The easiest way to do this is to extend `argparse.Action`, supplying an appropriate `__call__` method. The `__call__` method should accept four parameters:
• **parser** - The ArgumentParser object which contains this action.

• **namespace** - The Namespace object that will be returned by `parse_args()`. Most actions add an attribute to this object.

• **values** - The associated command-line arguments, with any type conversions applied. (Type conversions are specified with the `type` keyword argument to `add_argument()`)  

• **option_string** - The option string that was used to invoke this action. The `option_string` argument is optional, and will be absent if the action is associated with a positional argument.

An example of a custom action:

```python
>>> class FooAction(argparse.Action):
...     def __call__(self, parser, namespace, values, option_string=None):
...         print('%r %r %r' % (namespace, values, option_string))
...         setattr(namespace, self.dest, values)
...`

>>> parser = argparse.ArgumentParser()

>>> parser.add_argument('--foo', action=FooAction)

>>> parser.add_argument('bar', action=FooAction)

>>> args = parser.parse_args('1 --foo 2'.split())

Namespace(bar=None, foo=None) '1' None

Namespace(bar='1', foo=None) '2' '--foo'

>>> args
Namespace(bar='1', foo='2')
```

### nargs

ArgumentParser objects usually associate a single command-line argument with a single action to be taken. The `nargs` keyword argument associates a different number of command-line arguments with a single action. The supported values are:

• **N** (an integer). N arguments from the command line will be gathered together into a list. For example:

```python
>>> parser = argparse.ArgumentParser()

>>> parser.add_argument('--foo', nargs=2)

>>> parser.add_argument('bar', nargs=1)

>>> parser.parse_args('c --foo a b'.split())

Namespace(bar=['c'], foo=['a', 'b'])
```

Note that `nargs=1` produces a list of one item. This is different from the default, in which the item is produced by itself.

• **’?’**: One argument will be consumed from the command line if possible, and produced as a single item. If no command-line argument is present, the value from `default` will be produced. Note that for optional arguments, there is an additional case - the option string is present but not followed by a command-line argument. In this case the value from `const` will be produced. Some examples to illustrate this:

```python
>>> parser = argparse.ArgumentParser()

>>> parser.add_argument('--foo', nargs='?', const='c', default='d')

>>> parser.add_argument('bar', nargs='?', default='d')

>>> parser.parse_args('XX --foo YY'.split())

Namespace(bar='XX', foo='YY')

>>> parser.parse_args('XX --foo'.split())

Namespace(bar='XX', foo='c')

>>> parser.parse_args('').split())

Namespace(bar='d', foo='d')
```

One of the more common uses of `nargs='?'` is to allow optional input and output files:

```python
>>> parser = argparse.ArgumentParser()

>>> parser.add_argument('infile', nargs='?', type=argparse.FileType('r'),
```
... default=sys.stdin)
>>> parser.add_argument('outfile', nargs='?', type=argparse.FileType('w'),
... default=sys.stdout)
>>> parser.parse_args(['input.txt', 'output.txt'])
Namespace(infile=<_io.TextIOWrapper name='input.txt' encoding='UTF-8'>,
       outfile=<_io.TextIOWrapper name='output.txt' encoding='UTF-8'>)
>>> parser.parse_args([])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>,
       outfile=<_io.TextIOWrapper name='<stdout>' encoding='UTF-8'>)

• `*` All command-line arguments present are gathered into a list. Note that it generally doesn’t make much sense to have more than one positional argument with nargs='*', but multiple optional arguments with nargs='*' is possible. For example:

>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='*')
>>> parser.add_argument('--bar', nargs='*')
>>> parser.add_argument('baz', nargs='*')
>>> parser.parse_args('a b --foo x y --bar 1 2'.split())
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])

• `+` Just like `*`, all command-line args present are gathered into a list. Additionally, an error message will be generated if there wasn’t at least one command-line argument present. For example:

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.parse_args('a b'.split())
Namespace(foo=['a', 'b'])
>>> parser.parse_args('').split()usage: PROG [-h] foo [foo ...]
PROG: error: too few arguments

• argparse.REMAINDER All the remaining command-line arguments are gathered into a list. This is commonly useful for command line utilities that dispatch to other command line utilities:

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo')
>>> parser.add_argument('command')
>>> parser.add_argument('args', nargs=argparse.REMAINDER)
>>> print(parser.parse_args('--foo B cmd --arg1 XX ZZ'.split()))
Namespace(args=['--arg1', 'XX', 'ZZ'], command='cmd', foo='B')

If the nargs keyword argument is not provided, the number of arguments consumed is determined by the action. Generally this means a single command-line argument will be consumed and a single item (not a list) will be produced.

const

The const argument of add_argument() is used to hold constant values that are not read from the command line but are required for the various ArgumentParser actions. The two most common uses of it are:

• When add_argument() is called with action='store_const' or action='append_const'. These actions add the const value to one of the attributes of the object returned by parse_args(). See the action description for examples.

• When add_argument() is called with option strings (like --f or --foo) and nargs='?'. This creates an optional argument that can be followed by zero or one command-line arguments. When parsing the command line, if the option string is encountered with no command-line argument following it, the value of const will be assumed instead. See the nargs description for examples.

The const keyword argument defaults to None.
default

All optional arguments and some positional arguments may be omitted at the command line. The `default` keyword argument of `add_argument()`, whose value defaults to `None`, specifies what value should be used if the command-line argument is not present. For optional arguments, the `default` value is used when the option string was not present at the command line:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args('--foo 2'.split())
Namespace(foo='2')
```

If the `default` value is a string, the parser parses the value as if it were a command-line argument. In particular, the parser applies any type conversion argument, if provided, before setting the attribute on the `Namespace` return value. Otherwise, the parser uses the value as is:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

For positional arguments with `nargs` equal to `?` or `*`, the `default` value is used when no command-line argument was present:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args('a'.split())
Namespace(foo='a')
```

Providing `default=argparse.SUPPRESS` causes no attribute to be added if the command-line argument was not present:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([[]])
Namespace()
```

type

By default, `ArgumentParser` objects read command-line arguments in as simple strings. However, quite often the command-line string should instead be interpreted as another type, like a `float` or `int`. The `type` keyword argument of `add_argument()` allows any necessary type-checking and type conversions to be performed. Common built-in types and functions can be used directly as the value of the `type` argument:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.add_argument('bar', type=open)
```

See the section on the `default` keyword argument for information on when the `type` argument is applied to default arguments.
writable file:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar', type=argparse.FileType('w'))
>>> parser.parse_args(['out.txt'])
Namespace(bar=<_io.TextIOWrapper name='out.txt' encoding='UTF-8'>)
```

type= can take any callable that takes a single string argument and returns the converted value:

```python
>>> def perfect_square(string):
...     value = int(string)
...     sqrt = math.sqrt(value)
...     if sqrt != int(sqrt):
...         msg = f"{string} is not a perfect square" % string
...         raise argparse.ArgumentTypeError(msg)
...     return value
...

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=perfect_square)
>>> parser.parse_args('9'.split())
Namespace(foo=9)
>>> parser.parse_args('7'.split())
usage: PROG [-h] foo
PROG: error: argument foo: '7' is not a perfect square
```

The choices keyword argument may be more convenient for type checkers that simply check against a range of values:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=int, choices=range(5, 10))
>>> parser.parse_args('7'.split())
Namespace(foo=7)
>>> parser.parse_args('11'.split())
usage: PROG [-h] {5,6,7,8,9}
PROG: error: argument foo: invalid choice: 11 (choose from 5, 6, 7, 8, 9)
```

See the choices section for more details.

**choices**

Some command-line arguments should be selected from a restricted set of values. These can be handled by passing a container object as the choices keyword argument to add_argument(). When the command line is parsed, argument values will be checked, and an error message will be displayed if the argument was not one of the acceptable values:

```python
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock', 'paper', 'scissors')
```

Note that inclusion in the choices container is checked after any type conversions have been performed, so the type of the objects in the choices container should match the type specified:

```python
>>> parser = argparse.ArgumentParser(prog='doors.py')
>>> parser.add_argument('door', type=int, choices=range(1, 4))
>>> print(parser.parse_args(['3']))
Namespace(door=3)
>>> parser.parse_args(['4'])
```
usage: doors.py [-h] {1,2,3}

doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)

Any object that supports the `in` operator can be passed as the `choices` value, so `dict` objects, `set` objects, custom containers, etc. are all supported.

**required**

In general, the `argparse` module assumes that flags like `-f` and `--bar` indicate *optional* arguments, which can always be omitted at the command line. To make an option *required*, `True` can be specified for the `required=` keyword argument to `add_argument()`:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
```

As the example shows, if an option is marked as `required`, `parse_args()` will report an error if that option is not present at the command line.

**Note:** Required options are generally considered bad form because users expect *options* to be *optional*, and thus they should be avoided when possible.

**help**

The `help` value is a string containing a brief description of the argument. When a user requests help (usually by using `-h` or `--help` at the command line), these help descriptions will be displayed with each argument:

```python
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', action='store_true', ...
...                     help='foo the bars before frobbling')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
...                      help='the bar to frobble (default: 42)')
```

The `help` strings can include various format specifiers to avoid repetition of things like the program name or the argument `default`. The available specifiers include the program name, `%prog` and most keyword arguments to `add_argument()`, e.g. `%prog` and most keyword arguments to `add_argument()`, `%(type)s`, etc.:

```python
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42, ...
...                      help='the bar to frobble (default: 42)')
```

positional arguments:
- bar the bar to frobble (default: 42)
optional arguments:
   -h, --help    show this help message and exit

As the help string supports %-formatting, if you want a literal % to appear in the help string, you must escape it as %%.  

argparse supports silencing the help entry for certain options, by setting the help value to argparse.SUPPRESS:

```python
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]
optional arguments:
   -h, --help    show this help message and exit
```

metavar

When ArgumentParser generates help messages, it needs some way to refer to each expected argument. By default, ArgumentParser objects use the dest value as the “name” of each object. By default, for positional argument actions, the dest value is used directly, and for optional argument actions, the dest value is uppercased. So, a single positional argument with dest='bar' will be referred to as bar. A single optional argument --foo that should be followed by a single command-line argument will be referred to as FOO. An example:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage: [-h] [---foo FOO] bar
positional arguments:
   bar
optional arguments:
   -h, --help    show this help message and exit
   --foo FOO
```

An alternative name can be specified with metavar:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage: [-h] [---foo YYY] XXX
positional arguments:
   XXX
optional arguments:
   -h, --help    show this help message and exit
   --foo YYY
```

Note that metavar only changes the displayed name - the name of the attribute on the parse_args() object is still determined by the dest value.
Different values of \texttt{nargs} may cause the \texttt{metavar} to be used multiple times. Providing a tuple to \texttt{metavar} specifies a different display for each of the arguments:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs=2, metavar=('bar', 'baz'))
>>> parser.print_help()
usage: PROG [-h] [-x X X] [--foo bar baz]
```

optional arguments:
- \(-h\), \(--help\)  show this help message and exit
- \(-x\) \(X X\)
- \(--foo\) \(bar\) \(baz\)

\texttt{dest}

Most \texttt{ArgumentParser} actions add some value as an attribute of the object returned by \texttt{parse_args()}. The name of this attribute is determined by the \texttt{dest} keyword argument of \texttt{add_argument()}. For positional argument actions, \texttt{dest} is normally supplied as the first argument to \texttt{add_argument()}:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args('XXX'.split())
Namespace(bar='XXX')
```

For optional argument actions, the value of \texttt{dest} is normally inferred from the option strings. \texttt{ArgumentParser} generates the value of \texttt{dest} by taking the first long option string and stripping away the initial \texttt{--} string. If no long option strings were supplied, \texttt{dest} will be derived from the first short option string by stripping the initial \texttt{-} character. Any internal \texttt{-} characters will be converted to \texttt{_} characters to make sure the string is a valid attribute name. The examples below illustrate this behavior:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args('-f 1 -x 2'.split())
Namespace(foo_bar='1', x='2')
```

\texttt{dest} allows a custom attribute name to be provided:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

### 16.4.4 The \texttt{parse_args()} method

\texttt{ArgumentParser.parse_args(\texttt{args}=None, \texttt{namespace}=None)}

Convert argument strings to objects and assign them as attributes of the namespace. Return the populated namespace.

Previous calls to \texttt{add_argument()} determine exactly what objects are created and how they are assigned. See the documentation for \texttt{add_argument()} for details.

By default, the argument strings are taken from \texttt{sys.argv}, and a new empty \texttt{Namespace} object is created for the attributes.
Option value syntax

The `parse_args()` method supports several ways of specifying the value of an option (if it takes one). In the simplest case, the option and its value are passed as two separate arguments:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args('-x X'.split())
Namespace(foo=None, x='X')
>>> parser.parse_args('--foo FOO'.split())
Namespace(foo='FOO', x=None)
```

For long options (options with names longer than a single character), the option and value can also be passed as a single command-line argument, using `=` to separate them:

```
>>> parser.parse_args('--foo=FOO'.split())
Namespace(foo='FOO', x=None)
```

For short options (options only one character long), the option and its value can be concatenated:

```
>>> parser.parse_args('-xX'.split())
Namespace(foo=None, x='X')
```

Several short options can be joined together, using only a single `-` prefix, as long as only the last option (or none of them) requires a value:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', action='store_true')
>>> parser.add_argument('-y', action='store_true')
>>> parser.add_argument('-z')
>>> parser.parse_args('-xyzZ'.split())
Namespace(x=True, y=True, z='Z')
```

Invalid arguments

While parsing the command line, `parse_args()` checks for a variety of errors, including ambiguous options, invalid types, invalid options, wrong number of positional arguments, etc. When it encounters such an error, it exits and prints the error along with a usage message:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')

# invalid type
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'

# invalid option
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

# wrong number of arguments
>>> parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
```
Arguments containing –

The `parse_args()` method attempts to give errors whenever the user has clearly made a mistake, but some situations are inherently ambiguous. For example, the command-line argument `-1` could either be an attempt to specify an option or an attempt to provide a positional argument. The `parse_args()` method is cautious here: positional arguments may only begin with – if they look like negative numbers and there are no options in the parser that look like negative numbers:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional argument
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional arguments
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='5', x='-1')

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')

>>> # negative number options present, so -1 is an option
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')

>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

>>> # negative number options present, so both -1s are options
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

If you have positional arguments that must begin with – and don’t look like negative numbers, you can insert the pseudo-argument ‘--’ which tells `parse_args()` that everything after that is a positional argument:

```python
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

Argument abbreviations

The `parse_args()` method allows long options to be abbreviated if the abbreviation is unambiguous:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')

>>> parser.parse_args('-bac MMM'.split())
Namespace(bacon='MMM', badger=None)

>>> parser.parse_args('-bad WOOD'.split())
Namespace(bacon=None, badger='WOOD')

>>> parser.parse_args('-ba BA'.split())
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

An error is produced for arguments that could produce more than one options.
Beyond `sys.argv`

Sometimes it may be useful to have an `ArgumentParser` parse arguments other than those of `sys.argv`. This can be accomplished by passing a list of strings to `parse_args()`. This is useful for testing at the interactive prompt:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(  
...     'integers', metavar='int', type=int, choices=range(10),  
...     nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(  
...     '--sum', dest='accumulate', action='store_const', const=sum,  
...     default=max, help='sum the integers (default: find the max)')
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3, 4])
>>> parser.parse_args('1 2 3 4 --sum'.split())
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```

The `Namespace` object

```python
class argparse.Namespace
    Simple class used by default by `parse_args()` to create an object holding attributes and return it.
```

This class is deliberately simple, just an `object` subclass with a readable string representation. If you prefer to have dict-like view of the attributes, you can use the standard Python idiom, `vars()`:

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

It may also be useful to have an `ArgumentParser` assign attributes to an already existing object, rather than a new `Namespace` object. This can be achieved by specifying the `namespace=` keyword argument:

```python
>>> class C:
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

16.4.5 Other utilities

Sub-commands

```python
ArgumentParser.add_subparsers([title], description, prog[, parser_class][, action][, option_string][, dest][, help][, metavar])
```

Many programs split up their functionality into a number of sub-commands, for example, the `svn` program can invoke sub-commands like `svn checkout`, `svn update`, and `svn commit`. Splitting up functionality this way can be a particularly good idea when a program performs several different functions which require different kinds of command-line arguments. `ArgumentParser` supports the creation of such sub-commands with the `add_subparsers()` method. The `add_subparsers()` method is normally called with no arguments and returns a special action object. This object has a single method, `add_parser()`, which takes a command name and any `ArgumentParser` constructor arguments, and returns an `ArgumentParser` object that can be modified as usual.
Description of parameters:

- **title** - title for the sub-parser group in help output; by default “subcommands” if description is provided, otherwise uses title for positional arguments
- **description** - description for the sub-parser group in help output, by default None
- **prog** - usage information that will be displayed with sub-command help, by default the name of the program and any positional arguments before the subparser argument
- **parser_class** - class which will be used to create sub-parser instances, by default the class of the current parser (e.g. ArgumentParser)
- **dest** - name of the attribute under which sub-command name will be stored; by default None and no value is stored
- **help** - help for sub-parser group in help output, by default None
- **metavar** - string presenting available sub-commands in help; by default it is None and presents sub-commands in form {cmd1, cmd2, ..}

Some example usage:

```python
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true', help='foo help')
>>> subparsers = parser.add_subparsers(help='sub-command help')

>>> # create the parser for the "a" command
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')

>>> # create the parser for the "b" command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices='XYZ', help='baz help')

>>> # parse some argument lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

Note that the object returned by `parse_args()` will only contain attributes for the main parser and the subparser that was selected by the command line (and not any other subparsers). So in the example above, when the `a` command is specified, only the `foo` and `bar` attributes are present, and when the `b` command is specified, only the `foo` and `baz` attributes are present.

Similarly, when a help message is requested from a subparser, only the help for that particular parser will be printed. The help message will not include parent parser or sibling parser messages. (A help message for each subparser command, however, can be given by supplying the `help=` argument to `add_parser()` as above.)

```python
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b} sub-command help
    a  a help
    b  b help

optional arguments:
  -h, --help  show this help message and exit
  --foo  foo help
```
>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar  bar help

optional arguments:
  -h, --help  show this help message and exit

>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

optional arguments:
  -h, --help  show this help message and exit
  --baz {X,Y,Z}  baz help

The `add_subparsers()` method also supports `title` and `description` keyword arguments. When either is present, the subparser’s commands will appear in their own group in the help output. For example:

```python
>>> parser = argparse.ArgumentParser()

>>> subparsers = parser.add_subparsers(title='subcommands',
...                                     description='valid subcommands',
...                                     help='additional help')

>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')

>>> parser.parse_args([-h])
usage: [-h] {foo,bar} ...

optional arguments:
  -h, --help  show this help message and exit

subcommands:
  valid subcommands
    {foo,bar}  additional help
```

Furthermore, `add_parser` supports an additional `aliases` argument, which allows multiple strings to refer to the same subparser. This example, like `svn`, aliases `co` as a shorthand for `checkout`:

```python
>>> parser = argparse.ArgumentParser()

>>> subparsers = parser.add_subparsers()

>>> checkout = subparsers.add_parser('checkout', aliases=['co'])

>>> checkout.add_argument('foo')

>>> parser.parse_args(['co', 'bar'])
Namespace(foo='bar')
```

One particularly effective way of handling sub-commands is to combine the use of the `add_subparsers()` method with calls to `set_defaults()` so that each subparser knows which Python function it should execute. For example:

```python
>>> # sub-command functions
>>> def foo(args):
...     print(args.x * args.y)
...     ...

>>> def bar(args):
...     print('((%(s)s))' % args.z)
...     ...
```
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()

>>> # create the parser for the "foo" command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)

>>> # create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)

>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
2.0

>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
(‘XYZYX’)

This way, you can let `parse_args()` do the job of calling the appropriate function after argument parsing is complete. Associating functions with actions like this is typically the easiest way to handle the different actions for each of your subparsers. However, if it is necessary to check the name of the subparser that was invoked, the `dest` keyword argument to the `add_subparsers()` call will work:

```python
>>> parser = argparse.ArgumentParser()

>>> subparsers = parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')

>>> args = parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

**FileType objects**

The `argparse.FileType` factory creates objects that can be passed to the type argument of `ArgumentParser.add_argument()`. Arguments that have `FileType` objects as their type will open command-line arguments as files with the requested modes and buffer sizes:

```python
>>> parser = argparse.ArgumentParser()

>>> parser.add_argument('--output', type=argparse.FileType('wb', 0))
>>> parser.parse_args(['--output', 'out'])
Namespace(output=<_io.BufferedWriter name='out'>)
```

`FileType` objects understand the pseudo-argument ‘-’ and automatically convert this into `sys.stdin` for readable `FileType` objects and `sys.stdout` for writable `FileType` objects:

```python
>>> parser = argparse.ArgumentParser()

>>> parser.add_argument('infile', type=argparse.FileType('r'))
```
>>> parser.parse_args(['-'])
Namespace(infile=_io.TextIOWrapper name='<stdin>' encoding='UTF-8')

Argument groups

ArgumentParser.add_argument_group(title=None, description=None)

By default, ArgumentParser groups command-line arguments into “positional arguments” and “optional arguments” when displaying help messages. When there is a better conceptual grouping of arguments than this default one, appropriate groups can be created using the add_argument_group() method:

```python
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
>>> parser.print_help()
usage: PROG [--foo FOO] bar

group:
    bar  bar help
    --foo FOO foo help
```

The add_argument_group() method returns an argument group object which has an add_argument() method just like a regular ArgumentParser. When an argument is added to the group, the parser treats it just like a normal argument, but displays the argument in a separate group for help messages. The add_argument_group() method accepts title and description arguments which can be used to customize this display:

```python
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo

group1:
    group1 description
    foo  foo help

group2:
    group2 description
    --bar BAR bar help
```

Note that any arguments not in your user-defined groups will end up back in the usual “positional arguments” and “optional arguments” sections.

Mutual exclusion

ArgumentParser.add_mutually_exclusive_group(required=False)

Create a mutually exclusive group. argparse will make sure that only one of the arguments in the mutually exclusive group was present on the command line:

```python
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
```
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)

usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo

The add_mutually_exclusive_group() method also accepts a required argument, to indicate that at least one of the mutually exclusive arguments is required:

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')

usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required

Note that currently mutually exclusive argument groups do not support the title and description arguments of add_argument_group().

Parser defaults

ArgumentParser.set_defaults(**kwargs)
Most of the time, the attributes of the object returned by parse_args() will be fully determined by inspecting the command-line arguments and the argument actions. set_defaults() allows some additional attributes that are determined without any inspection of the command line to be added:

>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')

usage: PROG [-h] ['--foo' '--bar']
PROG: error: one of the arguments --foo --bar is required

Note that parser-level defaults always override argument-level defaults:

>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')

usage: PROG [-h] ['--foo' '--bar']

Parser-level defaults can be particularly useful when working with multiple parsers. See the add_subparsers() method for an example of this type.

ArgumentParser.get_default(dest)
Get the default value for a namespace attribute, as set by either add_argument() or by set_defaults():

>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')

usage: PROG [-h] ['--foo' '--bar']

'badger'
Printing help

In most typical applications, `parse_args()` will take care of formatting and printing any usage or error messages. However, several formatting methods are available:

**ArgumentParser**.`print_usage(file=None)`

Print a brief description of how the `ArgumentParser` should be invoked on the command line. If `file` is `None`, `sys.stdout` is assumed.

**ArgumentParser**.`print_help(file=None)`

Print a help message, including the program usage and information about the arguments registered with the `ArgumentParser`. If `file` is `None`, `sys.stdout` is assumed.

There are also variants of these methods that simply return a string instead of printing it:

**ArgumentParser**.`format_usage()`

Return a string containing a brief description of how the `ArgumentParser` should be invoked on the command line.

**ArgumentParser**.`format_help()`

Return a string containing a help message, including the program usage and information about the arguments registered with the `ArgumentParser`.

Partial parsing

**ArgumentParser**.`parse_known_args(args=None, namespace=None)`

Sometimes a script may only parse a few of the command-line arguments, passing the remaining arguments on to another script or program. In these cases, the `parse_known_args()` method can be useful. It works much like `parse_args()` except that it does not produce an error when extra arguments are present. Instead, it returns a two item tuple containing the populated namespace and the list of remaining argument strings.

```python
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

Customizing file parsing

**ArgumentParser**.`convert_arg_line_to_args(arg_line)`

Arguments that are read from a file (see the `fromfile_prefix_chars` keyword argument to the `ArgumentParser` constructor) are read one argument per line. `convert_arg_line_to_args()` can be overridden for fancier reading.

This method takes a single argument `arg_line` which is a string read from the argument file. It returns a list of arguments parsed from this string. The method is called once per line read from the argument file, in order.

A useful override of this method is one that treats each space-separated word as an argument:

```python
def convert_arg_line_to_args(self, arg_line):
    for arg in arg_line.split():
        if not arg.strip():
            continue
        yield arg
```
Exiting methods

ArgumentParser.exit(status=0, message=None)
This method terminates the program, exiting with the specified status and, if given, it prints a message before that.

ArgumentParser.error(message)
This method prints a usage message including the message to the standard error and terminates the program with a status code of 2.

16.4.6 Upgrading optparse code

Originally, the argparse module had attempted to maintain compatibility with optparse. However, optparse was difficult to extend transparently, particularly with the changes required to support the new nargs= specifiers and better usage messages. When most everything in optparse had either been copy-pasted over or monkey-patched, it no longer seemed practical to try to maintain the backwards compatibility.

A partial upgrade path from optparse to argparse:

• Replace all optparse.OptionParser.add_option() calls with ArgumentParser.add_argument() calls.

• Replace (options, args) = parser.parse_args() with args = parser.parse_args() and add additional ArgumentParser.add_argument() calls for the positional arguments. Keep in mind that what was previously called options, now in argparse context is called args.

• Replace callback actions and the callback_* keyword arguments with type or action arguments.

• Replace string names for type keyword arguments with the corresponding type objects (e.g. int, float, complex, etc).

• Replace optparse.Values with Namespace and optparse.OptionError and optparse.OptionValueError with ArgumentError.

• Replace strings with implicit arguments such as %default or %prog with the standard Python syntax to use dictionaries to format strings, that is, %(default)s and %(prog)s.

• Replace the OptionParser constructor version argument with a call to parser.add_argument('--version', action='version', version='<the version>')

16.5 optparse — Parser for command line options

Deprecated since version 3.2: The optparse module is deprecated and will not be developed further; development will continue with the argparse module. Source code: Lib/optparse.py

optparse is a more convenient, flexible, and powerful library for parsing command-line options than the old getopt module. optparse uses a more declarative style of command-line parsing: you create an instance of OptionParser, populate it with options, and parse the command line. optparse allows users to specify options in the conventional GNU/POSIX syntax, and additionally generates usage and help messages for you.

Here’s an example of using optparse in a simple script:

```python
from optparse import OptionParser
[...]
parsed = OptionParser()
parsed.add_option("-f", "--file", dest="filename",
                help="write report to FILE", metavar="FILE")
parsed.add_option("-q", "--quiet")
```
action="store_false", dest="verbose", default=True,
help="don’t print status messages to stdout")

(options, args) = parser.parse_args()

With these few lines of code, users of your script can now do the "usual thing" on the command-line, for example:

```bash
<yourscript> --file=outfile -q
```

As it parses the command line, optparse sets attributes of the options object returned by parse_args() based on user-supplied command-line values. When parse_args() returns from parsing this command line, options.filename will be "outfile" and options.verbose will be False. optparse supports both long and short options, allows short options to be merged together, and allows options to be associated with their arguments in a variety of ways. Thus, the following command lines are all equivalent to the above example:

```bash
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -f outfile
<yourscript> -qf outfile
```

Additionally, users can run one of

```bash
<yourscript> -h
<yourscript> --help
```

and optparse will print out a brief summary of your script’s options:

```
Usage: <yourscript> [options]

Options:
-h, --help show this help message and exit
-f FILE, --file=FILE write report to FILE
-q, --quiet don’t print status messages to stdout
```

where the value of yourscript is determined at runtime (normally from sys.argv[0]).

### 16.5.1 Background

optparse was explicitly designed to encourage the creation of programs with straightforward, conventional command-line interfaces. To that end, it supports only the most common command-line syntax and semantics conventionally used under Unix. If you are unfamiliar with these conventions, read this section to acquaint yourself with them.

#### Terminology

**argument** a string entered on the command-line, and passed by the shell to `excl()` or `execv()`. In Python, arguments are elements of `sys.argv[1:]` (sys.argv[0] is the name of the program being executed). Unix shells also use the term "word".

It is occasionally desirable to substitute an argument list other than `sys.argv[1:]`, so you should read “argument” as “an element of `sys.argv[1:]`, or of some other list provided as a substitute for `sys.argv[1:]`”.

**option** an argument used to supply extra information to guide or customize the execution of a program. There are many different syntaxes for options; the traditional Unix syntax is a hyphen (“-“) followed by a single letter, e.g. `-x` or `-F`. Also, traditional Unix syntax allows multiple options to be merged into a single argument, e.g. `-x` `-F` is equivalent to `-xF`. The GNU project introduced -- followed by a series of hyphen-separated words, e.g. `--file` or `--dry-run`. These are the only two option syntaxes provided by optparse.

Some other option syntaxes that the world has seen include:

- a hyphen followed by a few letters, e.g. `-pf` (this is not the same as multiple options merged into a single argument)
• a hyphen followed by a whole word, e.g. `-file` (this is technically equivalent to the previous syntax, but they aren’t usually seen in the same program)
• a plus sign followed by a single letter, or a few letters, or a word, e.g. `+f, +rgb`
• a slash followed by a letter, or a few letters, or a word, e.g. `/f, /file`

These option syntaxes are not supported by `optparse`, and they never will be. This is deliberate: the first three are non-standard on any environment, and the last only makes sense if you’re exclusively targeting VMS, MS-DOS, and/or Windows.

**option argument** an argument that follows an option, is closely associated with that option, and is consumed from the argument list when that option is. With `optparse`, option arguments may either be in a separate argument from their option:

```
-f foo
--file foo
```

or included in the same argument:

```
-ffoo
--file=foo
```

Typically, a given option either takes an argument or it doesn’t. Lots of people want an “optional option arguments” feature, meaning that some options will take an argument if they see it, and won’t if they don’t. This is somewhat controversial, because it makes parsing ambiguous: if `-a` takes an optional argument and `-b` is another option entirely, how do we interpret `-ab`? Because of this ambiguity, `optparse` does not support this feature.

**positional argument** something leftover in the argument list after options have been parsed, i.e. after options and their arguments have been parsed and removed from the argument list.

**required option** an option that must be supplied on the command-line; note that the phrase “required option” is self-contradictory in English. `optparse` doesn’t prevent you from implementing required options, but doesn’t give you much help at it either.

For example, consider this hypothetical command-line:

```
prog -v --report report.txt foo bar
```

`-v` and `--report` are both options. Assuming that `--report` takes one argument, `report.txt` is an option argument. `foo` and `bar` are positional arguments.

### What are options for?

Options are used to provide extra information to tune or customize the execution of a program. In case it wasn’t clear, options are usually *optional*. A program should be able to run just fine with no options whatsoever. (Pick a random program from the Unix or GNU toolsets. Can it run without any options at all and still make sense? The main exceptions are `find, tar, and dd—all of which are mutant oddballs that have been rightly criticized for their non-standard syntax and confusing interfaces."

Lots of people want their programs to have “required options”. Think about it. If it’s required, then it’s *not* optional! If there is a piece of information that your program absolutely requires in order to run successfully, that’s what positional arguments are for.

As an example of good command-line interface design, consider the humble `cp` utility, for copying files. It doesn’t make much sense to try to copy files without supplying a destination and at least one source. Hence, `cp` fails if you run it with no arguments. However, it has a flexible, useful syntax that does not require any options at all:

```
cp SOURCE DEST
```

You can get pretty far with just that. Most `cp` implementations provide a bunch of options to tweak exactly how the files are copied: you can preserve mode and modification time, avoid following symlinks, ask before clobbering existing files, etc. But none of this distracts from the core mission of `cp`, which is to copy either one file to another, or several files to another directory.
What are positional arguments for?

Positional arguments are for those pieces of information that your program absolutely, positively requires to run.

A good user interface should have as few absolute requirements as possible. If your program requires 17 distinct pieces of information in order to run successfully, it doesn’t much matter how you get that information from the user—most people will give up and walk away before they successfully run the program. This applies whether the user interface is a command-line, a configuration file, or a GUI: if you make that many demands on your users, most of them will simply give up.

In short, try to minimize the amount of information that users are absolutely required to supply—use sensible defaults whenever possible. Of course, you also want to make your programs reasonably flexible. That’s what options are for. Again, it doesn’t matter if they are entries in a config file, widgets in the “Preferences” dialog of a GUI, or command-line options—the more options you implement, the more flexible your program is, and the more complicated its implementation becomes. Too much flexibility has drawbacks as well, of course; too many options can overwhelm users and make your code much harder to maintain.

16.5.2 Tutorial

While optparse is quite flexible and powerful, it’s also straightforward to use in most cases. This section covers the code patterns that are common to any optparse-based program.

First, you need to import the OptionParser class; then, early in the main program, create an OptionParser instance:

```python
from optparse import OptionParser
parser = OptionParser()
```

Then you can start defining options. The basic syntax is:

```python
parser.add_option(opt_str, ..., 
                   attr=value, ...)
```

Each option has one or more option strings, such as `-f` or `--file`, and several option attributes that tell optparse what to expect and what to do when it encounters that option on the command line.

Typically, each option will have one short option string and one long option string, e.g.:

```python
parser.add_option("-f", "--file", ...)
```

You’re free to define as many short option strings and as many long option strings as you like (including zero), as long as there is at least one option string overall.

The option strings passed to `OptionParser.add_option()` are effectively labels for the option defined by that call. For brevity, we will frequently refer to encountering an option on the command line; in reality, optparse encounters option strings and looks up options from them.

Once all of your options are defined, instruct optparse to parse your program’s command line:

```python
(options, args) = parser.parse_args()
```

(If you like, you can pass a custom argument list to `parse_args()`, but that’s rarely necessary: by default it uses `sys.argv[1:]`.)

`parse_args()` returns two values:

- `options`, an object containing values for all of your options—e.g. if `--file` takes a single string argument, then `options.file` will be the filename supplied by the user, or `None` if the user did not supply that option
- `args`, the list of positional arguments leftover after parsing options

This tutorial section only covers the four most important option attributes: `action`, `type`, `dest` (destination), and `help`. Of these, `action` is the most fundamental.
Understanding option actions

Actions tell *optparse* what to do when it encounters an option on the command line. There is a fixed set of actions hard-coded into *optparse*; adding new actions is an advanced topic covered in section *Extending optparse*. Most actions tell *optparse* to store a value in some variable—for example, take a string from the command line and store it in an attribute of *options*.

If you don’t specify an option action, *optparse* defaults to *store*.

The store action

The most common option action is *store*, which tells *optparse* to take the next argument (or the remainder of the current argument), ensure that it is of the correct type, and store it to your chosen destination.

For example:

```python
parser.add_option("-f", "--file",
    action="store", type="string", dest="filename")
```

Now let’s make up a fake command line and ask *optparse* to parse it:

```python
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

When *optparse* sees the option string `-f`, it consumes the next argument, `foo.txt`, and stores it in *options.filename*. So, after this call to *parse_args()*,*options.filename* is "foo.txt".

Some other option types supported by *optparse* are *int* and *float*. Here’s an option that expects an integer argument:

```python
parser.add_option("-n", type="int", dest="num")
```

Note that this option has no long option string, which is perfectly acceptable. Also, there’s no explicit action, since the default is *store*.

Let’s parse another fake command-line. This time, we’ll jam the option argument right up against the option: since `-n42` (one argument) is equivalent to `-n 42` (two arguments), the code

```python
(options, args) = parser.parse_args(["-n42"])
print(options.num)
```

will print 42.

If you don’t specify a type, *optparse* assumes *string*. Combined with the fact that the default action is *store*, that means our first example can be a lot shorter:

```python
parser.add_option("-f", "--file", dest="filename")
```

If you don’t supply a destination, *optparse* figures out a sensible default from the option strings: if the first long option string is --foo-bar, then the default destination is foo_bar. If there are no long option strings, *optparse* looks at the first short option string: the default destination for -f is f.

*optparse* also includes the built-in *complex* type. Adding types is covered in section *Extending optparse*.

Handling boolean (flag) options

Flag options—set a variable to true or false when a particular option is seen—are quite common. *optparse* supports them with two separate actions, *store_true* and *store_false*. For example, you might have a *verbose* flag that is turned on with `-v` and off with `-q`:

```python
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```
Here we have two different options with the same destination, which is perfectly OK. (It just means you have to be a bit careful when setting default values—see below.)

When `optparse` encounters `-v` on the command line, it sets `options.verbose` to `True`; when it encounters `-q`, `options.verbose` is set to `False`.

**Other actions**

Some other actions supported by `optparse` are:

- "store_const" store a constant value
- "append" append this option’s argument to a list
- "count" increment a counter by one
- "callback" call a specified function

These are covered in section *Reference Guide*, Reference Guide and section *Option Callbacks*.

**Default values**

All of the above examples involve setting some variable (the “destination”) when certain command-line options are seen. What happens if those options are never seen? Since we didn’t supply any defaults, they are all set to `None`. This is usually fine, but sometimes you want more control. `optparse` lets you supply a default value for each destination, which is assigned before the command line is parsed.

First, consider the verbose/quiet example. If we want `optparse` to set `verbose` to `True` unless `-q` is seen, then we can do this:

```python
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

Since default values apply to the destination rather than to any particular option, and these two options happen to have the same destination, this is exactly equivalent:

```python
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Consider this:

```python
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Again, the default value for `verbose` will be `True`: the last default value supplied for any particular destination is the one that counts.

A clearer way to specify default values is the `set_defaults()` method of `OptionParser`, which you can call at any time before calling `parse_args()`:

```python
parser.set_defaults( verbose=True )
```

```python
(options, args) = parser.parse_args()
```

As before, the last value specified for a given option destination is the one that counts. For clarity, try to use one method or the other of setting default values, not both.

**Generating help**

`optparse`’s ability to generate help and usage text automatically is useful for creating user-friendly command-line interfaces. All you have to do is supply a `help` value for each option, and optionally a short usage message for your whole program. Here’s an `OptionParser` populated with user-friendly (documented) options:
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
    action="store_true", dest="verbose", default=True,
    help="make lots of noise [default]"
)
parser.add_option("-q", "--quiet",
    action="store_false", dest="verbose",
    help="be vewwy quiet (I’m hunting wabbits)"
)
parser.add_option("-f", "--filename",
    metavar="FILE", help="write output to FILE"
)
parser.add_option("-m", "--mode",
    default="intermediate",
    help="interaction mode: novice, intermediate, "
    "or expert [default: %default]"
)
If optparse encounters either --help on the command-line, or if you just call parser.print_help(), it prints the following to standard output:

Usage: <yourscript> [options] arg1 arg2

Options:
-h, --help       show this help message and exit
-v, --verbose    make lots of noise [default]
-q, --quiet      be vewwy quiet (I’m hunting wabbits)
-f FILE, --filename=FILE
                 write output to FILE
-m MODE, --mode=MODE interaction mode: novice, intermediate, or
                 expert [default: intermediate]

(If the help output is triggered by a help option, optparse exits after printing the help text.)

There’s a lot going on here to help optparse generate the best possible help message:

• the script defines its own usage message:

usage = "usage: %prog [options] arg1 arg2"

optparse expands %prog in the usage string to the name of the current program, i.e.
on.path.basename(sys.argv[0]). The expanded string is then printed before the detailed option
help.

If you don’t supply a usage string, optparse uses a bland but sensible default: "Usage: %prog
[options]", which is fine if your script doesn’t take any positional arguments.

• every option defines a help string, and doesn’t worry about line-wrapping— optparse takes care of
wrapping lines and making the help output look good.

• options that take a value indicate this fact in their automatically-generated help message, e.g. for the "mode"
option:

-m MODE, --mode=MODE

Here, “MODE” is called the meta-variable: it stands for the argument that the user is expected to supply to

-m/--mode. By default, optparse converts the destination variable name to uppercase and uses that for
the meta-variable. Sometimes, that’s not what you want—for example, the --filename option explicitly
sets metavar="FILE", resulting in this automatically-generated option description:

-f FILE, --filename=FILE

This is important for more than just saving space, though: the manually written help text uses the meta-
variable FILE to clue the user in that there’s a connection between the semi-formal syntax -f FILE and
the informal semantic description “write output to FILE”. This is a simple but effective way to make your
help text a lot clearer and more useful for end users.
• options that have a default value can include %default in the help string—optparse will replace it with str() of the option’s default value. If an option has no default value (or the default value is None), %default expands to none.

Grouping Options

When dealing with many options, it is convenient to group these options for better help output. An OptionParser can contain several option groups, each of which can contain several options.

An option group is obtained using the class OptionGroup:

class optparse.OptionGroup (parser, title, description=None)

where

• parser is the OptionParser instance the group will be inserted in to
• title is the group title
• description, optional, is a long description of the group

OptionGroup inherits from OptionContainer (like OptionParser) and so the add_option() method can be used to add an option to the group.

Once all the options are declared, using the OptionParser method add_option_group() the group is added to the previously defined parser.

Continuing with the parser defined in the previous section, adding an OptionGroup to a parser is easy:

group = OptionGroup(parser, "Dangerous Options", 
"Caution: use these options at your own risk. " 
"It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

This would result in the following help output:

Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help    show this help message and exit
  -v, --verbose make lots of noise [default]
  -q, --quiet   be vewwy quiet (I’m hunting wabbits)
  -f FILE, --filename=FILE write output to FILE
  -m MODE, --mode=MODE interaction mode: novice, intermediate, or expert [default: intermediate]

Dangerous Options:
  Caution: use these options at your own risk. It is believed that some of them bite.
  -g    Group option.

A bit more complete example might involve using more than one group: still extending the previous example:

group = OptionGroup(parser, "Dangerous Options", 
"Caution: use these options at your own risk. " 
"It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true", 
help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
    help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every action done")
parser.add_option_group(group)

that results in the following output:

Usage: <yourscript> [options] arg1 arg2

Options:
-h, --help          show this help message and exit
-v, --verbose       make lots of noise [default]
-q, --quiet         be vewwy quiet (I’m hunting wabbits)
-f FILE, --filename=FILE
    write output to FILE
-m MODE, --mode=MODE interaction mode: novice, intermediate, or expert
                       [default: intermediate]

Dangerous Options:
    Caution: use these options at your own risk. It is believed that some
    of them bite.
    -g Group option.

Debug Options:
-d, --debug         Print debug information
-s, --sql           Print all SQL statements executed
-e Print every action done

Another interesting method, in particular when working programmatically with option groups is:

OptionParser.get_option_group(opt_str)
    Return the OptionGroup to which the short or long option string opt_str (e.g. ‘-o’ or ‘--option’) 
    belongs. If there’s no such OptionGroup, return None.

Printing a version string

Similar to the brief usage string, optparse can also print a version string for your program. You have to supply 
the string as the version argument to OptionParser:

parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")

%prog is expanded just like it is in usage. Apart from that, version can contain anything you like. When you 
supply it, optparse automatically adds a --version option to your parser. If it encounters this option on the 
command line, it expands your version string (by replacing %prog), prints it to stdout, and exits.

For example, if your script is called /usr/bin/foo:

$ /usr/bin/foo --version
foo 1.0

The following two methods can be used to print and get the version string:

OptionParser.print_version(file=None)
    Print the version message for the current program (self.version) to file (default stdout). As with 
    print_usage(), any occurrence of %prog in self.version is replaced with the name of the current 
    program. Does nothing if self.version is empty or undefined.

OptionParser.get_version()
    Same as print_version() but returns the version string instead of printing it.
How `optparse` handles errors

There are two broad classes of errors that `optparse` has to worry about: programmer errors and user errors. Programmer errors are usually erroneous calls to `OptionParser.add_option()`, e.g. invalid option strings, unknown option attributes, missing option attributes, etc. These are dealt with in the usual way: raise an exception (either `optparse.OptionError` or `TypeError`) and let the program crash.

Handling user errors is much more important, since they are guaranteed to happen no matter how stable your code is. `optparse` can automatically detect some user errors, such as bad option arguments (passing `-n 4x` where `-n` takes an integer argument), missing arguments (`-n` at the end of the command line, where `-n` takes an argument of any type). Also, you can call `OptionParser.error()` to signal an application-defined error condition:

```python
(options, args) = parser.parse_args()
[...]
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

In either case, `optparse` handles the error the same way: it prints the program’s usage message and an error message to standard error and exits with error status 2.

Consider the first example above, where the user passes `4x` to an option that takes an integer:

```bash
$ /usr/bin/foo -n 4x
Usage: foo [options]
```

foo: error: option -n: invalid integer value: '4x'

Or, where the user fails to pass a value at all:

```bash
$ /usr/bin/foo -n
Usage: foo [options]
```

foo: error: -n option requires an argument

`optparse`-generated error messages take care always to mention the option involved in the error; be sure to do the same when calling `OptionParser.error()` from your application code.

If `optparse`’s default error-handling behaviour does not suit your needs, you’ll need to subclass `OptionParser` and override its `exit()` and/or `error()` methods.

Putting it all together

Here’s what `optparse`-based scripts usually look like:

```python
from optparse import OptionParser
[...]
def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
        help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
        action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
        action="store_false", dest="verbose")
    [...]
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print("reading %s..." % options.filename)
```
if __name__ == "__main__":
    main()

16.5.3 Reference Guide

Creating the parser

The first step in using optparse is to create an OptionParser instance.

class optparse.OptionParser(...)

The OptionParser constructor has no required arguments, but a number of optional keyword arguments. You should always pass them as keyword arguments, i.e. do not rely on the order in which the arguments are declared.

usage (default: "%prog [options]") The usage summary to print when your program is run incorrectly or with a help option. When optparse prints the usage string, it expands %prog to os.path.basename(sys.argv[0]) (or to prog if you passed that keyword argument). To suppress a usage message, pass the special value optparse.SUPPRESS_USAGE.

option_list (default: []) A list of Option objects to populate the parser with. The options in option_list are added after any options in standard_option_list (a class attribute that may be set by OptionParser subclasses), but before any version or help options. Deprecated; use add_option() after creating the parser instead.

option_class (default: optparse.Option) Class to use when adding options to the parser in add_option().

version (default: None) A version string to print when the user supplies a version option. If you supply a true value for version, optparse automatically adds a version option with the single option string --version. The substring %prog is expanded the same as for usage.

conflict_handler (default: "error") Specifies what to do when options with conflicting option strings are added to the parser; see section Conflicts between options.

description (default: None) A paragraph of text giving a brief overview of your program. optparse reformats this paragraph to fit the current terminal width and prints it when the user requests help (after usage, but before the list of options).

formatter (default: a new IndentedHelpFormatter) An instance of optparse.HelpFormatter that will be used for printing help text. optparse provides two concrete classes for this purpose: IndentedHelpFormatter and TitledHelpFormatter.

add_help_option (default: True) If true, optparse will add a help option (with option strings -h and --help) to the parser.

prog The string to use when expanding %prog in usage and version instead of os.path.basename(sys.argv[0]).

epilog (default: None) A paragraph of help text to print after the option help.

Populating the parser

There are several ways to populate the parser with options. The preferred way is by using OptionParser.add_option(), as shown in section Tutorial. add_option() can be called in one of two ways:

- pass it an Option instance (as returned by make_option())
- pass it any combination of positional and keyword arguments that are acceptable to make_option() (i.e., to the Option constructor), and it will create the Option instance for you
The other alternative is to pass a list of pre-constructed Option instances to the OptionParser constructor, as in:

```python
def make_option(*opt_str, attr=value, ...):
    pass
```

```python
option_list = [
    make_option("-f", "--filename",
               action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
               action="store_false", dest="verbose"),
]
parser = OptionParser(option_list=option_list)
```

(make_option() is a factory function for creating Option instances; currently it is an alias for the Option constructor. A future version of optparse may split Option into several classes, and make_option() will pick the right class to instantiate. Do not instantiate Option directly.)

### Defining options

Each Option instance represents a set of synonymous command-line option strings, e.g. `-f` and `--file`. You can specify any number of short or long option strings, but you must specify at least one overall option string.

The canonical way to create an Option instance is with the `add_option()` method of `OptionParser`.

```python
parser.add_option("-f", attr=value, ...)
```

And to define an option with only a long option string:

```python
parser.add_option("--foo", attr=value, ...)
```

The keyword arguments define attributes of the new Option object. The most important option attribute is `action`, and it largely determines which other attributes are relevant or required. If you pass irrelevant option attributes, or fail to pass required ones, `optparse` raises an `OptionError` exception explaining your mistake.

An option’s `action` determines what `optparse` does when it encounters this option on the command-line. The standard option actions hard-coded into `optparse` are:

- "store" store this option’s argument (default)
- "store_const" store a constant value
- "store_true" store a true value
- "store_false" store a false value
- "append" append this option’s argument to a list
- "append_const" append a constant value to a list
- "count" increment a counter by one
- "callback" call a specified function
- "help" print a usage message including all options and the documentation for them

(If you don’t supply an action, the default is "store". For this action, you may also supply `type` and `dest` option attributes; see Standard option actions.)

As you can see, most actions involve storing or updating a value somewhere. `optparse` always creates a special object for this, conventionally called `options` (it happens to be an instance of `optparse.Values`). Option arguments (and various other values) are stored as attributes of this object, according to the `dest` (destination) option attribute.

For example, when you call...
parser.parse_args()

one of the first things optparse does is create the options object:
options = Values()

If one of the options in this parser is defined with
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
and the command-line being parsed includes any of the following:
-f foo
--file=foo
--file foo

then optparse, on seeing this option, will do the equivalent of
options.filename = "foo"

The type and dest option attributes are almost as important as action, but action is the only one that makes sense for all options.

Option attributes

The following option attributes may be passed as keyword arguments to OptionParser.add_option(). If you pass an option attribute that is not relevant to a particular option, or fail to pass a required option attribute, optparse raises OptionError.

Option. action
  (default: "store")
  Determines optparse's behaviour when this option is seen on the command line; the available options are documented here.

Option. type
  (default: "string")
  The argument type expected by this option (e.g., "string" or "int"); the available option types are documented here.

Option. dest
  (default: derived from option strings)
  If the option’s action implies writing or modifying a value somewhere, this tells optparse where to write it: dest names an attribute of the options object that optparse builds as it parses the command line.

Option. default
  The value to use for this option’s destination if the option is not seen on the command line. See also OptionParser.set_defaults().

Option. nargs
  (default: 1)
  How many arguments of type type should be consumed when this option is seen. If > 1, optparse will store a tuple of values to dest.

Option. const
  For actions that store a constant value, the constant value to store.

Option. choices
  For options of type "choice", the list of strings the user may choose from.

Option. callback
  For options with action "callback", the callable to call when this option is seen. See section Option Callbacks for detail on the arguments passed to the callable.
Option. **callback_args**
Option. **callback_kwargs**
Additional positional and keyword arguments to pass to callback after the four standard callback arguments.

Option. **help**
Help text to print for this option when listing all available options after the user supplies a help option (such as --help). If no help text is supplied, the option will be listed without help text. To hide this option, use the special value *optparse.SUPPRESS_HELP*.

Option. **metavar**  
(default: derived from option strings)  
Stand-in for the option argument(s) to use when printing help text. See section *Tutorial* for an example.

**Standard option actions**

The various option actions all have slightly different requirements and effects. Most actions have several relevant option attributes which you may specify to guide *optparse*’s behaviour; a few have required attributes, which you must specify for any option using that action.

- **"store"** [relevant: *type*, *dest*, *nargs*, *choices*]
  The option must be followed by an argument, which is converted to a value according to *type* and stored in *dest*. If *nargs* > 1, multiple arguments will be consumed from the command line; all will be converted according to *type* and stored to *dest* as a tuple. See the *Standard option types* section.
  If *choices* is supplied (a list or tuple of strings), the type defaults to "choice".
  If *type* is not supplied, it defaults to "string".
  If *dest* is not supplied, *optparse* derives a destination from the first long option string (e.g., --foo-bar implies foo_bar). If there are no long option strings, *optparse* derives a destination from the first short option string (e.g., -f implies f).
  Example:
  ```python
  parser.add_option("-f")
  parser.add_option("-p", type="float", nargs=3, dest="point")
  ```
  As it parses the command line
  ```bash
  -f foo.txt -p 1 -3.5 4 -fbar.txt
  ```
  *optparse* will set
  ```python
  options.f = "foo.txt"
  options.point = (1.0, -3.5, 4.0)
  options.f = "bar.txt"
  ```

- **"store_const"** [required: *const*; relevant: *dest*]
  The value *const* is stored in *dest*.
  Example:
  ```python
  parser.add_option("-q", "--quiet",
                   action="store_const", const=0, dest="verbose")
  parser.add_option("-v", "--verbose",
                   action="store_const", const=1, dest="verbose")
  parser.add_option("--noisy",
                   action="store_const", const=2, dest="verbose")
  ```
  If --noisy is seen, *optparse* will set
  ```python
  options.verbose = 2
  ```
• "store_true" [relevant: dest]
   A special case of "store_const" that stores a true value to dest.

• "store_false" [relevant: dest]
   Like "store_true", but stores a false value.
   Example:
   ```python
   parser.add_option("--clobber", action="store_true", dest="clobber")
   parser.add_option("--no-clobber", action="store_false", dest="clobber")
   ```

• "append" [relevant: type, dest, nargs, choices]
   The option must be followed by an argument, which is appended to the list in dest. If no default value for dest is supplied, an empty list is automatically created when optparse first encounters this option on the command-line. If nargs > 1, multiple arguments are consumed, and a tuple of length nargs is appended to dest.
   The defaults for type and dest are the same as for the "store" action.
   Example:
   ```python
   parser.add_option("-t", "--tracks", action="append", type="int")
   ```
   If -t3 is seen on the command-line, optparse does the equivalent of:
   ```python
   options.tracks = []
   options.tracks.append(int("3"))
   ```
   If, a little later on, --tracks=4 is seen, it does:
   ```python
   options.tracks.append(int("4"))
   ```
   The append action calls the append method on the current value of the option. This means that any default value specified must have an append method. It also means that if the default value is non-empty, the default elements will be present in the parsed value for the option, with any values from the command line appended after those default values:
   ```python
   >>> parser.add_option("--files", action="append", default=['~/.mypkg/defaults'])
   >>> opts, args = parser.parse_args(['--files', 'overrides.mypkg'])
   >>> opts.files
   [~/.mypkg/defaults', 'overrides.mypkg']
   ```

• "append_const" [required: const; relevant: dest]
   Like "store_const", but the value const is appended to dest; as with "append", dest defaults to None, and an empty list is automatically created the first time the option is encountered.

• "count" [relevant: dest]
   Increment the integer stored at dest. If no default value is supplied, dest is set to zero before being incremented the first time.
   Example:
   ```python
   parser.add_option("-v", action="count", dest="verbosity")
   ```
   The first time -v is seen on the command line, optparse does the equivalent of:
   ```python
   options.verbosity = 0
   options.verbosity += 1
   ```
   Every subsequent occurrence of -v results in
   ```python
   options.verbosity += 1
   ```

• "callback" [required: callback; relevant: type, nargs, callback_args, callback_kwargs]
   Call the function specified by callback, which is called as
func(option, opt_str, value, parser, *args, **kwargs)

See section Option Callbacks for more detail.

• "help"

Prints a complete help message for all the options in the current option parser. The help message is con-structed from the usage string passed to OptionParser’s constructor and the help string passed to every option.

If no help string is supplied for an option, it will still be listed in the help message. To omit an option entirely, use the special value optparse.SUPPRESS_HELP.

optparse automatically adds a help option to all OptionParsers, so you do not normally need to create one.

Example:

from optparse import OptionParser, SUPPRESS_HELP

# usually, a help option is added automatically, but that can # be suppressed using the add_help_option argument
parser = OptionParser(add_help_option=False)

parser.add_option("-h", "--help", action="help")
parser.add_option("-v", action="store_true", dest="verbose", help="Be moderately verbose")
parser.add_option("--file", dest="filename", help="Input file to read data from")
parser.add_option("--secret", help=SUPPRESS_HELP)

If optparse sees either -h or --help on the command line, it will print something like the following help message to stdout (assuming sys.argv[0] is "foo.py"):

Usage: foo.py [options]

Options:
  -h, --help    Show this help message and exit
  -v            Be moderately verbose
  --file=FILENAME  Input file to read data from

After printing the help message, optparse terminates your process with sys.exit(0).

• "version"

Prints the version number supplied to the OptionParser to stdout and exits. The version number is actually formatted and printed by the print_version() method of OptionParser. Generally only relevant if the version argument is supplied to the OptionParser constructor. As with help options, you will rarely create version options, since optparse automatically adds them when needed.

Standard option types

optparse has five built-in option types: "string", "int", "choice", "float" and "complex". If you need to add new option types, see section Extending optparse.

Arguments to string options are not checked or converted in any way: the text on the command line is stored in the destination (or passed to the callback) as-is.

Integer arguments (type "int") are parsed as follows:

• if the number starts with 0x, it is parsed as a hexadecimal number
• if the number starts with 0, it is parsed as an octal number
• if the number starts with 0b, it is parsed as a binary number
• otherwise, the number is parsed as a decimal number

The conversion is done by calling `int()` with the appropriate base (2, 8, 10, or 16). If this fails, so will `optparse`, although with a more useful error message.

"float" and "complex" option arguments are converted directly with `float()` and `complex()`, with similar error-handling.

"choice" options are a subtype of "string" options. The `choices` option attribute (a sequence of strings) defines the set of allowed option arguments. `optparse.check_choice()` compares user-supplied option arguments against this master list and raises `OptionValueError` if an invalid string is given.

### Parsing arguments

The whole point of creating and populating an OptionParser is to call its `parse_args()` method:

```python
(options, args) = parser.parse_args(args=None, values=None)
```

where the input parameters are

- **args** the list of arguments to process (default: `sys.argv[1:]`)
- **values** a `optparse.Values` object to store option arguments in (default: a new instance of `Values`) – if you give an existing object, the option defaults will not be initialized on it

and the return values are

- **options** the same object that was passed in as `values`, or the `optparse.Values` instance created by `optparse`
- **args** the leftover positional arguments after all options have been processed

The most common usage is to supply neither keyword argument. If you supply `values`, it will be modified with repeated `setattr()` calls (roughly one for every option argument stored to an option destination) and returned by `parse_args()`.

If `parse_args()` encounters any errors in the argument list, it calls the OptionParser’s `error()` method with an appropriate end-user error message. This ultimately terminates your process with an exit status of 2 (the traditional Unix exit status for command-line errors).

### Querying and manipulating your option parser

The default behavior of the option parser can be customized slightly, and you can also poke around your option parser and see what’s there. `optparse` provides several methods to help you out:

**OptionParser.**

- **disable_interspersed_args()**
  
  Set parsing to stop on the first non-option. For example, if `-a` and `-b` are both simple options that take no arguments, `optparse` normally accepts this syntax:

  ```
  prog -a arg1 -b arg2
  ```

  and treats it as equivalent to

  ```
  prog -a -b arg1 arg2
  ```

  To disable this feature, call `disable_interspersed_args()`. This restores traditional Unix syntax, where option parsing stops with the first non-option argument.

  Use this if you have a command processor which runs another command which has options of its own and you want to make sure these options don’t get confused. For example, each command might have a different set of options.

- **enable_interspersed_args()**
  
  Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior.
OptionParser.get_option(opt_str)
    Returns the Option instance with the option string opt_str, or None if no options have that option string.

OptionParser.has_option(opt_str)
    Return true if the OptionParser has an option with option string opt_str (e.g., -q or --verbose).

OptionParser.remove_option(opt_str)
    If the OptionParser has an option corresponding to opt_str, that option is removed. If that option
    provided any other option strings, all of those option strings become invalid. If opt_str does not occur in
    any option belonging to this OptionParser, raises ValueError.

Conflicts between options

If you’re not careful, it’s easy to define options with conflicting option strings:

    parser.add_option("-n", "--dry-run", ...)
    [...]  # (This is particularly true if you’ve defined your own OptionParser subclass with some standard options.)
    parser.add_option("-n", "--noisy", ...)

Every time you add an option, optparse checks for conflicts with existing options. If it finds any, it invokes the
current conflict-handling mechanism. You can set the conflict-handling mechanism either in the constructor:

    parser = OptionParser(..., conflict_handler=handler)
    or with a separate call:
    parser.set_conflict_handler(handler)

The available conflict handlers are:

    "error" (default) assume option conflicts are a programming error and raise
    OptionConflictError
    "resolve" resolve option conflicts intelligently (see below)

As an example, let’s define an OptionParser that resolves conflicts intelligently and add conflicting options to
it:

    parser = OptionParser(conflict_handler="resolve")
    parser.add_option("-n", "--dry-run", ..., help="do no harm")
    parser.add_option("-n", "--noisy", ..., help="be noisy")

At this point, optparse detects that a previously-added option is already using the -n option string. Since
conflict_handler is "resolve", it resolves the situation by removing -n from the earlier option’s list of
option strings. Now --dry-run is the only way for the user to activate that option. If the user asks for help, the
help message will reflect that:

Options:
    --dry-run do no harm
    [...]  # (This is particularly true if you’ve defined your own OptionParser subclass with some standard options.)
    -n, --noisy be noisy

It’s possible to whittle away the option strings for a previously-added option until there are none left, and the
user has no way of invoking that option from the command-line. In that case, optparse removes that option
completely, so it doesn’t show up in help text or anywhere else. Carrying on with our existing OptionParser:

    parser.add_option("--dry-run", ..., help="new dry-run option")

At this point, the original -n/--dry-run option is no longer accessible, so optparse removes it, leaving this
help text:

Options:
    [...]  # (This is particularly true if you’ve defined your own OptionParser subclass with some standard options.)
    -n, --noisy be noisy
    --dry-run new dry-run option

Chapter 16. Generic Operating System Services
Cleanup

OptionParser instances have several cyclic references. This should not be a problem for Python’s garbage collector, but you may wish to break the cyclic references explicitly by calling `destroy()` on your OptionParser once you are done with it. This is particularly useful in long-running applications where large object graphs are reachable from your OptionParser.

Other methods

OptionParser supports several other public methods:

OptionParser. `set_usage(usage)`
Set the usage string according to the rules described above for the usage constructor keyword argument. Passing `None` sets the default usage string; use `optparse.SUPPRESS_USAGE` to suppress a usage message.

OptionParser. `print_usage(file=None)`
Print the usage message for the current program (`self.usage`) to `file` (default stdout). Any occurrence of the string `%prog` in `self.usage` is replaced with the name of the current program. Does nothing if `self.usage` is empty or not defined.

OptionParser. `get_usage()`
Same as `print_usage()` but returns the usage string instead of printing it.

OptionParser. `set_defaults(dest=value, ...)`
Set default values for several option destinations at once. Using `set_defaults()` is the preferred way to set default values for options, since multiple options can share the same destination. For example, if several “mode” options all set the same destination, any one of them can set the default, and the last one wins:

```python
parser.add_option("--advanced", action="store_const",
    dest="mode", const="advanced",
    default="novice")  # overridden below
parser.add_option("--novice", action="store_const",
    dest="mode", const="novice",
    default="advanced")  # overrides above setting
```

To avoid this confusion, use `set_defaults()`:

```python
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
    dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
    dest="mode", const="novice")
```

16.5.4 Option Callbacks

When `optparse`'s built-in actions and types aren’t quite enough for your needs, you have two choices: extend `optparse` or define a callback option. Extending `optparse` is more general, but overkill for a lot of simple cases. Quite often a simple callback is all you need.

There are two steps to defining a callback option:

- define the option itself using the "callback" action
- write the callback; this is a function (or method) that takes at least four arguments, as described below
Defining a callback option

As always, the easiest way to define a callback option is by using the `OptionParser.add_option()` method. Apart from `action`, the only option attribute you must specify is `callback`, the function to call:

```python
parser.add_option("-c", action="callback", callback=my_callback)
```

callback is a function (or other callable object), so you must have already defined `my_callback()` when you create this callback option. In this simple case, `optparse` doesn’t even know if `-c` takes any arguments, which usually means that the option takes no arguments—the mere presence of `-c` on the command-line is all it needs to know. In some circumstances, though, you might want your callback to consume an arbitrary number of command-line arguments. This is where writing callbacks gets tricky; it’s covered later in this section.

`optparse` always passes four particular arguments to your callback, and it will only pass additional arguments if you specify them via `callback_args` and `callback_kwargs`. Thus, the minimal callback function signature is:

```python
def my_callback(option, opt, value, parser):
```

The four arguments to a callback are described below.

There are several other option attributes that you can supply when you define a callback option:

- `type` has its usual meaning: as with the "store" or "append" actions, it instructs `optparse` to consume one argument and convert it to `type`. Rather than storing the converted value(s) anywhere, though, `optparse` passes it to your callback function.
- `nargs` also has its usual meaning: if it is supplied and > 1, `optparse` will consume `nargs` arguments, each of which must be convertible to `type`. It then passes a tuple of converted values to your callback.
- `callback_args` a tuple of extra positional arguments to pass to the callback
- `callback_kwargs` a dictionary of extra keyword arguments to pass to the callback

How callbacks are called

All callbacks are called as follows:

```python
func(option, opt_str, value, parser, *args, **kwargs)
```

where

- `option` is the Option instance that’s calling the callback
- `opt_str` is the option string seen on the command-line that’s triggering the callback. (If an abbreviated long option was used, `opt_str` will be the full, canonical option string—e.g. if the user puts `--foo` on the command-line as an abbreviation for `--foobar`, then `opt_str` will be `"--foobar"`.)
- `value` is the argument to this option seen on the command-line. `optparse` will only expect an argument if `type` is set; the type of `value` will be the type implied by the option’s type. If `type` for this option is `None` (no argument expected), then `value` will be `None`. If `nargs` > 1, `value` will be a tuple of values of the appropriate type.
- `parser` is the `OptionParser` instance driving the whole thing, mainly useful because you can access some other interesting data through its instance attributes:
  - `parser.largs` the current list of leftover arguments, i.e. arguments that have been consumed but are neither options nor option arguments. Feel free to modify `parser.largs`, e.g. by adding more arguments to it. (This list will become `args`, the second return value of `parse_args()`.)
  - `parser.rargs` the current list of remaining arguments, i.e. with `opt_str` and `value` (if applicable) removed, and only the arguments following them still there. Feel free to modify `parser.rargs`, e.g. by consuming more arguments.
  - `parser.values` the object where option values are by default stored (an instance of `optparse.OptionValues`). This lets callbacks use the same mechanism as the rest of `optparse` for storing
option values; you don’t need to mess around with globals or closures. You can also access or modify the value(s) of any options already encountered on the command-line.

args is a tuple of arbitrary positional arguments supplied via the callback_args option attribute.

kwargs is a dictionary of arbitrary keyword arguments supplied via callback_kwargs.

Raising errors in a callback

The callback function should raise OptionValueError if there are any problems with the option or its argument(s). optparse catches this and terminates the program, printing the error message you supply to stderr. Your message should be clear, concise, accurate, and mention the option at fault. Otherwise, the user will have a hard time figuring out what he did wrong.

Callback example 1: trivial callback

Here’s an example of a callback option that takes no arguments, and simply records that the option was seen:

```python
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True
```

parser.add_option("--foo", action="callback", callback=record_foo_seen)

Of course, you could do that with the "store_true" action.

Callback example 2: check option order

Here’s a slightly more interesting example: record the fact that -a is seen, but blow up if it comes after -b in the command-line.

```python
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can’t use -a after -b")
    parser.values.a = 1
```

[...]

parser.add_option("--a", action="callback", callback=check_order)
parser.add_option("--b", action="store_true", dest="b")

Callback example 3: check option order (generalized)

If you want to re-use this callback for several similar options (set a flag, but blow up if -b has already been seen), it needs a bit of work: the error message and the flag that it sets must be generalized.

```python
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can’t use %s after -b" % opt_str)
    setattr(parser.values, option.dest, 1)
```

[...]

parser.add_option("--a", action="callback", callback=check_order, dest='a')
parser.add_option("--b", action="store_true", dest="b")
parser.add_option("--c", action="callback", callback=check_order, dest='c')

Callback example 4: check arbitrary condition

Of course, you could put any condition in there—you’re not limited to checking the values of already-defined options. For example, if you have options that should not be called when the moon is full, all you have to do is this:
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is full" % opt_str)
    setattr(parser.values, option.dest, 1)

parser.add_option("--foo",
    action="callback", callback=check_moon, dest="foo")

(The definition of is_moon_full() is left as an exercise for the reader.)

**Callback example 5: fixed arguments**

Things get slightly more interesting when you define callback options that take a fixed number of arguments. Specifying that a callback option takes arguments is similar to defining a "store" or "append" option: if you define **type**, then the option takes one argument that must be convertible to that type; if you further define **nargs**, then the option takes **nargs** arguments.

Here’s an example that just emulates the standard "store" action:

def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)

parser.add_option("--foo",
    action="callback", callback=store_value,
    type="int", nargs=3, dest="foo")

Note that **optparse** takes care of consuming 3 arguments and converting them to integers for you; all you have to do is store them. (Or whatever; obviously you don’t need a callback for this example.)

**Callback example 6: variable arguments**

Things get hairy when you want an option to take a variable number of arguments. For this case, you must write a callback, as **optparse** doesn’t provide any built-in capabilities for it. And you have to deal with certain intricacies of conventional Unix command-line parsing that **optparse** normally handles for you. In particular, callbacks should implement the conventional rules for bare -- and – arguments:

- either -- or – can be option arguments
- bare -- (if not the argument to some option): halt command-line processing and discard the --
- bare – (if not the argument to some option): halt command-line processing but keep the – (append it to **parser.largs**)

If you want an option that takes a variable number of arguments, there are several subtle, tricky issues to worry about. The exact implementation you choose will be based on which trade-offs you’re willing to make for your application (which is why **optparse** doesn’t support this sort of thing directly).

Nevertheless, here’s a stab at a callback for an option with variable arguments:

def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []

    def floatable(str):
        try:
            float(str)
            return True
        except ValueError:
            return False

    for arg in parser.rargs:
# stop on --foo like options
if arg[:2] == "--" and len(arg) > 2:
    break

# stop on -a, but not on -3 or -3.0
if arg[:1] == "-" and len(arg) > 1 and not floatable(arg):
    break
value.append(arg)

del parser.rargs[:len(value)]
setattr(parser.values, option.dest, value)

[...]
parser.add_option("-c", "--callback", dest="vararg_attr",
                       action="callback", callback=vararg_callback)

16.5.5 Extending optparse

Since the two major controlling factors in how optparse interprets command-line options are the action and type of each option, the most likely direction of extension is to add new actions and new types.

Adding new types

To add new types, you need to define your own subclass of optparse's Option class. This class has a couple of attributes that define optparse's types: TYPES and TYPE_CHECKER.

Option.TYPES
A tuple of type names; in your subclass, simply define a new tuple TYPES that builds on the standard one.

Option.TYPE_CHECKER
A dictionary mapping type names to type-checking functions. A type-checking function has the following signature:

    def check_mytype(option, opt, value)

where option is an Option instance, opt is an option string (e.g., -f), and value is the string from the command line that must be checked and converted to your desired type. check_mytype() should return an object of the hypothetical type mytype. The value returned by a type-checking function will wind up in the OptionValues instance returned by OptionParser.parse_args(), or be passed to a callback as the value parameter.

Your type-checking function should raise OptionValueError if it encounters any problems. OptionValueError takes a single string argument, which is passed as-is to OptionParser's error() method, which in turn prepends the program name and the string "error:" and prints everything to stderr before terminating the process.

Here's a silly example that demonstrates adding a "complex" option type to parse Python-style complex numbers on the command line. (This is even sillier than it used to be, because optparse 1.3 added built-in support for complex numbers, but never mind.)

First, the necessary imports:

```
from copy import copy
from optparse import Option, OptionValueError
```

You need to define your type-checker first, since it's referred to later (in the TYPE_CHECKER class attribute of your Option subclass):

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
```

16.5. optparse — Parser for command line options
raise OptionValueError("option %s: invalid complex value: %r" % (opt, value))

Finally, the Option subclass:

class MyOption(Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex

(If we didn’t make a copy() of Option.TYPE_CHECKER, we would end up modifying the TYPE_CHECKER attribute of optparse’s Option class. This being Python, nothing stops you from doing that except good manners and common sense.)

That’s it! Now you can write a script that uses the new option type just like any other optparse-based script, except you have to instruct your OptionParser to use MyOption instead of Option:

    parser = OptionParser(option_class=MyOption)
    parser.add_option("-c", type="complex")

Alternately, you can build your own option list and pass it to OptionParser; if you don’t use add_option() in the above way, you don’t need to tell OptionParser which option class to use:

    option_list = [MyOption("-c", action="store", type="complex", dest="c")]
    parser = OptionParser(option_list=option_list)

Adding new actions

Adding new actions is a bit trickier, because you have to understand that optparse has a couple of classifications for actions:

"store" actions actions that result in optparse storing a value to an attribute of the current OptionValues instance; these options require a dest attribute to be supplied to the Option constructor.

"typed" actions actions that take a value from the command line and expect it to be of a certain type; or rather, a string that can be converted to a certain type. These options require a type attribute to the Option constructor.

These are overlapping sets: some default “store” actions are "store", "store_const", "append", and "count", while the default “typed” actions are "store", "append", and "callback".

When you add an action, you need to categorize it by listing it in at least one of the following class attributes of Option (all are lists of strings):

Option.ACTIONS
    All actions must be listed in ACTIONS.

Option.STORE_ACTIONS
    "store" actions are additionally listed here.

Option.TYPED_ACTIONS
    "typed" actions are additionally listed here.

Option.ALWAYS_TYPED_ACTIONS
    Actions that always take a type (i.e. whose options always take a value) are additionally listed here. The only effect of this is that optparse assigns the default type, "string", to options with no explicit type whose action is listed in ALWAYS_TYPED_ACTIONS.

In order to actually implement your new action, you must override Option’s take_action() method and add a case that recognizes your action.

For example, let’s add an "extend" action. This is similar to the standard "append" action, but instead of taking a single value from the command-line and appending it to an existing list, "extend" will take multiple values in a single comma-delimited string, and extend an existing list with them. That is, if --names is an "extend" option of type "string", the command line

488 Chapter 16. Generic Operating System Services
--names=foo,bar --names blah --names ding,dong
would result in a list

["foo", "bar", "blah", "ding", "dong"]

Again we define a subclass of Option:

class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)

    def take_action(self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(  
                self, action, dest, opt, value, values, parser)

Features of note:

- "extend" both expects a value on the command-line and stores that value somewhere, so it goes in both
  STORE_ACTIONS and TYPED_ACTIONS.
- to ensure that optparse assigns the default type of "string" to "extend" actions, we put the
  "extend" action in ALWAYS_TYPED_ACTIONS as well.
- MyOption.take_action() implements just this one new action, and passes control back to
  Option.take_action() for the standard optparse actions.
- values is an instance of the optparse.Values class, which provides the very useful
  ensure_value() method. ensure_value() is essentially getattr() with a safety valve; it is
called as

  values.ensure_value(attr, value)

If the attr attribute of values doesn’t exist or is None, then ensure_value() first sets it to value, and
then returns “value. This is very handy for actions like "extend", "append", and "count", all of
which accumulate data in a variable and expect that variable to be of a certain type (a list for the first two, an
integer for the latter). Using ensure_value() means that scripts using your action don’t have to worry
about setting a default value for the option destinations in question; they can just leave the default as None
and ensure_value() will take care of getting it right when it’s needed.

16.6 getopt — C-style parser for command line options

Source code: Lib/getopt.py

Note: The getopt module is a parser for command line options whose API is designed to be familiar to users
of the C getopt() function. Users who are unfamiliar with the C getopt() function or who would like to
write less code and get better help and error messages should consider using the argparse module instead.

This module helps scripts to parse the command line arguments in sys.argv. It supports the same conventions
as the Unix getopt() function (including the special meanings of arguments of the form ‘-‘ and ‘--‘). Long
options similar to those supported by GNU software may be used as well via an optional third argument.

This module provides two functions and an exception:
getopt.\texttt{getopt}(\texttt{args, shortopts, longopts=\[\]})

Parses command line options and parameter list. \texttt{args} is the argument list to be parsed, without the leading reference to the running program. Typically, this means \texttt{sys.argv[1:]}. \texttt{shortopts} is the string of option letters that the script wants to recognize, with options that require an argument followed by a colon (’:’; i.e., the same format that Unix \texttt{getopt()} uses).

\textbf{Note:} Unlike GNU \texttt{getopt()}, after a non-option argument, all further arguments are considered also non-options. This is similar to the way non-GNU Unix systems work.

\texttt{longopts}, if specified, must be a list of strings with the names of the long options which should be supported. The leading ‘--’ characters should not be included in the option name. Long options which require an argument should be followed by an equal sign (‘=’). Optional arguments are not supported. To accept only long options, \texttt{shortopts} should be an empty string. Long options on the command line can be recognized so long as they provide a prefix of the option name that matches exactly one of the accepted options. For example, if \texttt{longopts} is \[’foo’, ’frob’\], the option \texttt{--fo} will match as \texttt{--foo}, but \texttt{--f} will not match uniquely, so \texttt{GetoptError} will be raised.

The return value consists of two elements: the first is a list of \texttt{(option, value)} pairs; the second is the list of program arguments left after the option list was stripped (this is a trailing slice of \texttt{args}). Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., ‘-x’) or two hyphens for long options (e.g., ‘--long-option’), and the option argument as its second element, or an empty string if the option has no argument. The options occur in the list in the same order in which they were found, thus allowing multiple occurrences. Long and short options may be mixed.

getopt.\texttt{gnu_getopt}(\texttt{args, shortopts, longopts=\[\]})

This function works like \texttt{getopt()}, except that GNU style scanning mode is used by default. This means that option and non-option arguments may be intermixed. The \texttt{getopt()} function stops processing options as soon as a non-option argument is encountered.

If the first character of the option string is ‘+’, or if the environment variable \texttt{POSIXLY_CORRECT} is set, then option processing stops as soon as a non-option argument is encountered.

\textbf{exception} getopt.\texttt{GetoptError}

This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none. The argument to the exception is a string indicating the cause of the error. For long options, an argument given to an option which does not require one will also cause this exception to be raised. The attributes \texttt{msg} and \texttt{opt} give the error message and related option; if there is no specific option to which the exception relates, \texttt{opt} is an empty string.

\textbf{exception} getopt.\texttt{error}

Alias for \texttt{GetoptError}; for backward compatibility.

An example using only Unix style options:

```
>>> import getopt
>>> args = '\-a \-b \-cfoo \-d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

Using long option names is equally easy:

```
>>> s = '--condition=foo --testing --output-file abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', [ ...
    'condition=', 'output-file=', 'testing'])
```
In a script, typical usage is something like this:

```python
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])  
    except getopt.GetoptError as err:
        # print help information and exit:

        print(err)  # will print something like "option -a not recognized"
        usage()
        sys.exit(2)

    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
            verbose = True
        elif o in ("-h", "--help"):
            usage()
            sys.exit()
        elif o in ("-o", "--output"):
            output = a
        else:
            assert False, "unhandled option"

    # ...

if __name__ == '__main__':
    main()
```

Note that an equivalent command line interface could be produced with less code and more informative help and
error messages by using the `argparse` module:

```python
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    args = parser.parse_args()
    # ... do something with args.output ...
    # ... do something with args.verbose ..
```

See Also:

Module `argparse` Alternative command line option and argument parsing library.

16.7 logging — Logging facility for Python
This module defines functions and classes which implement a flexible event logging system for applications and libraries.

The key benefit of having the logging API provided by a standard library module is that all Python modules can participate in logging, so your application log can include your own messages integrated with messages from third-party modules.

The module provides a lot of functionality and flexibility. If you are unfamiliar with logging, the best way to get to grips with it is to see the tutorials (see the links on the right).

The basic classes defined by the module, together with their functions, are listed below.

- Loggers expose the interface that application code directly uses.
- Handlers send the log records (created by loggers) to the appropriate destination.
- Filters provide a finer grained facility for determining which log records to output.
- Formatters specify the layout of log records in the final output.

16.7.1 Logger Objects

Loggers have the following attributes and methods. Note that Loggers are never instantiated directly, but always through the module-level function `logging.getLogger(name)`. Multiple calls to `getLogger()` with the same name will always return a reference to the same Logger object.

The `name` is potentially a period-separated hierarchical value, like `foo.bar.baz` (though it could also be just plain `foo`, for example). Loggers that are further down in the hierarchical list are children of loggers higher up in the list. For example, given a logger with a name of `foo`, loggers with names of `foo.bar`, `foo.bar.baz`, and `foo.bam` are all descendants of `foo`. The logger name hierarchy is analogous to the Python package hierarchy, and identical to it if you organise your loggers on a per-module basis using the recommended construction `logging.getLogger(__name__)`. That’s because in a module, `__name__` is the module’s name in the Python package namespace.

```python
class logging.Logger

Logger.propagate
    If this evaluates to true, events logged to this logger will be passed to the handlers of higher level (ancestor) loggers, in addition to any handlers attached to this logger. Messages are passed directly to the ancestor loggers’ handlers - neither the level nor filters of the ancestor loggers in question are considered.

    If this evaluates to false, logging messages are not passed to the handlers of ancestor loggers.

The constructor sets this attribute to True.
```

**Note:** If you attach a handler to a logger and one or more of its ancestors, it may emit the same record multiple times. In general, you should not need to attach a handler to more than one logger - if you just attach it to the appropriate logger which is highest in the logger hierarchy, then it will see all events logged by all descendant loggers, provided that their propagate setting is left set to True. A common scenario is to attach handlers only to the root logger, and to let propagation take care of the rest.
Logger.

- **setLevel(lvl)**
  Sets the threshold for this logger to `lvl`. Logging messages which are less severe than `lvl` will be ignored. When a logger is created, the level is set to `NOTSET` (which causes all messages to be processed when the logger is the root logger, or delegation to the parent when the logger is a non-root logger). Note that the root logger is created with level `WARNING`.

  The term ‘delegation to the parent’ means that if a logger has a level of `NOTSET`, its chain of ancestor loggers is traversed until either an ancestor with a level other than `NOTSET` is found, or the root is reached.

  If an ancestor is found with a level other than `NOTSET`, then that ancestor’s level is treated as the effective level of the logger where the ancestor search began, and is used to determine how a logging event is handled.

  If the root is reached, and it has a level of `NOTSET`, then all messages will be processed. Otherwise, the root’s level will be used as the effective level. Changed in version 3.2: The `lvl` parameter now accepts a string representation of the level such as ‘INFO’ as an alternative to the integer constants such as `INFO`.

- **isEnabledFor(lvl)**
  Indicates if a message of severity `lvl` would be processed by this logger. This method checks first the module-level level set by `logging.disable(lvl)` and then the logger’s effective level as determined by `getEffectiveLevel()`.

- **getEffectiveLevel()**
  Indicates the effective level for this logger. If a value other than `NOTSET` has been set using `setLevel()`, it is returned. Otherwise, the hierarchy is traversed towards the root until a value other than `NOTSET` is found, and that value is returned.

- **getChild(suffix)**
  Returns a logger which is a descendant to this logger, as determined by the suffix. Thus, `logging.getLogger('abc').getChild('def.ghi')` would return the same logger as would be returned by `logging.getLogger('abc.def.ghi')`. This is a convenience method, useful when the parent logger is named using e.g. `__name__` rather than a literal string. New in version 3.2.

- **debug(msg, *args, **kwargs)**
  Logs a message with level `DEBUG` on this logger. The `msg` is the message format string, and the `args` are the arguments which are merged into `msg` using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.)

  There are three keyword arguments in `kwargs` which are inspected: `exc_info` which, if it does not evaluate as false, causes exception information to be added to the logging message. If an exception tuple (in the format returned by `sys.exc_info()`) is provided, it is used; otherwise, `sys.exc_info()` is called to get the exception information.

  The second optional keyword argument is `stack_info`, which defaults to False. If specified as True, stack information is added to the logging message, including the actual logging call. Note that this is not the same stack information as that displayed through specifying `exc_info`: The former is stack frames from the bottom of the stack up to the logging call in the current thread, whereas the latter is information about stack frames which have been unwound, following an exception, while searching for exception handlers.

  You can specify `stack_info` independently of `exc_info`, e.g. to just show how you got to a certain point in your code, even when no exceptions were raised. The stack frames are printed following a header line which says:

  `Stack (most recent call last):`

  This mimics the Traceback (most recent call last): which is used when displaying exception frames.

  The third keyword argument is `extra` which can be used to pass a dictionary which is used to populate the `__dict__` of the LogRecord created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example:

  ```
  FORMAT = '%(asctime)-15s %(clientip)s %(user)-8s %(message)s'
  logging.basicConfig(format=FORMAT)
  ```
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logger = logging.getLogger('tcpserver')
logger.warning('Protocol problem: %s', 'connection reset', extra=d)

would print something like

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs Protocol problem: connection reset
```

The keys in the dictionary passed in `extra` should not clash with the keys used by the logging system. (See the `Formatter` documentation for more information on which keys are used by the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above example, for instance, the `Formatter` has been set up with a format string which expects 'clientip' and 'user' in the attribute dictionary of the LogRecord. If these are missing, the message will not be logged because a string formatting exception will occur. So in this case, you always need to pass the `extra` dictionary with these keys.

While this might be annoying, this feature is intended for use in specialized circumstances, such as multi-threaded servers where the same code executes in many contexts, and interesting conditions which arise are dependent on this context (such as remote client IP address and authenticated user name, in the above example). In such circumstances, it is likely that specialized `Formatters` would be used with particular `Handlers`. New in version 3.2: The `stack_info` parameter was added.

```
Logger.info(msg, *args, **kwargs)
    Logs a message with level INFO on this logger. The arguments are interpreted as for `debug()`.

Logger.warning(msg, *args, **kwargs)
    Logs a message with level WARNING on this logger. The arguments are interpreted as for `debug()`.

Note: There is an obsolete method `warn` which is functionally identical to `warning`. As `warn` is deprecated, please do not use it - use `warning` instead.

Logger.error(msg, *args, **kwargs)
    Logs a message with level ERROR on this logger. The arguments are interpreted as for `debug()`.

Logger.critical(msg, *args, **kwargs)
    Logs a message with level CRITICAL on this logger. The arguments are interpreted as for `debug()`.

Logger.log(lvl, msg, *args, **kwargs)
    Logs a message with integer level `lvl` on this logger. The other arguments are interpreted as for `debug()`.

Logger.exception(msg, *args)
    Logs a message with level ERROR on this logger. The arguments are interpreted as for `debug()` . Exception info is added to the logging message. This method should only be called from an exception handler.

Logger.addFilter(filt)
    Adds the specified filter `filt` to this logger.

Logger.removeFilter(filt)
    Removes the specified filter `filt` from this logger.

Logger.filter(record)
    Applies this logger’s filters to the record and returns a true value if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be processed (passed to handlers). If one returns a false value, no further processing of the record occurs.

Logger.addHandler(hdlr)
    Adds the specified handler `hdlr` to this logger.

Logger.removeHandler(hdlr)
    Removes the specified handler `hdlr` from this logger.
```
Logger.findCaller(stack_info=False)
Finds the caller’s source filename and line number. Returns the filename, line number, function name and stack information as a 4-element tuple. The stack information is returned as None unless stack_info is True.

Logger.handle(record)
Handles a record by passing it to all handlers associated with this logger and its ancestors (until a false value of propagate is found). This method is used for unpickled records received from a socket, as well as those created locally. Logger-level filtering is applied using filter().

Logger.makeRecord(name, lvl, fn, lno, msg, args, exc_info, func=None, extra=None, sinfo=None)
This is a factory method which can be overridden in subclasses to create specialized LogRecord instances.

Logger.hasHandlers()
Checks to see if this logger has any handlers configured. This is done by looking for handlers in this logger and its parents in the logger hierarchy. Returns True if a handler was found, else False. The method stops searching up the hierarchy whenever a logger with the ‘propagate’ attribute set to False is found - that will be the last logger which is checked for the existence of handlers. New in version 3.2.

16.7.2 Handler Objects

Handlers have the following attributes and methods. Note that Handler is never instantiated directly; this class acts as a base for more useful subclasses. However, the __init__() method in subclasses needs to call Handler.__init__().

Handler.__init__(level=NOTSET)
Initializes the Handler instance by setting its level, setting the list of filters to the empty list and creating a lock (using createLock()) for serializing access to an I/O mechanism.

Handler.createLock()
Initializes a thread lock which can be used to serialize access to underlying I/O functionality which may not be threadsafe.

Handler.acquire()
Acquires the thread lock created with createLock().

Handler.release()
Releases the thread lock acquired with acquire().

Handler.setLevel(lvl)
Sets the threshold for this handler to lvl. Logging messages which are less severe than lvl will be ignored. When a handler is created, the level is set to NOTSET (which causes all messages to be processed). Changed in version 3.2: The lvl parameter now accepts a string representation of the level such as ‘INFO’ as an alternative to the integer constants such as INFO.

Handler.setFormatter(form)
Sets the Formatter for this handler to form.

Handler.addFilter(filt)
Adds the specified filter filt to this handler.

Handler.removeFilter(filt)
Removes the specified filter filt from this handler.

Handler.filter(record)
Applies this handler’s filters to the record and returns a true value if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be emitted. If one returns a false value, the handler will not emit the record.

Handler.flush()
Ensure all logging output has been flushed. This version does nothing and is intended to be implemented by subclasses.

Handler.close()
Tidy up any resources used by the handler. This version does no output but removes the handler from an
internal list of handlers which is closed when `shutdown()` is called. Subclasses should ensure that this gets called from overridden `close()` methods.

**Handler.handle** *(record)*
Conditionally emits the specified logging record, depending on filters which may have been added to the handler. Wraps the actual emission of the record with acquisition/release of the I/O thread lock.

**Handler.handleError** *(record)*
This method should be called from handlers when an exception is encountered during an `emit()` call. If the module-level attribute `raiseExceptions` is `False`, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The specified record is the one which was being processed when the exception occurred. (The default value of `raiseExceptions` is `True`, as that is more useful during development).

**Handler.format** *(record)*
Do formatting for a record - if a formatter is set, use it. Otherwise, use the default formatter for the module.

**Handler.emit** *(record)*
Do whatever it takes to actually log the specified logging record. This version is intended to be implemented by subclasses and so raises a `NotImplementedError`.

For a list of handlers included as standard, see `logging.handlers`.

### 16.7.3 Formatter Objects

**Formatter** objects have the following attributes and methods. They are responsible for converting a `LogRecord` to (usually) a string which can be interpreted by either a human or an external system. The base `Formatter` allows a formatting string to be specified. If none is supplied, the default value of `'%(message)s'` is used.

A Formatter can be initialized with a format string which makes use of knowledge of the `LogRecord` attributes - such as the default value mentioned above making use of the fact that the user’s message and arguments are pre-formatted into a `LogRecord`'s `message` attribute. This format string contains standard Python %-style mapping keys. See section `printf-style String Formatting` for more information on string formatting.

The useful mapping keys in a `LogRecord` are given in the section on `LogRecord attributes`.

```python
class logging.Formatter (fmt=None, datefmt=None, style='%')
Returns a new instance of the Formatter class. The instance is initialized with a format string for the message as a whole, as well as for a format string for the date/time portion of a message. If no `fmt` is specified, ‘%(message)s’ is used. If no `datefmt` is specified, the ISO8601 date format is used.

The `style` parameter can be one of ‘%’, ‘{’ or ‘$’ and determines how the format string will be merged with its data: using one of %-formatting, `str.format()` or `string.Template`. Changed in version 3.2: The `style` parameter was added.

**format** *(record)*
The record’s attribute dictionary is used as the operand to a string formatting operation. Returns the resulting string. Before formatting the dictionary, a couple of preparatory steps are carried out. The `message` attribute of the record is computed using `msg % args`. If the formatting string contains ‘%(asctime)’, `formatTime()` is called to format the event time. If there is exception information, it is formatted using `formatException()` and appended to the message. Note that the formatted exception information is cached in attribute `exc_text`. This is useful because the exception information can be pickled and sent across the wire, but you should be careful if you have more than one `Formatter` subclass which customizes the formatting of exception information. In this case, you will have to clear the cached value after a formatter has done its formatting, so that the next formatter to handle the event doesn’t use the cached value but recalculates it afresh.

If stack information is available, it’s appended after the exception information, using `formatStack()` to transform it if necessary.
```
formatTime(record, datefmt=None)

This method should be called from format() by a formatter which wants to make use of a formatted time. This method can be overridden in formatters to provide for any specific requirement, but the basic behavior is as follows: if datefmt (a string) is specified, it is used with time.strftime() to format the creation time of the record. Otherwise, the ISO8601 format is used. The resulting string is returned.

This function uses a user-configurable function to convert the creation time to a tuple. By default, time.localtime() is used; to change this for a particular formatter instance, set the converter attribute to a function with the same signature as time.localtime() or time.gmtime(). To change it for all formatters, for example if you want all logging times to be shown in GMT, set the converter attribute in the Formatter class. Changed in version 3.3: Previously, the default ISO 8601 format was hard-coded as in this example: 2010-09-06 22:38:15,292 where the part before the comma is handled by a strptime format string ("%Y-%m-%d %H:%M:%S"), and the part after the comma is a millisecond value. Because strftime does not have a format placeholder for milliseconds, the millisecond value is appended using another format string, "%s,%03d" – and both of these format strings have been hardcoded into this method. With the change, these strings are defined as class-level attributes which can be overridden at the instance level when desired. The names of the attributes are default_time_format (for the strftime format string) and default_msec_format (for appending the millisecond value).

formatException(exc_info)

Formats the specified exception information (a standard exception tuple as returned by sys.exc_info()) as a string. This default implementation just uses traceback.print_exception(). The resulting string is returned.

formatStack(stack_info)

Formats the specified stack information (a string as returned by traceback.print_stack(), but with the last newline removed) as a string. This default implementation just returns the input value.

16.7.4 Filter Objects

Filters can be used by Handlers and Loggers for more sophisticated filtering than is provided by levels. The base filter class only allows events which are below a certain point in the logger hierarchy. For example, a filter initialized with ‘A.B’ will allow events logged by loggers ‘A.B’, ‘A.B.C’, ‘A.B.C.D’, ‘A.B.D’ etc. but not ‘A.BB’, ‘B.A.B’ etc. If initialized with the empty string, all events are passed.

class logging.Filter(name='')

Returns an instance of the Filter class. If name is specified, it names a logger which, together with its children, will have its events allowed through the filter. If name is the empty string, allows every event.

filter(record)

Is the specified record to be logged? Returns zero for no, nonzero for yes. If deemed appropriate, the record may be modified in-place by this method.

Note that filters attached to handlers are consulted before an event is emitted by the handler, whereas filters attached to loggers are consulted whenever an event is logged (using debug(), info(), etc.), before sending an event to handlers. This means that events which have been generated by descendant loggers will not be filtered by a logger’s filter setting, unless the filter has also been applied to those descendant loggers.

You don’t actually need to subclass Filter: you can pass any instance which has a filter method with the same semantics. Changed in version 3.2: You don’t need to create specialized Filter classes, or use other classes with a filter method: you can use a function (or other callable) as a filter. The filtering logic will check to see if the filter object has a filter attribute: if it does, it’s assumed to be a Filter and its filter() method is called. Otherwise, it’s assumed to be a callable and called with the record as the single parameter. The returned value should conform to that returned by filter(). Although filters are used primarily to filter records based on more sophisticated criteria than levels, they get to see every record which is processed by the handler or logger they’re attached to: this can be useful if you want to do things like counting how many records were processed by a particular logger or handler, or adding, changing or removing attributes in the LogRecord being processed. Obviously changing the LogRecord needs to be done with some care, but it does allow the injection of contextual information into logs (see filters-contextual).
16.7.5 LogRecord Objects

LogRecord instances are created automatically by the Logger every time something is logged, and can be created manually via makeLogRecord() (for example, from a pickled event received over the wire).

class logging.LogRecord(name, level, pathname, lineno, msg, args, exc_info, func=None, sinfo=None)

Contains all the information pertinent to the event being logged.

The primary information is passed in msg and args, which are combined using msg % args to create the message field of the record.

Parameters

- name – The name of the logger used to log the event represented by this LogRecord. Note that this name will always have this value, even though it may be emitted by a handler attached to a different (ancestor) logger.

- level – The numeric level of the logging event (one of DEBUG, INFO etc.) Note that this is converted to two attributes of the LogRecord: levelno for the numeric value andlevelname for the corresponding level name.

- pathname – The full pathname of the source file where the logging call was made.

- lineno – The line number in the source file where the logging call was made.

- msg – The event description message, possibly a format string with placeholders for variable data.

- args – Variable data to merge into the msg argument to obtain the event description.

- exc_info – An exception tuple with the current exception information, or None if no exception information is available.

- func – The name of the function or method from which the logging call was invoked.

- sinfo – A text string representing stack information from the base of the stack in the current thread, up to the logging call.

getMessage() Returns the message for this LogRecord instance after merging any user-supplied arguments with the message. If the user-supplied message argument to the logging call is not a string, str() is called on it to convert it to a string. This allows use of user-defined classes as messages, whose __str__ method can return the actual format string to be used.

Changed in version 3.2: The creation of a LogRecord has been made more configurable by providing a factory which is used to create the record. The factory can be set using getLogRecordFactory() and setLogRecordFactory() (see this for the factory’s signature). This functionality can be used to inject your own values into a LogRecord at creation time. You can use the following pattern:

old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdeadbeef
    return record

logging.setLogRecordFactory(record_factory)

With this pattern, multiple factories could be chained, and as long as they don’t overwrite each other’s attributes or unintentionally overwrite the standard attributes listed above, there should be no surprises.
16.7.6 LogRecord attributes

The LogRecord has a number of attributes, most of which are derived from the parameters to the constructor. (Note that the names do not always correspond exactly between the LogRecord constructor parameters and the LogRecord attributes.) These attributes can be used to merge data from the record into the format string. The following table lists (in alphabetical order) the attribute names, their meanings and the corresponding placeholder in a %-style format string.

If you are using {}-formatting (str.format()), you can use {attrname} as the placeholder in the format string. If you are using $-formatting (string.Template), use the form ${attrname}. In both cases, of course, replace attrname with the actual attribute name you want to use.

In the case of {}-formatting, you can specify formatting flags by placing them after the attribute name, separated from it with a colon. For example: a placeholder of {msecs:03d} would format a millisecond value of 4 as 004. Refer to the str.format() documentation for full details on the options available to you.
### LoggerAdapter Objects

**LoggerAdapter** instances are used to conveniently pass contextual information into logging calls. For a usage example, see the section on *adding contextual information to your logging output.*

class logging.LoggerAdapter (logger, extra)

    Returns an instance of LoggerAdapter initialized with an underlying Logger instance and a dict-like object.

    process (msg, kwargs)

    Modifies the message and/or keyword arguments passed to a logging call in order to insert contextual
information. This implementation takes the object passed as *extra* to the constructor and adds it to *kwargs* using key ‘extra’. The return value is a (*msg*, *kwargs*) tuple which has the (possibly modified) versions of the arguments passed in.

In addition to the above, *LoggerAdapter* supports the following methods of *Logger*, i.e. *debug()*,, *info()*,, *warning()*,, *error()*,, *exception()*,, *critical()*,, *log()*, *isEnabledFor()*, *getEffectiveLevel()*, *setLevel()*, *hasHandlers()*. These methods have the same signatures as their counterparts in *Logger*, so you can use the two types of instances interchangeably. Changed in version 3.2: The *isEnabledFor()*, *getEffectiveLevel()*, *setLevel()* and *hasHandlers()* methods were added to *LoggerAdapter*. These methods delegate to the underlying logger.

### 16.7.8 Thread Safety

The logging module is intended to be thread-safe without any special work needing to be done by its clients. It achieves this though using threading locks; there is one lock to serialize access to the module’s shared data, and each handler also creates a lock to serialize access to its underlying I/O.

If you are implementing asynchronous signal handlers using the *signal* module, you may not be able to use logging from within such handlers. This is because lock implementations in the *threading* module are not always re-entrant, and so cannot be invoked from such signal handlers.

### 16.7.9 Module-Level Functions

In addition to the classes described above, there are a number of module-level functions.

**logging.getLogger** (*name=None*)

Return a logger with the specified name or, if name is *None*, return a logger which is the root logger of the hierarchy. If specified, the name is typically a dot-separated hierarchical name like ‘a’, ‘a.b’ or ‘a.b.c.d’.

Choice of these names is entirely up to the developer who is using logging.

All calls to this function with a given name return the same logger instance. This means that logger instances never need to be passed between different parts of an application.

**logging.getLoggerClass()**

Return either the standard *Logger* class, or the last class passed to *setLoggerClass()*. This function may be called from within a new class definition, to ensure that installing a customised *Logger* class will not undo customisations already applied by other code. For example:

```python
class MyLogger(logging.getLoggerClass()):
    # ... override behaviour here
```

**logging.getLoggerRecordFactory()**

Return a callable which is used to create a *LogRecord*. New in version 3.2: This function has been provided, along with *setLogRecordFactory()*, to allow developers more control over how the *LogRecord* representing a logging event is constructed. See *setLogRecordFactory()* for more information about how the factory is called.

**logging.debug** (*msg*, *args*, **kwargs*)

Logs a message with level *DEBUG* on the root logger. The *msg* is the message format string, and the *args* are the arguments which are merged into *msg* using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.)

There are three keyword arguments in *kwargs* which are inspected: *exc_info* which, if it does not evaluate as false, causes exception information to be added to the logging message. If an exception tuple (in the format returned by *sys.exc_info()*) is provided, it is used; otherwise, *sys.exc_info()* is called to get the exception information.

The second optional keyword argument is *stack_info*, which defaults to False. If specified as True, stack information is added to the logging message, including the actual logging call. Note that this is not the same stack information as that displayed through specifying *exc_info*: The former is stack frames from the
bottom of the stack up to the logging call in the current thread, whereas the latter is information about stack
frames which have been unwound, following an exception, while searching for exception handlers.

You can specify `stack_info` independently of `exc_info`, e.g. to just show how you got to a certain point in
your code, even when no exceptions were raised. The stack frames are printed following a header line which
says:

```
Stack (most recent call last):
```

This mimics the `Traceback (most recent call last):` which is used when displaying exception frames.

The third optional keyword argument is `extra` which can be used to pass a dictionary which is used to popu-
late the `__dict__` of the LogRecord created for the logging event with user-defined attributes. These custom
attributes can then be used as you like. For example, they could be incorporated into logged messages. For
example:

```
FORMAT = '%(asctime)-15s %(clientip)s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logging.warning('Protocol problem: %s', 'connection reset', extra=d)
```

would print something like:

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs Protocol problem: connection reset
```

The keys in the dictionary passed in `extra` should not clash with the keys used by the logging system. (See
the `Formatter` documentation for more information on which keys are used by the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above
example, for instance, the `Formatter` has been set up with a format string which expects ‘clientip’ and
‘user’ in the attribute dictionary of the LogRecord. If these are missing, the message will not be logged
because a string formatting exception will occur. So in this case, you always need to pass the `extra` dictionary
with these keys.

While this might be annoying, this feature is intended for use in specialized circumstances, such as multi-
threaded servers where the same code executes in many contexts, and interesting conditions which arise
are dependent on this context (such as remote client IP address and authenticated user name, in the above
example). In such circumstances, it is likely that specialized `Formatter`s would be used with particular
Handlers. New in version 3.2: The `stack_info` parameter was added.

```
logging.info(msg, *args, **kwargs)
Logs a message with level INFO on the root logger. The arguments are interpreted as for `debug()`.
```

```
logging.warning(msg, *args, **kwargs)
Logs a message with level WARNING on the root logger. The arguments are interpreted as for `debug()`.
```

```
logging.error(msg, *args, **kwargs)
Logs a message with level ERROR on the root logger. The arguments are interpreted as for `debug()`.
```

```
logging.critical(msg, *args, **kwargs)
Logs a message with level CRITICAL on the root logger. The arguments are interpreted as for `debug()`.
```

```
logging.exception(msg, *args)
Logs a message with level ERROR on the root logger. The arguments are interpreted as for `debug()`.
Exception info is added to the logging message. This function should only be called from an exception handler.
```

```
Note: There is an obsolete function `warn` which is functionally identical to `warning`. As `warn`
is deprecated, please do not use it - use `warning` instead.
```
logging.\texttt{log}(level, msg, *args, **kwargs)
Logs a message with level \textit{level} on the root logger. The other arguments are interpreted as for \texttt{debug}.

**Note:** The above module-level functions which delegate to the root logger should \textit{not} be used in threads, in versions of Python earlier than 2.7.1 and 3.2, unless at least one handler has been added to the root logger \textit{before} the threads are started. These convenience functions call \texttt{basicConfig()} to ensure that at least one handler is available; in earlier versions of Python, this can (under rare circumstances) lead to handlers being added multiple times to the root logger, which can in turn lead to multiple messages for the same event.

logging.\texttt{disable}(lvl)
Provides an overriding level \textit{lvl} for all loggers which takes precedence over the logger’s own level. When the need arises to temporarily throttle logging output down across the whole application, this function can be useful. Its effect is to disable all logging calls of severity \textit{lvl} and below, so that if you call it with a value of \texttt{INFO}, then all \texttt{INFO} and \texttt{DEBUG} events would be discarded, whereas those of severity \texttt{WARNING} and above would be processed according to the logger’s effective level. To undo the effect of a call to \texttt{logging.disable(lvl)}, call \texttt{logging.disable(logging.NOTSET)}.

logging.\texttt{addLevelName}(lvl, levelName)
Associates level \textit{lvl} with text \textit{levelName} in an internal dictionary, which is used to map numeric levels to a textual representation, for example when a \texttt{Formatter} formats a message. This function can also be used to define your own levels. The only constraints are that all levels used must be registered using this function, levels should be positive integers and they should increase in increasing order of severity.

**Note:** If you are thinking of defining your own levels, please see the section on \texttt{custom-levels}.

logging.\texttt{getLevelName}(lvl)
Returns the textual representation of logging level \textit{lvl}. If the level is one of the predefined levels \texttt{CRITICAL}, \texttt{ERROR}, \texttt{WARNING}, \texttt{INFO} or \texttt{DEBUG} then you get the corresponding string. If you have associated levels with names using \texttt{addLevelName()} then the name you have associated with \textit{lvl} is returned. If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned. Otherwise, the string ‘Level %s’ % \textit{lvl} is returned.

logging.\texttt{makeLogRecord}(attrdict)
Creates and returns a new \texttt{LogRecord} instance whose attributes are defined by \textit{attrdict}. This function is useful for taking a pickled \texttt{LogRecord} attribute dictionary, sent over a socket, and reconstituting it as a \texttt{LogRecord} instance at the receiving end.

logging.\texttt{basicConfig}(**kwargs)
Does basic configuration for the logging system by creating a \texttt{StreamHandler} with a default \texttt{Formatter} and adding it to the root logger. The functions \texttt{debug()}, \texttt{info()}, \texttt{warning()}, \texttt{error()} and \texttt{critical()} will call \texttt{basicConfig()} automatically if no handlers are defined for the root logger.

This function does nothing if the root logger already has handlers configured for it.

**Note:** This function should be called from the main thread before other threads are started. In versions of Python prior to 2.7.1 and 3.2, if this function is called from multiple threads, it is possible (in rare circumstances) that a handler will be added to the root logger more than once, leading to unexpected results such as messages being duplicated in the log.

The following keyword arguments are supported.

16.7. \texttt{logging} — Logging facility for Python

503
<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>Specifies that a FileHandler be created, using the specified filename, rather than a StreamHandler.</td>
</tr>
<tr>
<td>filemode</td>
<td>Specifies the mode to open the file, if filename is specified (if filemode is unspecified, it defaults to ‘a’).</td>
</tr>
<tr>
<td>format</td>
<td>Use the specified format string for the handler.</td>
</tr>
<tr>
<td>datefmt</td>
<td>Use the specified date/time format.</td>
</tr>
<tr>
<td>style</td>
<td>If format is specified, use this style for the format string. One of ‘%’, ‘{’ or ‘$’ for %-formatting, str.format() or string.Template respectively, and defaulting to ‘%’ if not specified.</td>
</tr>
<tr>
<td>level</td>
<td>Set the root logger level to the specified level.</td>
</tr>
<tr>
<td>stream</td>
<td>Use the specified stream to initialize the StreamHandler. Note that this argument is incompatible with ‘filename’ - if both are present, a ValueError is raised.</td>
</tr>
<tr>
<td>handlers</td>
<td>If specified, this should be an iterable of already created handlers to add to the root logger. Any handlers which don’t already have a formatter set will be assigned the default formatter created in this function. Note that this argument is incompatible with ‘filename’ or ‘stream’ - if both are present, a ValueError is raised.</td>
</tr>
</tbody>
</table>

Changed in version 3.2: The style argument was added. Changed in version 3.3: The handlers argument was added. Additional checks were added to catch situations where incompatible arguments are specified (e.g. handlers together with stream or filename, or stream together with filename).

**logging.shutdown()**

Informs the logging system to perform an orderly shutdown by flushing and closing all handlers. This should be called at application exit and no further use of the logging system should be made after this call.

**logging.setLoggerClass(klass)**

Tells the logging system to use the class klass when instantiating a logger. The class should define __init__() such that only a name argument is required, and the __init__() should call Logger.__init__(). This function is typically called before any loggers are instantiated by applications which need to use custom logger behavior.

**logging.setLogRecordFactory (factory)**

Set a callable which is used to create a LogRecord.

**Parameters**

- **factory** – The factory callable to be used to instantiate a log record.

New in version 3.2: This function has been provided, along with getLogRecordFactory(), to allow developers more control over how the LogRecord representing a logging event is constructed. The factory has the following signature:

```python
factory(name, level, fn, lno, msg, args, exc_info, func=None, sinfo=None, **kwargs)
```

- **name** – The logger name.
- **level** – The logging level (numeric).
- **fn** – The full pathname of the file where the logging call was made.
- **lno** – The line number in the file where the logging call was made.
- **msg** – The logging message.
- **args** – The arguments for the logging message.
- **exc_info** – An exception tuple, or None.
- **func** – The name of the function or method which invoked the logging call.
- **sinfo** – A stack traceback such as is provided by traceback.print_stack(), showing the call hierarchy.
- **kwargs** – Additional keyword arguments.
16.7.10 Module-Level Attributes

logging.lastResort
A “handler of last resort” is available through this attribute. This is a StreamHandler writing to sys.stderr with a level of WARNING, and is used to handle logging events in the absence of any logging configuration. The end result is to just print the message to sys.stderr. This replaces the earlier error message saying that “no handlers could be found for logger XYZ”. If you need the earlier behaviour for some reason, lastResort can be set to None. New in version 3.2.

16.7.11 Integration with the warnings module

The captureWarnings() function can be used to integrate logging with the warnings module.

logging.captureWarnings(capture)
This function is used to turn the capture of warnings by logging on and off.

If capture is True, warnings issued by the warnings module will be redirected to the logging system. Specifically, a warning will be formatted using warnings.formatwarning() and the resulting string logged to a logger named ‘py.warnings’ with a severity of WARNING.

If capture is False, the redirection of warnings to the logging system will stop, and warnings will be redirected to their original destinations (i.e. those in effect before captureWarnings(True) was called).

See Also:

Module logging.config Configuration API for the logging module.
Module logging.handlers Useful handlers included with the logging module.

PEP 282 - A Logging System The proposal which described this feature for inclusion in the Python standard library.

Original Python logging package This is the original source for the logging package. The version of the package available from this site is suitable for use with Python 1.5.2, 2.1.x and 2.2.x, which do not include the logging package in the standard library.

16.8 logging.config — Logging configuration

This section describes the API for configuring the logging module.

16.8.1 Configuration functions

The following functions configure the logging module. They are located in the logging.config module. Their use is optional — you can configure the logging module using these functions or by making calls to the main API (defined in logging itself) and defining handlers which are declared either in logging or logging.handlers.

logging.config.dictConfig(config)
Takes the logging configuration from a dictionary. The contents of this dictionary are described in Configuration dictionary schema below.

If an error is encountered during configuration, this function will raise a `ValueError`, `TypeError`, `AttributeError` or `ImportError` with a suitably descriptive message. The following is a (possibly incomplete) list of conditions which will raise an error:

- A level which is not a string or which is a string not corresponding to an actual logging level.
- A `propagate` value which is not a boolean.
- An id which does not have a corresponding destination.
- A non-existent handler id found during an incremental call.
- An invalid logger name.
- Inability to resolve to an internal or external object.

Parsing is performed by the `DictConfigurator` class, whose constructor is passed the dictionary used for configuration, and has a `configure()` method. The `logging.config` module has a callable attribute `dictConfigClass` which is initially set to `DictConfigurator`. You can replace the value of `dictConfigClass` with a suitable implementation of your own.

`dictConfig()` calls `dictConfigClass` passing the specified dictionary, and then calls the `configure()` method on the returned object to put the configuration into effect:

```python
def dictConfig(config):
    dictConfigClass(config).configure()
```

For example, a subclass of `DictConfigurator` could call `DictConfigurator.__init__() in its own __init__(), then set up custom prefixes which would be usable in the subsequent `configure()` call. `dictConfigClass` would be bound to this new subclass, and then `dictConfig()` could be called exactly as in the default, uncustomized state.

New in version 3.2.

`logging.config.fileConfig(fname, defaults=None, disable_existing_loggers=True)`

Reads the logging configuration from a `configparser`-format file named `fname`. This function can be called several times from an application, allowing an end user to select from various pre-canned configurations (if the developer provides a mechanism to present the choices and load the chosen configuration).

Parameters

- `defaults` – Defaults to be passed to the ConfigParser can be specified in this argument.
- `disable_existing_loggers` – If specified as `False`, loggers which exist when this call is made are left alone. The default is `True` because this enables old behaviour in a backward-compatible way. This behaviour is to disable any existing loggers unless they or their ancestors are explicitly named in the logging configuration.

`logging.config.listen(port=DEFAULT_LOGGING_CONFIG_PORT)`

Starts up a socket server on the specified port, and listens for new configurations. If no port is specified, the module’s default `DEFAULT_LOGGING_CONFIG_PORT` is used. Logging configurations will be sent as a file suitable for processing by `fileConfig()`. Returns a `Thread` instance on which you can call `start()` to start the server, and which you can join() when appropriate. To stop the server, call `stopListening()`.

To send a configuration to the socket, read in the configuration file and send it to the socket as a string of bytes preceded by a four-byte length string packed in binary using `struct.pack('>L', n)`.

Note: Because portions of the configuration are passed through `eval()`, use of this function may open its users to a security risk. While the function only binds to a socket on `localhost`, and so does not accept
connections from remote machines, there are scenarios where untrusted code could be run under the account of the process which calls `listen()`. Specifically, if the process calling `listen()` runs on a multi-user machine where users cannot trust each other, then a malicious user could arrange to run essentially arbitrary code in a victim user’s process, simply by connecting to the victim’s `listen()` socket and sending a configuration which runs whatever code the attacker wants to have executed in the victim’s process. This is especially easy to do if the default port is used, but not hard even if a different port is used).

logging.config.stopListening()

Stops the listening server which was created with a call to `listen()`. This is typically called before calling `join()` on the return value from `listen()`.

### 16.8.2 Configuration dictionary schema

Describing a logging configuration requires listing the various objects to create and the connections between them; for example, you may create a handler named ‘console’ and then say that the logger named ‘startup’ will send its messages to the ‘console’ handler. These objects aren’t limited to those provided by the `logging` module because you might write your own formatter or handler class. The parameters to these classes may also need to include external objects such as `sys.stderr`. The syntax for describing these objects and connections is defined in `Object connections` below.

#### Dictionary Schema Details

The dictionary passed to `dictConfig()` must contain the following keys:

- **version** - to be set to an integer value representing the schema version. The only valid value at present is 1, but having this key allows the schema to evolve while still preserving backwards compatibility.

All other keys are optional, but if present they will be interpreted as described below. In all cases below where a ‘configuring dict’ is mentioned, it will be checked for the special ‘()`’ key to see if a custom instantiation is required. If so, the mechanism described in `User-defined objects` below is used to create an instance; otherwise, the context is used to determine what to instantiate.

- **formatters** - the corresponding value will be a dict in which each key is a formatter id and each value is a dict describing how to configure the corresponding `Formatter` instance.

  The configuring dict is searched for keys `format` and `datefmt` (with defaults of `None`) and these are used to construct a `Formatter` instance.

- **filters** - the corresponding value will be a dict in which each key is a filter id and each value is a dict describing how to configure the corresponding `Filter` instance.

  The configuring dict is searched for the key `name` (defaulting to the empty string) and this is used to construct a `logging.Filter` instance.

- **handlers** - the corresponding value will be a dict in which each key is a handler id and each value is a dict describing how to configure the corresponding `Handler` instance.

The configuring dict is searched for the following keys:

- **class** (mandatory). This is the fully qualified name of the handler class.
- **level** (optional). The level of the handler.
- **formatter** (optional). The id of the formatter for this handler.
- **filters** (optional). A list of ids of the filters for this handler.

All other keys are passed through as keyword arguments to the handler’s constructor. For example, given the snippet:

```python
handlers:
  console:
    class : logging.StreamHandler
```

---

507
formatter: brief
level : INFO
filters: [allow_foo]
stream : ext://sys.stdout

file:
class : logging.handlers.RotatingFileHandler
formatter: precise
filename: logconfig.log
maxBytes: 1024
backupCount: 3

the handler with id console is instantiated as a logging.StreamHandler, using sys.stdout as the underlying stream. The handler with id file is instantiated as a logging.handlers.RotatingFileHandler with the keyword arguments filename='logconfig.log', maxBytes=1024, backupCount=3.

• loggers - the corresponding value will be a dict in which each key is a logger name and each value is a dict describing how to configure the corresponding Logger instance.

The configuring dict is searched for the following keys:

  – level (optional). The level of the logger.
  – propagate (optional). The propagation setting of the logger.
  – filters (optional). A list of ids of the filters for this logger.
  – handlers (optional). A list of ids of the handlers for this logger.

The specified loggers will be configured according to the level, propagation, filters and handlers specified.

• root - this will be the configuration for the root logger. Processing of the configuration will be as for any logger, except that the propagate setting will not be applicable.

• incremental - whether the configuration is to be interpreted as incremental to the existing configuration. This value defaults to False, which means that the specified configuration replaces the existing configuration with the same semantics as used by the existing fileConfig() API.

  If the specified value is True, the configuration is processed as described in the section on Incremental Configuration.

• disable_existing_loggers - whether any existing loggers are to be disabled. This setting mirrors the parameter of the same name in fileConfig(). If absent, this parameter defaults to True. This value is ignored if incremental is True.

Incremental Configuration

It is difficult to provide complete flexibility for incremental configuration. For example, because objects such as filters and formatters are anonymous, once a configuration is set up, it is not possible to refer to such anonymous objects when augmenting a configuration.

Furthermore, there is not a compelling case for arbitrarily altering the object graph of loggers, handlers, filters, formatters at run-time, once a configuration is set up; the verbosity of loggers and handlers can be controlled just by setting levels (and, in the case of loggers, propagation flags). Changing the object graph arbitrarily in a safe way is problematic in a multi-threaded environment; while not impossible, the benefits are not worth the complexity it adds to the implementation.

Thus, when the incremental key of a configuration dict is present and is True, the system will completely ignore any formatters and filters entries, and process only the level settings in the handlers entries, and the level and propagate settings in the loggers and root entries.

Using a value in the configuration dict lets configurations to be sent over the wire as pickled dicts to a socket listener. Thus, the logging verbosity of a long-running application can be altered over time with no need to stop and restart the application.
Object connections

The schema describes a set of logging objects - loggers, handlers, formatters, filters - which are connected to each other in an object graph. Thus, the schema needs to represent connections between the objects. For example, say that, once configured, a particular logger has attached to it a particular handler. For the purposes of this discussion, we can say that the logger represents the source, and the handler the destination, of a connection between the two. Of course in the configured objects this is represented by the logger holding a reference to the handler. In the configuration dict, this is done by giving each destination object an id which identifies it unambiguously, and then using the id in the source object’s configuration to indicate that a connection exists between the source and the destination object with that id.

So, for example, consider the following YAML snippet:

```
formatters:
    brief:
        # configuration for formatter with id 'brief' goes here
    precise:
        # configuration for formatter with id 'precise' goes here

handlers:
    h1: This is an id
        # configuration of handler with id 'h1' goes here
    formatter: brief
    h2: This is another id
        # configuration of handler with id 'h2' goes here
    formatter: precise

loggers:
    foo.bar.baz:
        # other configuration for logger 'foo.bar.baz'
    handlers: [h1, h2]
```

(Note: YAML used here because it’s a little more readable than the equivalent Python source form for the dictionary.)

The ids for loggers are the logger names which would be used programmatically to obtain a reference to those loggers, e.g. `foo.bar.baz`. The ids for Formatters and Filters can be any string value (such as `brief`, `precise` above) and they are transient, in that they are only meaningful for processing the configuration dictionary and used to determine connections between objects, and are not persisted anywhere when the configuration call is complete.

The above snippet indicates that logger named `foo.bar.baz` should have two handlers attached to it, which are described by the handler ids `h1` and `h2`. The formatter for `h1` is that described by id `brief`, and the formatter for `h2` is that described by id `precise`.

User-defined objects

The schema supports user-defined objects for handlers, filters and formatters. (Loggers do not need to have different types for different instances, so there is no support in this configuration schema for user-defined logger classes.)

Objects to be configured are described by dictionaries which detail their configuration. In some places, the logging system will be able to infer from the context how an object is to be instantiated, but when a user-defined object is to be instantiated, the system will not know how to do this. In order to provide complete flexibility for user-defined object instantiation, the user needs to provide a ‘factory’ - a callable which is called with a configuration dictionary and which returns the instantiated object. This is signalled by an absolute import path to the factory being made available under the special key ‘()’. Here’s a concrete example:

```
formatters:
    brief:
        format: '%(message)s'
    default:
        format: '%(asctime)s %(levelname)-8s %(name)-15s %(message)s'
        datefmt: '%Y-%m-%d %H:%M:%S'
```
The above YAML snippet defines three formatters. The first, with id `brief`, is a standard `logging.Formatter` instance with the specified format string. The second, with id `default`, has a longer format and also defines the time format explicitly, and will result in a `logging.Formatter` initialized with those two format strings. Shown in Python source form, the `brief` and `default` formatters have configuration sub-dictionaries:

```
{'format': '%(message)s'}
```

and:

```
{'format': '%(asctime)s %(levelname)-8s %(name)-15s %(message)s',
 'datefmt': '%Y-%m-%d %H:%M:%S'}
```

respectively, and as these dictionaries do not contain the special key `()` , the instantiation is inferred from the context: as a result, standard `logging.Formatter` instances are created. The configuration sub-dictionary for the third formatter, with id `custom`, is:

```
{()
 my.package.customFormatterFactory,
 'bar': 'baz',
 'spam': 99.9,
 'answer': 42
}
```

and this contains the special key `()` , which means that user-defined instantiation is wanted. In this case, the specified factory callable will be used. If it is an actual callable it will be used directly - otherwise, if you specify a string (as in the example) the actual callable will be located using normal import mechanisms. The callable will be called with the remaining items in the configuration sub-dictionary as keyword arguments. In the above example, the formatter with id `custom` will be assumed to be returned by the call:

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

The key `()` has been used as the special key because it is not a valid keyword parameter name, and so will not clash with the names of the keyword arguments used in the call. The `()` also serves as a mnemonic that the corresponding value is a callable.

### Access to external objects

There are times where a configuration needs to refer to objects external to the configuration, for example `sys.stderr`. If the configuration dict is constructed using Python code, this is straightforward, but a problem arises when the configuration is provided via a text file (e.g. JSON, YAML). In a text file, there is no standard way to distinguish `sys.stderr` from the literal string `sys.stderr`. To facilitate this distinction, the configuration system looks for certain special prefixes in string values and treat them specially. For example, if the literal string `ext://sys.stderr` is provided as a value in the configuration, then the `ext://` will be stripped off and the remainder of the value processed using normal import mechanisms.

The handling of such prefixes is done in a way analogous to protocol handling: there is a generic mechanism to look for prefixes which match the regular expression `^(?P<prefix>[a-z]+)://(?P<suffix>.*$)` whereby, if the prefix is recognised, the suffix is processed in a prefix-dependent manner and the result of the processing replaces the string value. If the prefix is not recognised, then the string value will be left as-is.
Access to internal objects

As well as external objects, there is sometimes also a need to refer to objects in the configuration. This will be done implicitly by the configuration system for things that it knows about. For example, the string value ‘DEBUG’ for a level in a logger or handler will automatically be converted to the value logging.DEBUG, and the handlers, filters and formatter entries will take an object id and resolve to the appropriate destination object.

However, a more generic mechanism is needed for user-defined objects which are not known to the logging module. For example, consider logging.handlers.MemoryHandler, which takes a target argument which is another handler to delegate to. Since the system already knows about this class, then in the configuration, the given target just needs to be the object id of the relevant target handler, and the system will resolve to the handler from the id. If, however, a user defines a my.package.MyHandler which has an alternate handler, the configuration system would not know that the alternate referred to a handler. To cater for this, a generic resolution system allows the user to specify:

```
handlers:
  file:
    # configuration of file handler goes here
  custom:
    (): my.package.MyHandler
    alternate: cfg://handlers.file
```

The literal string ‘cfg://handlers.file’ will be resolved in an analogous way to strings with the ext:// prefix, but looking in the configuration itself rather than the import namespace. The mechanism allows access by dot or by index, in a similar way to that provided by str.format. Thus, given the following snippet:

```
handlers:
  email:
    class: logging.handlers.SMTPHandler
    mailhost: localhost
    fromaddr: my_app@domain.tld
    toaddrs:
      - support_team@domain.tld
      - dev_team@domain.tld
    subject: Houston, we have a problem.
```

in the configuration, the string ‘cfg://handlers’ would resolve to the dict with key handlers, the string ‘cfg://handlers.email’ would resolve to the dict with key email in the handlers dict, and so on. The string ‘cfg://handlers.email.toaddrs[1]’ would resolve to ‘dev_team@domain.tld’ and the string ‘cfg://handlers.email.toaddrs[0]’ would resolve to the value ‘support_team@domain.tld’. The subject value could be accessed using either ‘cfg://handlers.email.subject’ or, equivalently, ‘cfg://handlers.email[subject]’. The latter form only needs to be used if the key contains spaces or non-alphanumeric characters. If an index value consists only of decimal digits, access will be attempted using the corresponding integer value, falling back to the string value if needed.

Given a string ‘cfg://handlers.myhandler.mykey.123’, this will resolve to config_dict[‘handlers’][‘myhandler’][‘mykey’][123]. If the string is specified as ‘cfg://handlers.myhandler.mykey[123]’, the system will attempt to retrieve the value from config_dict[‘handlers’][‘myhandler’][‘mykey’][123], and fall back to config_dict[‘handlers’][‘myhandler’][‘mykey’][123] if that fails.

Import resolution and custom importers

Import resolution, by default, uses the built-in __import__() function to do its importing. You may want to replace this with your own importing mechanism: if so, you can replace the importer attribute of the DictConfigurator or its superclass, the BaseConfigurator class. However, you need to be careful because of the way functions are accessed from classes via descriptors. If you are using a Python callable to
do your imports, and you want to define it at class level rather than instance level, you need to wrap it with `staticmethod()`. For example:

```python
from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)
```

You don’t need to wrap with `staticmethod()` if you’re setting the import callable on a configurator `instance`.

### 16.8.3 Configuration file format

The configuration file format understood by `fileConfig()` is based on `configparser` functionality. The file must contain sections called `[loggers]`, `[handlers]` and `[formatters]` which identify by name the entities of each type which are defined in the file. For each such entity, there is a separate section which identifies how that entity is configured. Thus, for a logger named `log01` in the `[loggers]` section, the relevant configuration details are held in a section `[logger_log01]`. Similarly, a handler called `hand01` in the `[handlers]` section will have its configuration held in a section called `[handler_hand01]`. While a formatter called `form01` in the `[formatters]` section will have its configuration specified in a section called `[formatter_form01]`. The root logger configuration must be specified in a section called `[logger_root]`.

Examples of these sections in the file are given below.

```ini
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

The root logger must specify a level and a list of handlers. An example of a root logger section is given below.

```ini
[logger_root]
level=NOTSET
handlers=hand01
```

The `level` entry can be one of `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL` or `NOTSET`. For the root logger only, `NOTSET` means that all messages will be logged. Level values are `eval()`uated in the context of the logging package’s namespace.

The `handlers` entry is a comma-separated list of handler names, which must appear in the `[handlers]` section. These names must appear in the `[handlers]` section and have corresponding sections in the configuration file.

For loggers other than the root logger, some additional information is required. This is illustrated by the following example.

```ini
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

The `level` and `handlers` entries are interpreted as for the root logger, except that if a non-root logger’s level is specified as `NOTSET`, the system consults loggers higher up the hierarchy to determine the effective level of the logger. The `propagate` entry is set to 1 to indicate that messages must propagate to handlers higher up the logger hierarchy from this logger, or 0 to indicate that messages are **not** propagated to handlers up the hierarchy. The `qualname` entry is the hierarchical channel name of the logger, that is to say the name used by the application to get the logger.
Sections which specify handler configuration are exemplified by the following.

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

The class entry indicates the handler’s class (as determined by `eval()` in the logging package’s namespace). The level is interpreted as for loggers, and NOTSET is taken to mean ‘log everything’.

The formatter entry indicates the key name of the formatter for this handler. If blank, a default formatter (logging._defaultFormatter) is used. If a name is specified, it must appear in the [formatters] section and have a corresponding section in the configuration file.

The args entry, when evaluated in the context of the logging package’s namespace, is the list of arguments to the constructor for the handler class. Refer to the constructors for the relevant handlers, or to the examples below, to see how typical entries are constructed.

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')
```

```
[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)
```

```
[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)
```

```
[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=((('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)
```

```
[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')
```

```
[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')
```

```
[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)
```
Sections which specify formatter configuration are typified by the following.

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s
datefmt=
class=logging.Formatter
```

The `format` entry is the overall format string, and the `datefmt` entry is the `strftime()`-compatible date/time format string. If empty, the package substitutes ISO8601 format date/times, which is almost equivalent to specifying the date format string `'%Y-%m-%d %H:%M:%S'`. The ISO8601 format also specifies milliseconds, which are appended to the result of using the above format string, with a comma separator. An example time in ISO8601 format is `2003-01-23 00:29:50,411`.

The `class` entry is optional. It indicates the name of the formatter’s class (as a dotted module and class name.) This option is useful for instantiating a `Formatter` subclass. Subclasses of `Formatter` can present exception tracebacks in an expanded or condensed format.

**Note:** Due to the use of `eval()` as described above, there are potential security risks which result from using the `listen()` to send and receive configurations via sockets. The risks are limited to where multiple users with no mutual trust run code on the same machine; see the `listen()` documentation for more information.

---

**See Also:**

- Module `logging` API reference for the logging module.
- Module `logging.handlers` Useful handlers included with the logging module.

## 16.9 logging.handlers — Logging handlers

### Important

This page contains only reference information. For tutorials, please see

- Basic Tutorial
- Advanced Tutorial
- Logging Cookbook

**Source code:** Lib/logging/handlers.py

The following useful handlers are provided in the package. Note that three of the handlers (`StreamHandler`, `FileHandler` and `NullHandler`) are actually defined in the `logging` module itself, but have been documented here along with the other handlers.

### 16.9.1 StreamHandler

The `StreamHandler` class, located in the core `logging` package, sends logging output to streams such as `sys.stdout`, `sys.stderr` or any file-like object (or, more precisely, any object which supports `write()` and `flush()` methods).
class logging.StreamHandler (stream=None)
Returns a new instance of the StreamHandler class. If stream is specified, the instance will use it for logging output; otherwise, sys.stderr will be used.

    emit (record)
    If a formatter is specified, it is used to format the record. The record is then written to the stream with a terminator. If exception information is present, it is formatted using traceback.print_exception() and appended to the stream.

    flush()
    Flushes the stream by calling its flush() method. Note that the close() method is inherited from Handler and so does no output, so an explicit flush() call may be needed at times.

Changed in version 3.2: The StreamHandler class now has a terminator attribute, default value ‘\n’, which is used as the terminator when writing a formatted record to a stream. If you don’t want this newline termination, you can set the handler instance’s terminator attribute to the empty string. In earlier versions, the terminator was hardcoded as ‘\n’.

16.9.2 FileHandler
The FileHandler class, located in the core logging package, sends logging output to a disk file. It inherits the output functionality from StreamHandler.

class logging.FileHandler (filename, mode='a', encoding=None, delay=False)
Returns a new instance of the FileHandler class. The specified file is opened and used as the stream for logging. If mode is not specified, ‘a’ is used. If encoding is not None, it is used to open the file with that encoding. If delay is true, then file opening is deferred until the first call to emit(). By default, the file grows indefinitely.

    close()
    Closes the file.

    emit (record)
    Outputs the record to the file.

16.9.3 NullHandler
New in version 3.1. The NullHandler class, located in the core logging package, does not do any formatting or output. It is essentially a ‘no-op’ handler for use by library developers.

class logging.NullHandler
Returns a new instance of the NullHandler class.

    emit (record)
    This method does nothing.

    handle (record)
    This method does nothing.

    createLock()
    This method returns None for the lock, since there is no underlying I/O to which access needs to be serialized.

See library-config for more information on how to use NullHandler.

16.9.4 WatchedFileHandler

The WatchedFileHandler class, located in the logging.handlers module, is a FileHandler which watches the file it is logging to. If the file changes, it is closed and reopened using the file name.

A file change can happen because of usage of programs such as newsyslog and logrotate which perform log file rotation. This handler, intended for use under Unix/Linux, watches the file to see if it has changed since the last
emit. (A file is deemed to have changed if its device or inode have changed.) If the file has changed, the old file
stream is closed, and the file opened to get a new stream.

This handler is not appropriate for use under Windows, because under Windows open log files cannot be moved or
renamed - logging opens the files with exclusive locks - and so there is no need for such a handler. Furthermore,
ST_INO is not supported under Windows; stat() always returns zero for this value.

```python
class logging.handlers.WatchedFileHandler(filename[, mode[, encoding[, delay]]])
```

Returns a new instance of the WatchedFileHandler class. The specified file is opened and used as the
stream for logging. If mode is not specified, ‘a’ is used. If encoding is not None, it is used to open the file
with that encoding. If delay is true, then file opening is deferred until the first call to emit(). By default,
the file grows indefinitely.

```python
emit(record)
```

Outputs the record to the file, but first checks to see if the file has changed. If it has, the existing stream
is flushed and closed and the file opened again, before outputting the record to the file.

### 16.9.5 BaseRotatingHandler

The BaseRotatingHandler class, located in the logging.handlers module, is the base class for the
rotating file handlers, RotatingFileHandler and TimedRotatingFileHandler. You should not need
to instantiate this class, but it has attributes and methods you may need to override.

```python
class logging.handlers.BaseRotatingHandler(filename, mode, encoding=None, delay=False)
```

The parameters are as for FileHandler. The attributes are:

- **namer**: If this attribute is set to a callable, the rotation_filename() method delegates to this callable.
The parameters passed to the callable are those passed to rotation_filename().

  **Note:** The namer function is called quite a few times during rollover, so it should be as simple and
  as fast as possible. It should also return the same output every time for a given input, otherwise the
  rollover behaviour may not work as expected.

  New in version 3.3.

- **rotator**: If this attribute is set to a callable, the rotate() method delegates to this callable. The parameters
  passed to the callable are those passed to rotate(). New in version 3.3.

- **rotation_filename(default_name)**: Modify the filename of a log file when rotating.

  This is provided so that a custom filename can be provided.

  The default implementation calls the ‘namer’ attribute of the handler, if it’s callable, passing the default
  name to it. If the attribute isn’t callable (the default is None), the name is returned unchanged.

  **Parameters**

  - **default_name** – The default name for the log file.

  New in version 3.3.

- **rotate(source, dest)**: When rotating, rotate the current log.

  The default implementation calls the ‘rotator’ attribute of the handler, if it’s callable, passing the
  source and dest arguments to it. If the attribute isn’t callable (the default is None), the source is
  simply renamed to the destination.

  **Parameters**

  - **source** – The source filename. This is normally the base filename, e.g. ‘test.log’
• **dest** – The destination filename. This is normally what the source is rotated to, e.g. 'test.log.1'.

    New in version 3.3.

The reason the attributes exist is to save you having to subclass - you can use the same callables for instances of `RotatingFileHandler` and `TimedRotatingFileHandler`. If either the namer or rotator callable raises an exception, this will be handled in the same way as any other exception during an `emit()` call, i.e. via the `handleError()` method of the handler.

If you need to make more significant changes to rotation processing, you can override the methods.

For an example, see `cookbook-rotator-namer`.

### 16.9.6 RotatingFileHandler

The `RotatingFileHandler` class, located in the `logging.handlers` module, supports rotation of disk log files.

```python
class logging.handlers.RotatingFileHandler (filename, mode='a', maxBytes=0, backupCount=0, encoding=None, delay=0)
```

Returns a new instance of the `RotatingFileHandler` class. The specified file is opened and used as the stream for logging. If `mode` is not specified, `'a'` is used. If `encoding` is not `None`, it is used to open the file with that encoding. If `delay` is true, then file opening is deferred until the first call to `emit()`. By default, the file grows indefinitely.

You can use the `maxBytes` and `backupCount` values to allow the file to rollover at a predetermined size. When the size is about to be exceeded, the file is closed and a new file is silently opened for output. Rollover occurs whenever the current log file is nearly `maxBytes` in length; if `maxBytes` is zero, rollover never occurs. If `backupCount` is non-zero, the system will save old log files by appending the extensions `.1`, `.2` etc., to the filename. For example, with a `backupCount` of 5 and a base file name of `app.log`, you would get `app.log`, `app.log.1`, `app.log.2`, up to `app.log.5`. The file being written to is always `app.log`. When this file is filled, it is closed and renamed to `app.log.1`, and if files `app.log.1`, `app.log.2`, etc. exist, then they are renamed to `app.log.2`, `app.log.3` etc. respectively.

```python
doRollover ()
```

Does a rollover, as described above.

```python
emit (record)
```

Outputs the record to the file, catering for rollover as described previously.

### 16.9.7 TimedRotatingFileHandler

The `TimedRotatingFileHandler` class, located in the `logging.handlers` module, supports rotation of disk log files at certain timed intervals.

```python
class logging.handlers.TimedRotatingFileHandler (filename, when='h', interval=1, backupCount=0, encoding=None, delay=False, utc=False)
```

Returns a new instance of the `TimedRotatingFileHandler` class. The specified file is opened and used as the stream for logging. On rotating it also sets the filename suffix. Rotating happens based on the product of `when` and `interval`.

You can use the `when` to specify the type of `interval`. The list of possible values is below. Note that they are not case sensitive.

<table>
<thead>
<tr>
<th>Value</th>
<th>Type of interval</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>'S'</code></td>
<td>Seconds</td>
</tr>
<tr>
<td><code>'M'</code></td>
<td>Minutes</td>
</tr>
<tr>
<td><code>'H'</code></td>
<td>Hours</td>
</tr>
<tr>
<td><code>'D'</code></td>
<td>Days</td>
</tr>
<tr>
<td><code>'W0'-'W6'</code></td>
<td>Weekday (0=Monday)</td>
</tr>
<tr>
<td><code>'midnight'</code></td>
<td>Roll over at midnight</td>
</tr>
</tbody>
</table>
When using weekday-based rotation, specify ‘W0’ for Monday, ‘W1’ for Tuesday, and so on up to ‘W6’ for Sunday. In this case, the value passed for interval isn’t used.

The system will save old log files by appending extensions to the filename. The extensions are date-and-time based, using the strftime format %Y-%m-%d_%H-%M-%S or a leading portion thereof, depending on the rollover interval.

When computing the next rollover time for the first time (when the handler is created), the last modification time of an existing log file, or else the current time, is used to compute when the next rotation will occur.

If the utc argument is true, times in UTC will be used; otherwise local time is used.

If backupCount is nonzero, at most backupCount files will be kept, and if more would be created when rollover occurs, the oldest one is deleted. The deletion logic uses the interval to determine which files to delete, so changing the interval may leave old files lying around.

If delay is true, then file opening is deferred until the first call to emit().

```
16.9.8 SocketHandler
```

The SocketHandler class, located in the logging.handlers module, sends logging output to a network socket. The base class uses a TCP socket.

```python
class logging.handlers.SocketHandler(host, port):
    Returns a new instance of the SocketHandler class intended to communicate with a remote machine whose address is given by host and port.

close()
    Closes the socket.

emit(record)
    Pickles the record’s attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. If the connection was previously lost, re-establishes the connection. To unpickle the record at the receiving end into a LogRecord, use the makeLogRecord() function.

handleError()
    Handles an error which has occurred during emit(). The most likely cause is a lost connection. Closes the socket so that we can retry on the next event.

makeSocket()
    This is a factory method which allows subclasses to define the precise type of socket they want. The default implementation creates a TCP socket (socket.SOCK_STREAM).

makePickle(record)
    Pickles the record’s attribute dictionary in binary format with a length prefix, and returns it ready for transmission across the socket.

    Note that pickles aren’t completely secure. If you are concerned about security, you may want to override this method to implement a more secure mechanism. For example, you can sign pickles using HMAC and then verify them on the receiving end, or alternatively you can disable unpickling of global objects on the receiving end.

send(packet)
    Send a pickled string packet to the socket. This function allows for partial sends which can happen when the network is busy.

createSocket()
    Tries to create a socket; on failure, uses an exponential back-off algorithm. On intial failure, the
The Python Library Reference, Release 3.3.3

handler will drop the message it was trying to send. When subsequent messages are handled by the same instance, it will not try connecting until some time has passed. The default parameters are such that the initial delay is one second, and if after that delay the connection still can’t be made, the handler will double the delay each time up to a maximum of 30 seconds.

This behaviour is controlled by the following handler attributes:

• retryStart (initial delay, defaulting to 1.0 seconds).
• retryFactor (multiplier, defaulting to 2.0).
• retryMax (maximum delay, defaulting to 30.0 seconds).

This means that if the remote listener starts up after the handler has been used, you could lose messages (since the handler won’t even attempt a connection until the delay has elapsed, but just silently drop messages during the delay period).

16.9.9 DatagramHandler

The DatagramHandler class, located in the logging.handlers module, inherits from SocketHandler to support sending logging messages over UDP sockets.

class logging.handlers.DatagramHandler (host, port)

Returns a new instance of the DatagramHandler class intended to communicate with a remote machine whose address is given by host and port.

emit ()

Pickles the record’s attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. To unpickle the record at the receiving end into a LogRecord, use the makeLogRecord() function.

makeSocket ()

The factory method of SocketHandler is here overridden to create a UDP socket (socket.SOCK_DGRAM).

send(s)

Send a pickled string to a socket.

16.9.10 SysLogHandler

The SysLogHandler class, located in the logging.handlers module, supports sending logging messages to a remote or local Unix syslog.

class logging.handlers.SysLogHandler (address=('localhost', SYSLOG_UDP_PORT), facility=LOG_USER, socktype=socket.SOCK_DGRAM)

Returns a new instance of the SysLogHandler class intended to communicate with a remote Unix machine whose address is given by address in the form of a (host, port) tuple. If address is not specified, ('localhost', 514) is used. The address is used to open a socket. An alternative to providing a (host, port) tuple is providing an address as a string, for example '/dev/log'. In this case, a Unix domain socket is used to send the message to the syslog. If facility is not specified, LOG_USER is used. The type of socket opened depends on the socktype argument, which defaults to socket.SOCK_DGRAM and thus opens a UDP socket. To open a TCP socket (for use with the newer syslog daemons such as rsyslog), specify a value of socket.SOCK_STREAM.

Note that if your server is not listening on UDP port 514, SysLogHandler may appear not to work. In that case, check what address you should be using for a domain socket - it’s system dependent. For example, on Linux it’s usually '/dev/log' but on OS/X it’s '/var/run/syslog'. You’ll need to check your platform and use the appropriate address (you may need to do this check at runtime if your application needs to run on several platforms). On Windows, you pretty much have to use the UDP option. Changed in version 3.2: socktype was added.

close ()

Closes the socket to the remote host.

16.9. logging.handlers — Logging handlers
emit (record)

The record is formatted, and then sent to the syslog server. If exception information is present, it is not sent to the server. Changed in version 3.2.1: (See: issue 12168.) In earlier versions, the message sent to the syslog daemons was always terminated with a NUL byte, because early versions of these daemons expected a NUL terminated message - even though it’s not in the relevant specification (RF 5424). More recent versions of these daemons don’t expect the NUL byte but strip it off if it’s there, and even more recent daemons (which adhere more closely to RFC 5424) pass the NUL byte on as part of the message. To enable easier handling of syslog messages in the face of all these differing daemon behaviours, the appending of the NUL byte has been made configurable, through the use of a class-level attribute, append_nul. This defaults to True (preserving the existing behaviour) but can be set to False on a SysLogHandler instance in order for that instance to not append the NUL terminator. Changed in version 3.3: (See: issue 12419.) In earlier versions, there was no facility for an “ident” or “tag” prefix to identify the source of the message. This can now be specified using a class-level attribute, defaulting to "" to preserve existing behaviour, but which can be overridden on a SysLogHandler instance in order for that instance to prepend the ident to every message handled. Note that the provided ident must be text, not bytes, and is prepended to the message exactly as is.

encodePriority (facility, priority)

Encodes the facility and priority into an integer. You can pass in strings or integers - if strings are passed, internal mapping dictionaries are used to convert them to integers.

The symbolic LOG_ values are defined in SysLogHandler and mirror the values defined in the sys/syslog.h header file.

<table>
<thead>
<tr>
<th>Priorities</th>
<th>Symbolic value</th>
</tr>
</thead>
<tbody>
<tr>
<td>alert</td>
<td>LOG_ALERT</td>
</tr>
<tr>
<td>crit or critical</td>
<td>LOG_CRIT</td>
</tr>
<tr>
<td>debug</td>
<td>LOG_DEBUG</td>
</tr>
<tr>
<td>emerg or panic</td>
<td>LOG_EMERG</td>
</tr>
<tr>
<td>err or error</td>
<td>LOG_ERR</td>
</tr>
<tr>
<td>info</td>
<td>LOG_INFO</td>
</tr>
<tr>
<td>notice</td>
<td>LOG_NOTICE</td>
</tr>
<tr>
<td>warn or warning</td>
<td>LOG_WARNING</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Facilities</th>
<th>Symbolic value</th>
</tr>
</thead>
<tbody>
<tr>
<td>auth</td>
<td>LOG_AUTH</td>
</tr>
<tr>
<td>authpriv</td>
<td>LOG_AUTHPRIV</td>
</tr>
<tr>
<td>cron</td>
<td>LOG_CRON</td>
</tr>
<tr>
<td>daemon</td>
<td>LOG_DAEMON</td>
</tr>
<tr>
<td>ftp</td>
<td>LOG_FTP</td>
</tr>
<tr>
<td>kern</td>
<td>LOG_KERN</td>
</tr>
<tr>
<td>lpr</td>
<td>LOG_LPR</td>
</tr>
<tr>
<td>mail</td>
<td>LOG_MAIL</td>
</tr>
<tr>
<td>news</td>
<td>LOG_NEWS</td>
</tr>
<tr>
<td>syslog</td>
<td>LOG_SYSLOG</td>
</tr>
<tr>
<td>user</td>
<td>LOG_USER</td>
</tr>
<tr>
<td>uucp</td>
<td>LOG_UUCP</td>
</tr>
<tr>
<td>local0</td>
<td>LOG_LOCAL0</td>
</tr>
<tr>
<td>local1</td>
<td>LOG_LOCAL1</td>
</tr>
<tr>
<td>local2</td>
<td>LOG_LOCAL2</td>
</tr>
<tr>
<td>local3</td>
<td>LOG_LOCAL3</td>
</tr>
<tr>
<td>local4</td>
<td>LOG_LOCAL4</td>
</tr>
<tr>
<td>local5</td>
<td>LOG_LOCAL5</td>
</tr>
<tr>
<td>local6</td>
<td>LOG_LOCAL6</td>
</tr>
<tr>
<td>local7</td>
<td>LOG_LOCAL7</td>
</tr>
</tbody>
</table>

mapPriority (levelname)

Maps a logging level name to a syslog priority name. You may need to override this if you are using
custom levels, or if the default algorithm is not suitable for your needs. The default algorithm maps DEBUG, INFO, WARNING, ERROR and CRITICAL to the equivalent syslog names, and all other level names to ‘warning’.

16.9.11 NTEventLogHandler

The NTEventLogHandler class, located in the logging.handlers module, supports sending logging messages to a local Windows NT, Windows 2000 or Windows XP event log. Before you can use it, you need Mark Hammond’s Win32 extensions for Python installed.

class logging.handlers.NTEventLogHandler (appname, dllname=None, logtype='Application')
Returns a new instance of the NTEventLogHandler class. The appname is used to define the application name as it appears in the event log. An appropriate registry entry is created using this name. The dllname should give the fully qualified pathname of a .dll or .exe which contains message definitions to hold in the log (if not specified, ‘win32service.pyd’ is used - this is installed with the Win32 extensions and contains some basic placeholder message definitions. Note that use of these placeholders will make your event logs big, as the entire message source is held in the log. If you want slimmer logs, you have to pass in the name of your own .dll or .exe which contains the message definitions you want to use in the event log). The logtype is one of ‘Application’, ‘System’ or ‘Security’, and defaults to ‘Application’.

close()
At this point, you can remove the application name from the registry as a source of event log entries. However, if you do this, you will not be able to see the events as you intended in the Event Log Viewer - it needs to be able to access the registry to get the .dll name. The current version does not do this.

edit (record)
Determines the message ID, event category and event type, and then logs the message in the NT event log.

getEventCategory (record)
Returns the event category for the record. Override this if you want to specify your own categories. This version returns 0.

getEventType (record)
Returns the event type for the record. Override this if you want to specify your own types. This version does a mapping using the handler’s typemap attribute, which is set up in __init__() to a dictionary which contains mappings for DEBUG, INFO, WARNING, ERROR and CRITICAL. If you are using your own levels, you will either need to override this method or place a suitable dictionary in the handler’s typemap attribute.

getMessageID (record)
Returns the message ID for the record. If you are using your own messages, you could do this by having the msg passed to the logger being an ID rather than a format string. Then, in here, you could use a dictionary lookup to get the message ID. This version returns 1, which is the base message ID in win32service.pyd.

16.9.12 SMTPHandler

The SMTPHandler class, located in the logging.handlers module, supports sending logging messages to an email address via SMTP.

class logging.handlers.SMTPHandler (mailhost, fromaddr, toaddrs, subject, credentials=None, secure=None, timeout=1.0)
Returns a new instance of the SMTPHandler class. The instance is initialized with the from and to addresses and subject line of the email. The toaddrs should be a list of strings. To specify a non-standard SMTP port, use the (host, port) tuple format for the mailhost argument. If you use a string, the standard SMTP port is used. If your SMTP server requires authentication, you can specify a (username, password) tuple for the credentials argument.
To specify the use of a secure protocol (TLS), pass in a tuple to the `secure` argument. This will only be used when authentication credentials are supplied. The tuple should be either an empty tuple, or a single-value tuple with the name of a keyfile, or a 2-value tuple with the names of the keyfile and certificate file. (This tuple is passed to the `smtplib.SMTP.starttls()` method.)

A timeout can be specified for communication with the SMTP server using the `timeout` argument. New in version 3.3: The `timeout` argument was added.

```python
def emit(record):
    Formats the record and sends it to the specified addressees.

def getSubject(record):
    If you want to specify a subject line which is record-dependent, override this method.
```

### 16.9.13 MemoryHandler

The `MemoryHandler` class, located in the `logging.handlers` module, supports buffering of logging records in memory, periodically flushing them to a target handler. flushing occurs whenever the buffer is full, or when an event of a certain severity or greater is seen.

`MemoryHandler` is a subclass of the more general `BufferingHandler`, which is an abstract class. This buffers logging records in memory. Whenever each record is added to the buffer, a check is made by calling `shouldFlush()` to see if the buffer should be flushed. If it should, then `flush()` is expected to do the flushing.

```python
class logging.handlers.BufferingHandler(capacity):
    Initializes the handler with a buffer of the specified capacity.

    def emit(record):
        Appends the record to the buffer. If `shouldFlush()` returns true, calls `flush()` to process the buffer.

    def flush()
        You can override this to implement custom flushing behavior. This version just zaps the buffer to empty.

    def shouldFlush(record)
        Checks for buffer full or a record at the `flushLevel` or higher.
```

```python
class logging.handlers.MemoryHandler(capacity, flushLevel=ERROR, target=None):
    Returns a new instance of the `MemoryHandler` class. The instance is initialized with a buffer size of `capacity`. If `flushLevel` is not specified, `ERROR` is used. If no `target` is specified, the target will need to be set using `setTarget()` before this handler does anything useful.

    def close()
        Calls `flush()`, sets the target to `None` and clears the buffer.

    def flush()
        For a `MemoryHandler`, flushing means just sending the buffered records to the target, if there is one. The buffer is also cleared when this happens. Override if you want different behavior.

    def setTarget(target)
        Sets the target handler for this handler.

    def shouldFlush(record)
        Checks for buffer full or a record at the `flushLevel` or higher.
```

### 16.9.14 HTTPHandler

The `HTTPHandler` class, located in the `logging.handlers` module, supports sending logging messages to a Web server, using either `GET` or `POST` semantics.
class `logging.handlers.HTTPHandler`(
    `host`, `url`, `method='GET'`, `secure=False`, `credentials=None`
)

Returns a new instance of the `HTTPHandler` class. The `host` can be of the form `host:port`, should you need to use a specific port number. If no `method` is specified, `GET` is used. If `secure` is True, an HTTPS connection will be used. If `credentials` is specified, it should be a 2-tuple consisting of userid and password, which will be placed in an HTTP ‘Authorization’ header using Basic authentication. If you specify credentials, you should also specify `secure=True` so that your userid and password are not passed in cleartext across the wire.

**emit** *(record)*

Sends the record to the Web server as a percent-encoded dictionary.

### 16.9.15 QueueHandler

New in version 3.2. The `QueueHandler` class, located in the `logging.handlers` module, supports sending logging messages to a queue, such as those implemented in the `queue` or `multiprocessing` modules.

Along with the `QueueListener` class, `QueueHandler` can be used to let handlers do their work on a separate thread from the one which does the logging. This is important in Web applications and also other service applications where threads servicing clients need to respond as quickly as possible, while any potentially slow operations (such as sending an email via `SMTPHandler`) are done on a separate thread.

class `logging.handlers.QueueHandler` *(queue)*

Returns a new instance of the `QueueHandler` class. The instance is initialized with the queue to send messages to. The queue can be any queue-like object; it’s used as-is by the `enqueue()` method, which needs to know how to send messages to it.

**emit** *(record)*

Enqueues the result of preparing the LogRecord.

**prepare** *(record)*

Prepares a record for queuing. The object returned by this method is enqueued.

The base implementation formats the record to merge the message and arguments, and removes unpickable items from the record in-place.

You might want to override this method if you want to convert the record to a dict or JSON string, or send a modified copy of the record while leaving the original intact.

**enqueue** *(record)*

Enqueues the record on the queue using `put_nowait()`; you may want to override this if you want to use blocking behaviour, or a timeout, or a customised queue implementation.

### 16.9.16 QueueListener

New in version 3.2. The `QueueListener` class, located in the `logging.handlers` module, supports receiving logging messages from a queue, such as those implemented in the `queue` or `multiprocessing` modules. The messages are received from a queue in an internal thread and passed, on the same thread, to one or more handlers for processing. While `QueueListener` is not itself a handler, it is documented here because it works hand-in-hand with `QueueHandler`.

Along with the `QueueHandler` class, `QueueListener` can be used to let handlers do their work on a separate thread from the one which does the logging. This is important in Web applications and also other service applications where threads servicing clients need to respond as quickly as possible, while any potentially slow operations (such as sending an email via `SMTPHandler`) are done on a separate thread.

class `logging.handlers.QueueListener` *(queue, *handlers)*

Returns a new instance of the `QueueListener` class. The instance is initialized with the queue to send messages to and a list of handlers which will handle entries placed on the queue. The queue can be any queue-like object; it’s passed as-is to the `dequeue()` method, which needs to know how to get messages from it.
The Python Library Reference, Release 3.3.3

**dequeue**(block)

Dequeues a record and return it, optionally blocking.

The base implementation uses `get()`. You may want to override this method if you want to use timeouts or work with custom queue implementations.

**prepare**(record)

Prepare a record for handling.

This implementation just returns the passed-in record. You may want to override this method if you need to do any custom marshalling or manipulation of the record before passing it to the handlers.

**handle**(record)

Handle a record.

This just loops through the handlers offering them the record to handle. The actual object passed to the handlers is that which is returned from `prepare()`.

**start**()

Starts the listener.

This starts up a background thread to monitor the queue for LogRecords to process.

**stop**()

Stops the listener.

This asks the thread to terminate, and then waits for it to do so. Note that if you don’t call this before your application exits, there may be some records still left on the queue, which won’t be processed.

**enqueue_sentinel**()

Writes a sentinel to the queue to tell the listener to quit. This implementation uses `put_nowait()`.

You may want to override this method if you want to use timeouts or work with custom queue implementations. New in version 3.3.

See Also:

- Module `logging` API reference for the logging module.
- Module `logging.config` Configuration API for the logging module.

### 16.10 `getpass` — Portable password input

The `getpass` module provides two functions:

**getpass.getpass**(prompt='Password: ', stream=None)

Prompt the user for a password without echoing. The user is prompted using the string `prompt`, which defaults to ‘Password: ’. On Unix, the prompt is written to the file-like object `stream`. `stream` defaults to the controlling terminal (`/dev/tty`) or if that is unavailable to `sys.stderr` (this argument is ignored on Windows).

If echo free input is unavailable `getpass()` falls back to printing a warning message to `stream` and reading from `sys.stdin` and issuing a `GetPassWarning`.

Availability: Macintosh, Unix, Windows.

**Note:** If you call `getpass` from within IDLE, the input may be done in the terminal you launched IDLE from rather than the idle window itself.

**exception `getpass.GetPassWarning`**

A `UserWarning` subclass issued when password input may be echoed.

**getpass.getuser**()

Return the “login name” of the user. Availability: Unix, Windows.

This function checks the environment variables `LOGNAME`, `USER`, and `USERNAME`. New in version 3.3.
USER, LNAME and USERNAME, in order, and returns the value of the first one which is set to a non-empty string. If none are set, the login name from the password database is returned on systems which support the `pwd` module, otherwise, an exception is raised.

## 16.11 curses — Terminal handling for character-cell displays

### Platforms: Unix

The `curses` module provides an interface to the curses library, the de-facto standard for portable advanced terminal handling.

While curses is most widely used in the Unix environment, versions are available for DOS, OS/2, and possibly other systems as well. This extension module is designed to match the API of ncurses, an open-source curses library hosted on Linux and the BSD variants of Unix.

**Note:** Since version 5.4, the ncurses library decides how to interpret non-ASCII data using the `nl_langinfo` function. That means that you have to call `locale.setlocale()` in the application and encode Unicode strings using one of the system’s available encodings. This example uses the system’s default encoding:

```python
import locale
locale.setlocale(locale.LC_ALL, '')
code = locale.getpreferredencoding()
```

Then use `code` as the encoding for `str.encode()` calls.

### See Also:

- **Module** `curses.ascii` Utilities for working with ASCII characters, regardless of your locale settings.
- **Module** `curses.panel` A panel stack extension that adds depth to curses windows.
- **Module** `curses.textpad` Editable text widget for curses supporting Emacs-like bindings.
- `curses-howto` Tutorial material on using curses with Python, by Andrew Kuchling and Eric Raymond.

The `Tools/demo/` directory in the Python source distribution contains some example programs using the curses bindings provided by this module.

### 16.11.1 Functions

The module `curses` defines the following exception:

**exception** `curses.error` Exception raised when a curses library function returns an error.

**Note:** Whenever `x` or `y` arguments to a function or a method are optional, they default to the current cursor location. Whenever `attr` is optional, it defaults to `A_NORMAL`.

The module `curses` defines the following functions:

- **curses.baudrate()**
  
  Return the output speed of the terminal in bits per second. On software terminal emulators it will have a fixed high value. Included for historical reasons; in former times, it was used to write output loops for time delays and occasionally to change interfaces depending on the line speed.

- **curses.beep()**
  
  Emit a short attention sound.

- **curses.can_change_color()**
  
  Return `True` or `False`, depending on whether the programmer can change the colors displayed by the terminal.
Enter cbreak mode. In cbreak mode (sometimes called “rare” mode) normal tty line buffering is turned off and characters are available to be read one by one. However, unlike raw mode, special characters (interrupt, quit, suspend, and flow control) retain their effects on the tty driver and calling program. Calling first raw() then cbreak() leaves the terminal in cbreak mode.

curses.color_content(color_number)

Return the intensity of the red, green, and blue (RGB) components in the color color_number, which must be between 0 and COLORS. A 3-tuple is returned, containing the R,G,B values for the given color, which will be between 0 (no component) and 1000 (maximum amount of component).

curses.color_pair(color_number)

Return the attribute value for displaying text in the specified color. This attribute value can be combined with A_STANDOUT, A_REVERSE, and the other A_* attributes. pair_number() is the counterpart to this function.

curses.curs_set(visibility)

Set the cursor state. visibility can be set to 0, 1, or 2, for invisible, normal, or very visible. If the terminal supports the visibility requested, the previous cursor state is returned; otherwise, an exception is raised. On many terminals, the “visible” mode is an underline cursor and the “very visible” mode is a block cursor.

curses.def_prog_mode()

Save the current terminal mode as the “program” mode, the mode when the running program is using curses. (Its counterpart is the “shell” mode, for when the program is not in curses.) Subsequent calls to reset_prog_mode() will restore this mode.

curses.def_shell_mode()

Save the current terminal mode as the “shell” mode, the mode when the running program is not using curses. (Its counterpart is the “program” mode, when the program is using curses capabilities.) Subsequent calls to reset_shell_mode() will restore this mode.

curses.delay_output(ms)

Insert an ms millisecond pause in output.

curses.doupdate()

Update the physical screen. The curses library keeps two data structures, one representing the current physical screen contents and a virtual screen representing the desired next state. The doupdate() ground updates the physical screen to match the virtual screen.

The virtual screen may be updated by a noutrefresh() call after write operations such as addstr() have been performed on a window. The normal refresh() call is simply noutrefresh() followed by doupdate(); if you have to update multiple windows, you can speed performance and perhaps reduce screen flicker by issuing noutrefresh() calls on all windows, followed by a single doupdate().

curses.echo()

Enter echo mode. In echo mode, each character input is echoed to the screen as it is entered.

curses.endwin()

De-initialize the library, and return terminal to normal status.

curses.erasechar()

Return the user’s current erase character. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses.filter()

The filter() routine, if used, must be called before initscr() is called. The effect is that, during those calls, LINES is set to 1; the capabilities clear, cup, cud, cud1, cuu1, cuu, vpa are disabled; and the home string is set to the value of cr. The effect is that the cursor is confined to the current line, and so are screen updates. This may be used for enabling character-at-a-time line editing without touching the rest of the screen.

curses.flash()

Flash the screen. That is, change it to reverse-video and then change it back in a short interval. Some people prefer such as ‘visible bell’ to the audible attention signal produced by beep().
curses.flushinp()
Flush all input buffers. This throws away any typeahead that has been typed by the user and has not yet been processed by the program.

curses.getmouse()
After getch() returns KEY_MOUSE to signal a mouse event, this method should be call to retrieve the queued mouse event, represented as a 5-tuple (id, x, y, z, bstate). id is an ID value used to distinguish multiple devices, and x, y, z are the event’s coordinates. (z is currently unused.)
bstate is an integer value whose bits will be set to indicate the type of event, and will be the bitwise OR of one or more of the following constants, where n is the button number from 1 to 4: BUTTONn_PRESSED, BUTTONnRELEASED, BUTTONn_CLICKED, BUTTONn_DOUBLE_CLICKED, BUTTONn_TRIPLE_CLICKED, BUTTON_SHIFT, BUTTON_CTRL, BUTTON_ALT.

curses.getsyx()
Return the current coordinates of the virtual screen cursor in y and x. If leaveok is currently true, then -1,-1 is returned.

curses.getwin(file)
Read window related data stored in the file by an earlier putwin() call. The routine then creates and initializes a new window using that data, returning the new window object.

curses.has_colors()
Return True if the terminal can display colors; otherwise, return False.

curses.has_ic()
Return True if the terminal has insert- and delete-character capabilities. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

curses.has_il()
Return True if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

curses.has_key(ch)
Take a key value ch, and return True if the current terminal type recognizes a key with that value.

curses.halfdelay(tenths)
Used for half-delay mode, which is similar to cbreak mode in that characters typed by the user are immediately available to the program. However, after blocking for tenths tenths of seconds, an exception is raised if nothing has been typed. The value of tenths must be a number between 1 and 255. Use nocbreak() to leave half-delay mode.

curses.init_color(color_number, r, g, b)
Change the definition of a color, taking the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of color_number must be between 0 and COLORS. Each of r, g, b, must be a value between 0 and 1000. When init_color() is used, all occurrences of that color on the screen immediately change to the new definition. This function is a no-op on most terminals; it is active only if can_change_color() returns 1.

curses.init_pair(pair_number, fg, bg)
Change the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of pair_number must be between 1 and COLOR_PAIRS - 1 (the 0 color pair is wired to white on black and cannot be changed). The value of fg and bg arguments must be between 0 and COLORS. If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair are changed to the new definition.

curses.initscr()
Initialize the library. Return a WindowObject which represents the whole screen.

Note: If there is an error opening the terminal, the underlying curses library may cause the interpreter to exit.
curses.is_term_resized(nlines, ncols)
    Return True if resize_term() would modify the window structure, False otherwise.

curses.isendwin()
    Return True if endwin() has been called (that is, the curses library has been deinitialized).

curses.keyname(k)
    Return the name of the key numbered k. The name of a key generating printable ASCII character is the key’s character. The name of a control-key combination is a two-character string consisting of a caret followed by the corresponding printable ASCII character. The name of an alt-key combination (128-255) is a string consisting of the prefix ‘M-‘ followed by the name of the corresponding ASCII character.

curses.killchar()
    Return the user’s current line kill character. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses.longname()
    Return a string containing the terminfo long name field describing the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to initscr().

curses.meta(yes)
    If yes is 1, allow 8-bit characters to be input. If yes is 0, allow only 7-bit chars.

curses.mouseinterval(interval)
    Set the maximum time in milliseconds that can elapse between press and release events in order for them to be recognized as a click, and return the previous interval value. The default value is 200 msec, or one fifth of a second.

curses.mousemask(mousemask)
    Set the mouse events to be reported, and return a tuple (availmask, oldmask). availmask indicates which of the specified mouse events can be reported; on complete failure it returns 0. oldmask is the previous value of the given window’s mouse event mask. If this function is never called, no mouse events are ever reported.

curses.napms(ms)
    Sleep for ms milliseconds.

curses.newpad(nlines, ncols)
    Create and return a pointer to a new pad data structure with the given number of lines and columns. A pad is returned as a window object.

    A pad is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (such as from scrolling or echoing of input) do not occur. The refresh() and noutrefresh() methods of a pad require 6 arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display. The arguments are pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol; the p arguments refer to the upper left corner of the pad region to be displayed and the s arguments define a clipping box on the screen within which the pad region is to be displayed.

curses.newwin(begin_y, begin_x)

curses.newwin(nlines, ncols, begin_y, begin_x)
    Return a new window, whose left-upper corner is at (begin_y, begin_x), and whose height/width is nlines/ncols.

    By default, the window will extend from the specified position to the lower right corner of the screen.

curses.nl()
    Enter newline mode. This mode translates the return key into newline on input, and translates newline into return and line-feed on output. Newline mode is initially on.

curses.nocbreak()
    Leave cbreak mode. Return to normal “cooked” mode with line buffering.

curses.noecho()
    Leave echo mode. Echoing of input characters is turned off.
curses.nonl()
Leaves newline mode. Disable translation of return into newline on input, and disable low-level translation of newline into newline/return on output (but this does not change the behavior of addch(‘\n’), which always does the equivalent of return and line feed on the virtual screen). With translation off, curses can sometimes speed up vertical motion a little; also, it will be able to detect the return key on input.

curses.noqiflush()
When the noqiflush() routine is used, normal flush of input and output queues associated with the INTR, QUIT and SUSP characters will not be done. You may want to call noqiflush() in a signal handler if you want output to continue as though the interrupt had not occurred, after the handler exits.

curses.noraw()
Leaves raw mode. Return to normal “cooked” mode with line buffering.

curses.pair_content(pair_number)
Return a tuple (fg, bg) containing the colors for the requested color pair. The value of pair_number must be between 1 and COLOR_PAIRS - 1.

curses.pair_number(attr)
Return the number of the color-pair set by the attribute value attr. color_pair() is the counterpart to this function.

curses.putp(string)
Equivalent to tputs(str, 1, putchar); emit the value of a specified terminfo capability for the current terminal. Note that the output of putp() always goes to standard output.

curses.qiflush(flag)
If flag is False, the effect is the same as calling noqiflush(). If flag is True, or no argument is provided, the queues will be flushed when these control characters are read.

curses.raw()
Enter raw mode. In raw mode, normal line buffering and processing of interrupt, quit, suspend, and flow control keys are turned off; characters are presented to curses input functions one by one.

curses.reset_prog_mode()
Restore the terminal to “program” mode, as previously saved by def_prog_mode().

curses.reset_shell_mode()
Restore the terminal to “shell” mode, as previously saved by def_shell_mode().

curses.resetty()
Restore the state of the terminal modes to what it was at the last call to savetty().

curses.resize_term(nlines, ncols)
Backend function used by resizeterm(), performing most of the work; when resizing the windows, resize_term() blank-fills the areas that are extended. The calling application should fill in these areas with appropriate data. The resize_term() function attempts to resize all windows. However, due to the calling convention of pads, it is not possible to resize these without additional interaction with the application.

curses.resizeterm(nlines, ncols)
Resize the standard and current windows to the specified dimensions, and adjusts other bookkeeping data used by the curses library that record the window dimensions (in particular the SIGWINCH handler).

curses.savetty()
Save the current state of the terminal modes in a buffer, usable by resetty().

curses.setsyx(y, x)
Set the virtual screen cursor to y, x. If y and x are both -1, then leaveok is set.

curses.setupterm([termstr, fd])
Initialize the terminal. termstr is a string giving the terminal name; if omitted, the value of the TERM environment variable will be used. fd is the file descriptor to which any initialization sequences will be sent; if not supplied, the file descriptor for sys.stdout will be used.
curses.start_color()
Must be called if the programmer wants to use colors, and before any other color manipulation routine is
called. It is good practice to call this routine right after initscr().

start_color() initializes eight basic colors (black, red, green, yellow, blue, magenta, cyan, and white),
and two global variables in the curses module, COLORS and COLOR_PAIRS, containing the maximum
number of colors and color-pairs the terminal can support. It also restores the colors on the terminal to the
values they had when the terminal was just turned on.

curses.termattrs()
Return a logical OR of all video attributes supported by the terminal. This information is useful when a
curses program needs complete control over the appearance of the screen.

curses.termname()
Return the value of the environment variable TERM, truncated to 14 characters.

curses.tigetflag(capname)
Return the value of the Boolean capability corresponding to the terminfo capability name capname. The
value -1 is returned if capname is not a Boolean capability, or 0 if it is canceled or absent from the terminal
description.

curses.tigetnum(capname)
Return the value of the numeric capability corresponding to the terminfo capability name capname. The
value -2 is returned if capname is not a numeric capability, or -1 if it is canceled or absent from the
terminal description.

curses.tigetstr(capname)
Return the value of the string capability corresponding to the terminfo capability name capname. None is
returned if capname is not a string capability, or is canceled or absent from the terminal description.

curses.tparm(str[,...])
Instantiate the string str with the supplied parameters, where str should be a parameterized string
obtained from the terminfo database. E.g. tparm(tigetstr("cup"), 5, 3) could result in
b’\033[6;4H’, the exact result depending on terminal type.

curses.typeahead(fd)
Specify that the file descriptor fd be used for typeahead checking. If fd is -1, then no typeahead checking
is done.
The curses library does “line-breakout optimization” by looking for typeahead periodically while updating
the screen. If input is found, and it is coming from a tty, the current update is postponed until refresh or
doupdate is called again, allowing faster response to commands typed in advance. This function allows
specifying a different file descriptor for typeahead checking.

curses.unctrl(ch)
Return a string which is a printable representation of the character ch. Control characters are displayed as a
caret followed by the character, for example as ^C. Printing characters are left as they are.

curses.ungetch(ch)
Push ch so the next getch() will return it.

Note: Only one ch can be pushed before getch() is called.

New in version 3.3.
curses.unget_wch(ch)
Push ch so the next get_wch() will return it.

Note: Only one ch can be pushed before get_wch() is called.

curses.ungetmouse(id, x, y, z, bstate)
Push a KEY_MOUSE event onto the input queue, associating the given state data with it.
curses.**use_env**(flag)

If used, this function should be called before **initscr()** or **newterm** are called. When **flag** is **False**, the values of lines and columns specified in the terminfo database will be used, even if environment variables **LINES** and **COLUMNS** (used by default) are set, or if curses is running in a window (in which case default behavior would be to use the window size if **LINES** and **COLUMNS** are not set).

curses.**use_default_colors**()

Allow use of default values for colors on terminals supporting this feature. Use this to support transparency in your application. The default color is assigned to the color number -1. After calling this function, **init_pair**(x, curses.COLOR_RED, -1) initializes, for instance, color pair x to a red foreground color on the default background.

curses.**wrapper**(func, ...)

Initialize curses and call another callable object, **func**, which should be the rest of your curses-using application. If the application raises an exception, this function will restore the terminal to a sane state before re-raising the exception and generating a traceback. The callable object **func** is then passed the main window **'stdscr'** as its first argument, followed by any other arguments passed to **wrapper()**. Before calling **func**, **wrapper()** turns on cbreak mode, turns off echo, enables the terminal keypad, and initializes colors if the terminal has color support. On exit (whether normally or by exception) it restores cooked mode, turns on echo, and disables the terminal keypad.

### 16.11.2 Window Objects

Window objects, as returned by **initscr()** and **newwin()** above, have the following methods and attributes:

window.**addch**(ch[, attr])

Paint character **ch** at (**y**, **x**) with attributes **attr**, overwriting any character previously painted at that location. By default, the character position and attributes are the current settings for the window object.

window.**addnstr**(str, n[, attr])

Paint at most **n** characters of the string **str** at (**y**, **x**) with attributes **attr**, overwriting anything previously on the display.

window.**addstr**(str[, attr])

Paint the string **str** at (**y**, **x**) with attributes **attr**, overwriting anything previously on the display.

window.**attroff**(attr)

Remove attribute **attr** from the “background” set applied to all writes to the current window.

window.**attron**(attr)

Add attribute **attr** from the “background” set applied to all writes to the current window.

window.**attrset**(attr)

Set the “background” set of attributes to **attr**. This set is initially 0 (no attributes).

window.**bkgd**(ch[, attr])

Set the background property of the window to the character **ch**, with attributes **attr**. The change is then applied to every character position in that window:

- The attribute of every character in the window is changed to the new background attribute.
- Wherever the former background character appears, it is changed to the new background character.
The Python Library Reference, Release 3.3.3

**window.bkgdset(ch[., attr])**

Set the window’s background. A window’s background consists of a character and any combination of attributes. The attribute part of the background is combined (OR’ed) with all non-blank characters that are written into the window. Both the character and attribute parts of the background are combined with the blank characters. The background becomes a property of the character and moves with the character through any scrolling and insert/delete line/character operations.

**window.border([ls, rs, ts, bs, tl, tr, bl, br]])**

Draw a border around the edges of the window. Each parameter specifies the character to use for a specific part of the border; see the table below for more details. The characters can be specified as integers or as one-character strings.

**Note:** A 0 value for any parameter will cause the default character to be used for that parameter. Keyword parameters can not be used. The defaults are listed in this table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ls</td>
<td>Left side</td>
<td>ACS_VLINE</td>
</tr>
<tr>
<td>rs</td>
<td>Right side</td>
<td>ACS_VLINE</td>
</tr>
<tr>
<td>ts</td>
<td>Top</td>
<td>ACS_HLINE</td>
</tr>
<tr>
<td>bs</td>
<td>Bottom</td>
<td>ACS_HLINE</td>
</tr>
<tr>
<td>tl</td>
<td>Upper-left corner</td>
<td>ACS_ULCORNER</td>
</tr>
<tr>
<td>tr</td>
<td>Upper-right corner</td>
<td>ACS_URCORNER</td>
</tr>
<tr>
<td>bl</td>
<td>Bottom-left corner</td>
<td>ACS_LLCORNER</td>
</tr>
<tr>
<td>br</td>
<td>Bottom-right corner</td>
<td>ACS_LRCORNER</td>
</tr>
</tbody>
</table>

**window.box([vertch, horch])**

Similar to `border()`, but both `ls` and `rs` are `vertch` and both `ts` and `bs` are `horch`. The default corner characters are always used by this function.

**window.chgat(attr)**

**window.chgat(num, attr)**

**window.chgat(y, x, attr)**

**window.chgat(y, x, num, attr)**

Set the attributes of `num` characters at the current cursor position, or at position `(y, x)` if supplied. If no value of `num` is given or `num` = -1, the attribute will be set on all the characters to the end of the line. This function does not move the cursor. The changed line will be touched using the `touchline()` method so that the contents will be redisplayed by the next window refresh.

**window.clear**

Like `erase()`, but also cause the whole window to be repainted upon next call to `refresh()`.

**window.clearok(yes)**

If `yes` is 1, the next call to `refresh()` will clear the window completely.

**window.clrtobot**

Erase from cursor to the end of the window: all lines below the cursor are deleted, and then the equivalent of `clrtoeol()` is performed.

**window.clrtoeol**

Erase from cursor to the end of the line.

**window.cursyncup()**

Update the current cursor position of all the ancestors of the window to reflect the current cursor position of the window.

**window.delch([y,x])**

Delete any character at `(y, x)`.

**window.delete()**

Delete the line under the cursor. All following lines are moved up by one line.

**window.derwin(begin_y, begin_x)**

532 Chapter 16. Generic Operating System Services
window.derwin(nlines, ncols, begin_y, begin_x)
An abbreviation for “derive window”, derwin() is the same as calling subwin(), except that begin_y
and begin_x are relative to the origin of the window, rather than relative to the entire screen. Return a
window object for the derived window.

window.echochar(ch[, attr])
Add character ch with attribute attr, and immediately call refresh() on the window.

window.enclose(y, x)
Test whether the given pair of screen-relative character-cell coordinates are enclosed by the given window,
returning True or False. It is useful for determining what subset of the screen windows enclose the
location of a mouse event.

window.encoding
Encoding used to encode method arguments (Unicode strings and characters). The encoding attribute is
inherited from the parent window when a subwindow is created, for example with window.subwin().
By default, the locale encoding is used (see locale.getpreferredencoding()). New in version
3.3.

window.erase()
Clear the window.

window.getbegyx()
Return a tuple (y, x) of co-ordinates of upper-left corner.

window.getbkgd()
Return the given window’s current background character/attribute pair.

window.getch([y, x])
Get a character. Note that the integer returned does not have to be in ASCII range: function keys, keypad
keys and so on return numbers higher than 256. In no-delay mode, -1 is returned if there is no input, else
getch() waits until a key is pressed.

window.get_wch([y, x])
Get a wide character. Return a character for most keys, or an integer for function keys, keypad keys, and
other special keys. New in version 3.3.

window.getkey([y, x])
Get a character, returning a string instead of an integer, as getch() does. Function keys, keypad keys
and other special keys return a multibyte string containing the key name. In no-delay mode, an exception is
raised if there is no input.

window.getmaxyx()
Return a tuple (y, x) of the height and width of the window.

window.getparyx()
Return the beginning coordinates of this window relative to its parent window into two integer variables y
and x. Return -1, -1 if this window has no parent.

window.getstr([y, x])
Read a string from the user, with primitive line editing capacity.

window.getyx()
Return a tuple (y, x) of current cursor position relative to the window’s upper-left corner.

window.hline(ch, n)
window.hline(y, x, ch, n)
Display a horizontal line starting at (y, x) with length n consisting of the character ch.

window.idcok(flag)
If flag is False, curses no longer considers using the hardware insert/delete character feature of the termin-
inal; if flag is True, use of character insertion and deletion is enabled. When curses is first initialized, use
of character insert/delete is enabled by default.
If called with `yes` equal to 1, `curses` will try and use hardware line editing facilities. Otherwise, line insertion/deletion are disabled.

If `flag` is `True`, any change in the window image automatically causes the window to be refreshed; you no longer have to call `refresh()` yourself. However, it may degrade performance considerably, due to repeated calls to `wrefresh`. This option is disabled by default.

Return the character at the given position in the window. The bottom 8 bits are the character proper, and upper bits are the attributes.

Paint character `ch` at `(y, x)` with attributes `attr`, moving the line from position `x` right by one character.

Insert `nlines` lines into the specified window above the current line. The `nlines` bottom lines are lost. For negative `nlines`, delete `nlines` lines starting with the one under the cursor, and move the remaining lines up. The bottom `nlines` lines are cleared. The current cursor position remains the same.

Insert a blank line under the cursor. All following lines are moved down by one line.

Insert a character string (as many characters as will fit on the line) before the character under the cursor, up to `n` characters. If `n` is zero or negative, the entire string is inserted. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to `y, x`, if specified).

Insert a character string (as many characters as will fit on the line) before the character under the cursor. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to `y, x`, if specified).

Return a string of characters, extracted from the window starting at the current cursor position, or at `y, x` if specified. Attributes are stripped from the characters. If `n` is specified, `instr()` returns a string at most `n` characters long (exclusive of the trailing NUL).

Return `True` if the specified line was modified since the last call to `refresh()`; otherwise return `False`. Raise a `curses.error` exception if `line` is not valid for the given window.

Return `True` if the specified window was modified since the last call to `refresh()`; otherwise return `False`.

If `yes` is 1, escape sequences generated by some keys (keypad, function keys) will be interpreted by `curses`. If `yes` is 0, escape sequences will be left as is in the input stream.

If `yes` is 1, cursor is left where it is on update, instead of being at “cursor position.” This reduces cursor movement where possible. If possible the cursor will be made invisible.

If `yes` is 0, cursor will always be at “cursor position” after an update.

Move cursor to `(new_y, new_x)`. 

534 Chapter 16. Generic Operating System Services
window.mvderwin(y, x)
    Move the window inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

window.mvwin(new_y, new_x)
    Move the window so its upper-left corner is at (new_y, new_x).

window.nodelay(yes)
    If yes is 1, getch() will be non-blocking.

window.notimeout(yes)
    If yes is 1, escape sequences will not be timed out.
    If yes is 0, after a few milliseconds, an escape sequence will not be interpreted, and will be left in the input stream as is.

window.noutrefresh()
    Mark for refresh but wait. This function updates the data structure representing the desired state of the window, but does not force an update of the physical screen. To accomplish that, call doupdate().

window.overlay(destwin[, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol])
    Overlay the window on top of destwin. The windows need not be the same size, only the overlapping region is copied. This copy is non-destructive, which means that the current background character does not overwrite the old contents of destwin.

To get fine-grained control over the copied region, the second form of overlay() can be used. sminrow and smincol are the upper-left coordinates of the source window, and the other variables mark a rectangle in the destination window.

window.overwrite(destwin[, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol])
    Overwrite the window on top of destwin. The windows need not be the same size, in which case only the overlapping region is copied. This copy is destructive, which means that the current background character overwrites the old contents of destwin.

To get fine-grained control over the copied region, the second form of overwrite() can be used. sminrow and smincol are the upper-left coordinates of the source window, the other variables mark a rectangle in the destination window.

window.putwin(file)
    Write all data associated with the window into the provided file object. This information can be later retrieved using the getwin() function.

window.redrawln(beg, num)
    Indicate that the num screen lines, starting at line beg, are corrupted and should be completely redrawn on the next refresh() call.

window.redrawwin()
    Touch the entire window, causing it to be completely redrawn on the next refresh() call.

window.refresh([pminrow, pmincol, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol])
    Update the display immediately (sync actual screen with previous drawing/deleting methods).

The 6 optional arguments can only be specified when the window is a pad created with newpad(). The additional parameters are needed to indicate what part of the pad and screen are involved. pminrow and pmincol specify the upper left-hand corner of the rectangle to be displayed in the pad. sminrow, smincol, smaxrow, and smaxcol specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of pminrow, pmincol, sminrow, or smincol are treated as if they were zero.

window.resize(nlines, ncols)
    Reallocate storage for a curses window to adjust its dimensions to the specified values. If either dimension is larger than the current values, the window’s data is filled with blanks that have the current background rendition (as set by bkgdset()) merged into them.
window.scroll([lines=1])
Scroll the screen or scrolling region upward by lines lines.

window.scrolllok(flag)
Control what happens when the cursor of a window is moved off the edge of the window or scrolling region, either as a result of a newline action on the bottom line, or typing the last character of the last line. If flag is false, the cursor is left on the bottom line. If flag is true, the window is scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call idlok().

window.setscrreg(top, bottom)
Set the scrolling region from line top to line bottom. All scrolling actions will take place in this region.

window.stanend()
Turn off the standout attribute. On some terminals this has the side effect of turning off all attributes.

window.standout()
Turn on attribute A_STANDOUT.

window.subpad(begin_y, begin_x)
window.subpad(nlines, ncols, begin_y, begin_x)
Return a sub-window, whose upper-left corner is at (begin_y, begin_x), and whose width/height is ncols/nlines.

window.subwin(begin_y, begin_x)
window.subwin(nlines, ncols, begin_y, begin_x)
Return a sub-window, whose upper-left corner is at (begin_y, begin_x), and whose width/height is ncols/nlines.
By default, the sub-window will extend from the specified position to the lower right corner of the window.

window.syncdown()
Touch each location in the window that has been touched in any of its ancestor windows. This routine is called by refresh(), so it should almost never be necessary to call it manually.

window.syncok(flag)
If called with flag set to True, then syncup() is called automatically whenever there is a change in the window.

window.syncup()
Touch all locations in ancestors of the window that have been changed in the window.

window.timeout(delay)
Set blocking or non-blocking read behavior for the window. If delay is negative, blocking read is used (which will wait indefinitely for input). If delay is zero, then non-blocking read is used, and -1 will be returned by getch() if no input is waiting. If delay is positive, then getch() will block for delay milliseconds, and return -1 if there is still no input at the end of that time.

window.touchline(start, count[, changed ])
Pretend count lines have been changed, starting with line start. If changed is supplied, it specifies whether the affected lines are marked as having been changed (changed=1) or unchanged (changed=0).

window.touchwin()
Pretend the whole window has been changed, for purposes of drawing optimizations.

window.untouchwin()
Mark all lines in the window as unchanged since the last call to refresh().

window.vline(ch, n)
window.vline(y, x, ch, n)
Display a vertical line starting at (y, x) with length n consisting of the character ch.

16.11.3 Constants

The curses module defines the following data members:
curses.ERR
Some curses routines that return an integer, such as `getch()`, return `ERR` upon failure.

curses.OK
Some curses routines that return an integer, such as `napms()`, return `OK` upon success.

curses.version
A string representing the current version of the module. Also available as `__version__`.

Several constants are available to specify character cell attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_ALTCHARSET</td>
<td>Alternate character set mode.</td>
</tr>
<tr>
<td>A_BLINK</td>
<td>Blink mode.</td>
</tr>
<tr>
<td>A_BOLD</td>
<td>Bold mode.</td>
</tr>
<tr>
<td>A_DIM</td>
<td>Dim mode.</td>
</tr>
<tr>
<td>A_NORMAL</td>
<td>Normal attribute.</td>
</tr>
<tr>
<td>A_REVERSE</td>
<td>Reverse background and foreground colors.</td>
</tr>
<tr>
<td>A_STANDOUT</td>
<td>Standout mode.</td>
</tr>
<tr>
<td>A_UNDERLINE</td>
<td>Underline mode.</td>
</tr>
</tbody>
</table>

Keys are referred to by integer constants with names starting with `KEY_`. The exact keycaps available are system dependent.

<table>
<thead>
<tr>
<th>Key constant</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY_MIN</td>
<td>Minimum key value</td>
</tr>
<tr>
<td>KEY_BREAK</td>
<td>Break key (unreliable)</td>
</tr>
<tr>
<td>KEY_DOWN</td>
<td>Down-arrow</td>
</tr>
<tr>
<td>KEY_UP</td>
<td>Up-arrow</td>
</tr>
<tr>
<td>KEY_LEFT</td>
<td>Left-arrow</td>
</tr>
<tr>
<td>KEY_RIGHT</td>
<td>Right-arrow</td>
</tr>
<tr>
<td>KEY_HOME</td>
<td>Home key (upward+left arrow)</td>
</tr>
<tr>
<td>KEY_BACKSPACE</td>
<td>Backspace (unreliable)</td>
</tr>
<tr>
<td>KEY_F0</td>
<td>Function keys. Up to 64 function keys are supported.</td>
</tr>
<tr>
<td>KEY_Fn</td>
<td>Value of function key n</td>
</tr>
<tr>
<td>KEY_DL</td>
<td>Delete line</td>
</tr>
<tr>
<td>KEY_IL</td>
<td>Insert line</td>
</tr>
<tr>
<td>KEY_DC</td>
<td>Delete character</td>
</tr>
<tr>
<td>KEY_IC</td>
<td>Insert char or enter insert mode</td>
</tr>
<tr>
<td>KEY_EIC</td>
<td>Exit insert char mode</td>
</tr>
<tr>
<td>KEY_CLEAR</td>
<td>Clear screen</td>
</tr>
<tr>
<td>KEY_EOS</td>
<td>Clear to end of screen</td>
</tr>
<tr>
<td>KEY_EOL</td>
<td>Clear to end of line</td>
</tr>
<tr>
<td>KEY_SF</td>
<td>Scroll 1 line forward</td>
</tr>
<tr>
<td>KEY_SR</td>
<td>Scroll 1 line backward (reverse)</td>
</tr>
<tr>
<td>KEY_NPAGE</td>
<td>Next page</td>
</tr>
<tr>
<td>KEY_PPAGE</td>
<td>Previous page</td>
</tr>
<tr>
<td>KEY_STAB</td>
<td>Set tab</td>
</tr>
<tr>
<td>KEY_CTAB</td>
<td>Clear tab</td>
</tr>
<tr>
<td>KEY_CATAB</td>
<td>Clear all tabs</td>
</tr>
<tr>
<td>KEY_ENTER</td>
<td>Enter or send (unreliable)</td>
</tr>
<tr>
<td>KEY_SRESET</td>
<td>Soft (partial) reset (unreliable)</td>
</tr>
<tr>
<td>KEY_RESET</td>
<td>Reset or hard reset (unreliable)</td>
</tr>
<tr>
<td>KEY_PRINT</td>
<td>Print</td>
</tr>
<tr>
<td>KEY_LL</td>
<td>Home down or bottom (lower left)</td>
</tr>
<tr>
<td>KEY_A1</td>
<td>Upper left of keypad</td>
</tr>
<tr>
<td>KEY_A3</td>
<td>Upper right of keypad</td>
</tr>
<tr>
<td>KEY_B2</td>
<td>Center of keypad</td>
</tr>
<tr>
<td>KEY_C1</td>
<td>Lower left of keypad</td>
</tr>
<tr>
<td>KEY_C3</td>
<td>Lower right of keypad</td>
</tr>
</tbody>
</table>

Continued on next page
Table 16.1 – continued from previous page

<table>
<thead>
<tr>
<th>Key Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY_BTAB</td>
<td>Back tab</td>
</tr>
<tr>
<td>KEY_BEG</td>
<td>Beg (beginning)</td>
</tr>
<tr>
<td>KEY_CANCEL</td>
<td>Cancel</td>
</tr>
<tr>
<td>KEY_CLOSE</td>
<td>Close</td>
</tr>
<tr>
<td>KEY_COMMAND</td>
<td>Cmd (command)</td>
</tr>
<tr>
<td>KEY_COPY</td>
<td>Copy</td>
</tr>
<tr>
<td>KEY_CREATE</td>
<td>Create</td>
</tr>
<tr>
<td>KEY_END</td>
<td>End</td>
</tr>
<tr>
<td>KEY_EXIT</td>
<td>Exit</td>
</tr>
<tr>
<td>KEY_FIND</td>
<td>Find</td>
</tr>
<tr>
<td>KEY_HELP</td>
<td>Help</td>
</tr>
<tr>
<td>KEY_MARK</td>
<td>Mark</td>
</tr>
<tr>
<td>KEY_MESSAGE</td>
<td>Message</td>
</tr>
<tr>
<td>KEY_MOVE</td>
<td>Move</td>
</tr>
<tr>
<td>KEY_NEXT</td>
<td>Next</td>
</tr>
<tr>
<td>KEY_OPEN</td>
<td>Open</td>
</tr>
<tr>
<td>KEY_OPTIONS</td>
<td>Options</td>
</tr>
<tr>
<td>KEY_PREVIOUS</td>
<td>Prev (previous)</td>
</tr>
<tr>
<td>KEY_REDO</td>
<td>Redo</td>
</tr>
<tr>
<td>KEY_REFERENCE</td>
<td>Ref (reference)</td>
</tr>
<tr>
<td>KEY_REFRESH</td>
<td>Refresh</td>
</tr>
<tr>
<td>KEY_REPLACE</td>
<td>Replace</td>
</tr>
<tr>
<td>KEY_RESTART</td>
<td>Restart</td>
</tr>
<tr>
<td>KEY_RESUME</td>
<td>Resume</td>
</tr>
<tr>
<td>KEY_SAVE</td>
<td>Save</td>
</tr>
<tr>
<td>KEY_SBEG</td>
<td>Shifted Beg (beginning)</td>
</tr>
<tr>
<td>KEY_SCANCEL</td>
<td>Shifted Cancel</td>
</tr>
<tr>
<td>KEY_SCOMMAND</td>
<td>Shifted Command</td>
</tr>
<tr>
<td>KEY_SCOPY</td>
<td>Shifted Copy</td>
</tr>
<tr>
<td>KEY_SCREATE</td>
<td>Shifted Create</td>
</tr>
<tr>
<td>KEY_SDC</td>
<td>Shifted Delete char</td>
</tr>
<tr>
<td>KEY_SDL</td>
<td>Shifted Delete line</td>
</tr>
<tr>
<td>KEY_SELECT</td>
<td>Select</td>
</tr>
<tr>
<td>KEY_SEND</td>
<td>Shifted End</td>
</tr>
<tr>
<td>KEY_SEOL</td>
<td>Shifted Clear line</td>
</tr>
<tr>
<td>KEY_SEXIT</td>
<td>Shifted Dxit</td>
</tr>
<tr>
<td>KEY_SFIND</td>
<td>Shifted Find</td>
</tr>
<tr>
<td>KEY_SHELP</td>
<td>Shifted Help</td>
</tr>
<tr>
<td>KEY_SHOME</td>
<td>Shifted Home</td>
</tr>
<tr>
<td>KEY_SIC</td>
<td>Shifted Input</td>
</tr>
<tr>
<td>KEY_SLEFT</td>
<td>Shifted Left arrow</td>
</tr>
<tr>
<td>KEY_SMMESSAGE</td>
<td>Shifted Message</td>
</tr>
<tr>
<td>KEY_SMOVE</td>
<td>Shifted Move</td>
</tr>
<tr>
<td>KEY_SNEXT</td>
<td>Shifted Next</td>
</tr>
<tr>
<td>KEY_SOPTIONS</td>
<td>Shifted Options</td>
</tr>
<tr>
<td>KEY_SPREREVIOUS</td>
<td>Shifted Prev</td>
</tr>
<tr>
<td>KEY_SPRINT</td>
<td>Shifted Print</td>
</tr>
<tr>
<td>KEY_SREDO</td>
<td>Shifted Redo</td>
</tr>
<tr>
<td>KEY_SREPLACE</td>
<td>Shifted Replace</td>
</tr>
<tr>
<td>KEY_SRIGHT</td>
<td>Shifted Right arrow</td>
</tr>
<tr>
<td>KEY_SRSUME</td>
<td>Shifted Resume</td>
</tr>
<tr>
<td>KEY_SSAVE</td>
<td>Shifted Save</td>
</tr>
<tr>
<td>KEY_SSUSPEND</td>
<td>Shifted Suspend</td>
</tr>
<tr>
<td>KEY_SUNDO</td>
<td>Shifted Undo</td>
</tr>
<tr>
<td>KEY_SUSPEND</td>
<td>Suspend</td>
</tr>
<tr>
<td>KEY_UNDO</td>
<td>Undo</td>
</tr>
</tbody>
</table>

Continued on next page
On VT100s and their software emulations, such as X terminal emulators, there are normally at least four function keys (KEY_F1, KEY_F2, KEY_F3, KEY_F4) available, and the arrow keys mapped to KEY_UP, KEY_DOWN, KEY_LEFT and KEY_RIGHT in the obvious way. If your machine has a PC keyboard, it is safe to expect arrow keys and twelve function keys (older PC keyboards may have only ten function keys); also, the following keypad mappings are standard:

<table>
<thead>
<tr>
<th>Keycap</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>KEY_IC</td>
</tr>
<tr>
<td>Delete</td>
<td>KEY_DC</td>
</tr>
<tr>
<td>Home</td>
<td>KEY_HOME</td>
</tr>
<tr>
<td>End</td>
<td>KEY_END</td>
</tr>
<tr>
<td>Page Up</td>
<td>KEY_NPAGE</td>
</tr>
<tr>
<td>Page Down</td>
<td>KEY_PPAGE</td>
</tr>
</tbody>
</table>

The following table lists characters from the alternate character set. These are inherited from the VT100 terminal, and will generally be available on software emulations such as X terminals. When there is no graphic available, curses falls back on a crude printable ASCII approximation.

**Note:** These are available only after `initscr()` has been called.

<table>
<thead>
<tr>
<th>ACS code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACS_BBSS</td>
<td>alternate name for upper right corner</td>
</tr>
<tr>
<td>ACS_BLOCK</td>
<td>solid square block</td>
</tr>
<tr>
<td>ACS_BOARD</td>
<td>board of squares</td>
</tr>
<tr>
<td>ACS_BBBS</td>
<td>alternate name for horizontal line</td>
</tr>
<tr>
<td>ACS_BSSB</td>
<td>alternate name for upper left corner</td>
</tr>
<tr>
<td>ACS_BSSS</td>
<td>alternate name for top tee</td>
</tr>
<tr>
<td>ACS_BTEE</td>
<td>bottom tee</td>
</tr>
<tr>
<td>ACS_BULLET</td>
<td>bullet</td>
</tr>
<tr>
<td>ACS_CKBOARD</td>
<td>checker board (stipple)</td>
</tr>
<tr>
<td>ACS_DARROW</td>
<td>arrow pointing down</td>
</tr>
<tr>
<td>ACS_DEGREE</td>
<td>greater-than-or-equal-to</td>
</tr>
<tr>
<td>ACS_DIAMOND</td>
<td>diamond</td>
</tr>
<tr>
<td>ACS_GEQUAL</td>
<td>less-than-or-equal-to</td>
</tr>
<tr>
<td>ACS_HLINE</td>
<td>horizontal line</td>
</tr>
<tr>
<td>ACS_LANTERN</td>
<td>lantern symbol</td>
</tr>
<tr>
<td>ACS_LARROW</td>
<td>left arrow</td>
</tr>
<tr>
<td>ACS_LEQUAL</td>
<td>left-hand corner</td>
</tr>
<tr>
<td>ACS_LLCORNER</td>
<td>lower left-hand corner</td>
</tr>
<tr>
<td>ACS_LRCORNER</td>
<td>lower right-hand corner</td>
</tr>
<tr>
<td>ACS_LTEE</td>
<td>left tee</td>
</tr>
<tr>
<td>ACS_NEQUAL</td>
<td>not-equal sign</td>
</tr>
<tr>
<td>ACS_PI</td>
<td>letter pi</td>
</tr>
<tr>
<td>ACS_PLMINUS</td>
<td>plus-or-minus sign</td>
</tr>
<tr>
<td>ACS_PLUS</td>
<td>big plus sign</td>
</tr>
<tr>
<td>ACS_RARROW</td>
<td>right arrow</td>
</tr>
<tr>
<td>ACS_RTEE</td>
<td>right tee</td>
</tr>
<tr>
<td>ACS_S1</td>
<td>scan line 1</td>
</tr>
<tr>
<td>ACS_S3</td>
<td>scan line 3</td>
</tr>
<tr>
<td>ACS_S7</td>
<td>scan line 7</td>
</tr>
<tr>
<td>ACS_S9</td>
<td>scan line 9</td>
</tr>
<tr>
<td>ACS_SBBBS</td>
<td>alternate name for lower right corner</td>
</tr>
</tbody>
</table>

Continued on next page
The following table lists the predefined colors:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR_BLACK</td>
<td>Black</td>
</tr>
<tr>
<td>COLOR_BLUE</td>
<td>Blue</td>
</tr>
<tr>
<td>COLOR_CYAN</td>
<td>Cyan (light greenish blue)</td>
</tr>
<tr>
<td>COLOR_GREEN</td>
<td>Green</td>
</tr>
<tr>
<td>COLOR_MAGENTA</td>
<td>Magenta (purplish red)</td>
</tr>
<tr>
<td>COLOR_RED</td>
<td>Red</td>
</tr>
<tr>
<td>COLOR_WHITE</td>
<td>White</td>
</tr>
<tr>
<td>COLOR_YELLOW</td>
<td>Yellow</td>
</tr>
</tbody>
</table>

16.12 `curses.textpad` — Text input widget for curses programs

The `curses.textpad` module provides a `Textbox` class that handles elementary text editing in a curses window, supporting a set of keybindings resembling those of Emacs (thus, also of Netscape Navigator, BBedit 6.x, FrameMaker, and many other programs). The module also provides a rectangle-drawing function useful for framing text boxes or for other purposes.

The module `curses.textpad` defines the following function:

```python
curses.textpad.rectangle(win, uly, ulx, lry, lrx)
```

Draw a rectangle. The first argument must be a window object; the remaining arguments are coordinates relative to that window. The second and third arguments are the y and x coordinates of the upper left hand corner of the rectangle to be drawn; the fourth and fifth arguments are the y and x coordinates of the lower right hand corner. The rectangle will be drawn using VT100/IBM PC forms characters on terminals that make this possible (including xterm and most other software terminal emulators). Otherwise it will be drawn with ASCII dashes, vertical bars, and plus signs.

16.12.1 Textbox objects

You can instantiate a `Textbox` object as follows:

```python
class curses.textpad.Textbox(win)
```

Return a textbox widget object. The `win` argument should be a `curses.WindowObject` in which the textbox is to be contained. The edit cursor of the textbox is initially located at the upper left hand corner of the containing window, with coordinates (0, 0). The instance’s `stripspaces` flag is initially on.

Textbox objects have the following methods:

```python
edit([validator])
```

This is the entry point you will normally use. It accepts editing keystrokes until one of the termination keystrokes is entered. If `validator` is supplied, it must be a function. It will be called for each keystroke entered with the keystroke as a parameter; command dispatch is done on the result. This method
returns the window contents as a string; whether blanks in the window are included is affected by the `stripspaces` attribute.

**do_command**(ch)

Process a single command keystroke. Here are the supported special keystrokes:

<table>
<thead>
<tr>
<th>Keystroke</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control-A</td>
<td>Go to left edge of window.</td>
</tr>
<tr>
<td>Control-B</td>
<td>Cursor left, wrapping to previous line if appropriate.</td>
</tr>
<tr>
<td>Control-D</td>
<td>Delete character under cursor.</td>
</tr>
<tr>
<td>Control-E</td>
<td>Go to right edge (stripspaces off) or end of line (stripspaces on).</td>
</tr>
<tr>
<td>Control-F</td>
<td>Cursor right, wrapping to next line when appropriate.</td>
</tr>
<tr>
<td>Control-G</td>
<td>Terminate, returning the window contents.</td>
</tr>
<tr>
<td>Control-H</td>
<td>Delete character backward.</td>
</tr>
<tr>
<td>Control-J</td>
<td>Terminate if the window is 1 line, otherwise insert newline.</td>
</tr>
<tr>
<td>Control-K</td>
<td>If line is blank, delete it, otherwise clear to end of line.</td>
</tr>
<tr>
<td>Control-L</td>
<td>Refresh screen.</td>
</tr>
<tr>
<td>Control-N</td>
<td>Cursor down; move down one line.</td>
</tr>
<tr>
<td>Control-O</td>
<td>Insert a blank line at cursor location.</td>
</tr>
<tr>
<td>Control-P</td>
<td>Cursor up; move up one line.</td>
</tr>
</tbody>
</table>

Move operations do nothing if the cursor is at an edge where the movement is not possible. The following synonyms are supported where possible:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Keystroke</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY_LEFT</td>
<td>Control-B</td>
</tr>
<tr>
<td>KEY_RIGHT</td>
<td>Control-F</td>
</tr>
<tr>
<td>KEY_UP</td>
<td>Control-P</td>
</tr>
<tr>
<td>KEY_DOWN</td>
<td>Control-N</td>
</tr>
<tr>
<td>KEY_BACKSPACE</td>
<td>Control-h</td>
</tr>
</tbody>
</table>

All other keystrokes are treated as a command to insert the given character and move right (with line wrapping).

**gather**()

Return the window contents as a string; whether blanks in the window are included is affected by the `stripspaces` member.

**stripspaces**

This attribute is a flag which controls the interpretation of blanks in the window. When it is on, trailing blanks on each line are ignored; any cursor motion that would land the cursor on a trailing blank goes to the end of that line instead, and trailing blanks are stripped when the window contents are gathered.

### 16.13 `curses.ascii` — Utilities for ASCII characters

The `curses.ascii` module supplies name constants for ASCII characters and functions to test membership in various ASCII character classes. The constants supplied are names for control characters as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>Start of heading, console interrupt</td>
</tr>
<tr>
<td>SOH</td>
<td>Start of text</td>
</tr>
<tr>
<td>STX</td>
<td>End of text</td>
</tr>
<tr>
<td>ETX</td>
<td>End of transmission</td>
</tr>
<tr>
<td>ENQ</td>
<td>Enquiry, goes with ACK flow control</td>
</tr>
<tr>
<td>ACK</td>
<td>Acknowledgement</td>
</tr>
<tr>
<td>BEL</td>
<td>Bell</td>
</tr>
<tr>
<td>BS</td>
<td>Backspace</td>
</tr>
<tr>
<td>TAB</td>
<td>Tab</td>
</tr>
</tbody>
</table>

Continued on next page
Table 16.3 – continued from previous page

<table>
<thead>
<tr>
<th>HT</th>
<th>Alias for TAB: “Horizontal tab”</th>
</tr>
</thead>
<tbody>
<tr>
<td>LF</td>
<td>Line feed</td>
</tr>
<tr>
<td>NL</td>
<td>Alias for LF: “New line”</td>
</tr>
<tr>
<td>VT</td>
<td>Vertical tab</td>
</tr>
<tr>
<td>FF</td>
<td>Form feed</td>
</tr>
<tr>
<td>CR</td>
<td>Carriage return</td>
</tr>
<tr>
<td>SO</td>
<td>Shift-out, begin alternate character set</td>
</tr>
<tr>
<td>SI</td>
<td>Shift-in, resume default character set</td>
</tr>
<tr>
<td>DLE</td>
<td>Data-link escape</td>
</tr>
<tr>
<td>DC1</td>
<td>XON, for flow control</td>
</tr>
<tr>
<td>DC2</td>
<td>Device control 2, block-mode flow control</td>
</tr>
<tr>
<td>DC3</td>
<td>XOFF, for flow control</td>
</tr>
<tr>
<td>DC4</td>
<td>Device control 4</td>
</tr>
<tr>
<td>NAK</td>
<td>Negative acknowledgement</td>
</tr>
<tr>
<td>SYN</td>
<td>Synchronous idle</td>
</tr>
<tr>
<td>ETB</td>
<td>End transmission block</td>
</tr>
<tr>
<td>CAN</td>
<td>Cancel</td>
</tr>
<tr>
<td>EM</td>
<td>End of medium</td>
</tr>
<tr>
<td>SUB</td>
<td>Substitute</td>
</tr>
<tr>
<td>ESC</td>
<td>Escape</td>
</tr>
<tr>
<td>FS</td>
<td>File separator</td>
</tr>
<tr>
<td>GS</td>
<td>Group separator</td>
</tr>
<tr>
<td>RS</td>
<td>Record separator, block-mode terminator</td>
</tr>
<tr>
<td>US</td>
<td>Unit separator</td>
</tr>
<tr>
<td>SP</td>
<td>Space</td>
</tr>
<tr>
<td>DEL</td>
<td>Delete</td>
</tr>
</tbody>
</table>

Note that many of these have little practical significance in modern usage. The mnemonics derive from teleprinter conventions that predate digital computers.

The module supplies the following functions, patterned on those in the standard C library:

- `curses.ascii.isalnum(c)`
  Checks for an ASCII alphanumeric character; it is equivalent to `isalpha(c) or isdigit(c)`.

- `curses.ascii.isalpha(c)`
  Checks for an ASCII alphabetic character; it is equivalent to `isupper(c) or islower(c)`.

- `curses.ascii.isascii(c)`
  Checks for a character value that fits in the 7-bit ASCII set.

- `curses.ascii.isblank(c)`
  Checks for an ASCII whitespace character.

- `curses.ascii.iscntrl(c)`
  Checks for an ASCII control character (in the range 0x00 to 0x1f).

- `curses.ascii.isdigit(c)`
  Checks for an ASCII decimal digit, ’0’ through ’9’. This is equivalent to `c in string.digits`.

- `curses.ascii.isgraph(c)`
  Checks for ASCII any printable character except space.

- `curses.ascii.islower(c)`
  Checks for an ASCII lower-case character.

- `curses.ascii.isprint(c)`
  Checks for any ASCII printable character including space.

- `curses.ascii.ispunct(c)`
  Checks for any printable ASCII character which is not a space or an alphanumerical character.
curses.ascii.isspace(c)
Checks for ASCII white-space characters; space, line feed, carriage return, form feed, horizontal tab, vertical tab.

curses.ascii.isupper(c)
Checks for an ASCII uppercase letter.

curses.ascii.isxdigit(c)
Checks for an ASCII hexadecimal digit. This is equivalent to c in string.hexdigits.

curses.ascii.isctrl(c)
Checks for an ASCII control character (ordinal values 0 to 31).

curses.ascii.ismeta(c)
Checks for a non-ASCII character (ordinal values 0x80 and above).

These functions accept either integers or strings; when the argument is a string, it is first converted using the built-in function ord().

Note that all these functions check ordinal bit values derived from the first character of the string you pass in; they do not actually know anything about the host machine’s character encoding. For functions that know about the character encoding (and handle internationalization properly) see the string module.

The following two functions take either a single-character string or integer byte value; they return a value of the same type.

curses.ascii.ascii(c)
Return the ASCII value corresponding to the low 7 bits of c.

curses.ascii.ctrl(c)
Return the control character corresponding to the given character (the character bit value is bitwise-anded with 0x1f).

curses.ascii.alt(c)
Return the 8-bit character corresponding to the given ASCII character (the character bit value is bitwise-ored with 0x80).

The following function takes either a single-character string or integer value; it returns a string.

curses.ascii.unctrl(c)
Return a string representation of the ASCII character c. If c is printable, this string is the character itself. If the character is a control character (0x00-0x1f) the string consists of a caret (‘^’) followed by the corresponding uppercase letter. If the character is an ASCII delete (0x7f) the string is ‘^?’ . If the character has its meta bit (0x80) set, the meta bit is stripped, the preceding rules applied, and ‘! ’ prepended to the result.

curses.ascii.controlnames
A 33-element string array that contains the ASCII mnemonics for the thirty-two ASCII control characters from 0 (NUL) to 0x1f (US), in order, plus the mnemonic SP for the space character.

16.14 curses.panel — A panel stack extension for curses

Panels are windows with the added feature of depth, so they can be stacked on top of each other, and only the visible portions of each window will be displayed. Panels can be added, moved up or down in the stack, and removed.

16.14.1 Functions

The module curses.panel defines the following functions:

curses.panel.bottom_panel()
Returns the bottom panel in the panel stack.
curses.panel.new_panel(win)
   Returns a panel object, associating it with the given window win. Be aware that you need to keep the returned panel object referenced explicitly. If you don’t, the panel object is garbage collected and removed from the panel stack.

curses.panel.top_panel()
   Returns the top panel in the panel stack.

curses.panel.update_panels()
   Updates the virtual screen after changes in the panel stack. This does not call curses.doupdate(), so you’ll have to do this yourself.

16.14.2 Panel Objects

Panel objects, as returned by new_panel() above, are windows with a stacking order. There’s always a window associated with a panel which determines the content, while the panel methods are responsible for the window’s depth in the panel stack.

Panel objects have the following methods:

Panel.above()
   Returns the panel above the current panel.

Panel.below()
   Returns the panel below the current panel.

Panel.bottom()
   Push the panel to the bottom of the stack.

Panel.hidden()
   Returns true if the panel is hidden (not visible), false otherwise.

Panel.hide()
   Hide the panel. This does not delete the object, it just makes the window on screen invisible.

Panel.move(y, x)
   Move the panel to the screen coordinates (y, x).

Panel.replace(win)
   Change the window associated with the panel to the window win.

Panel.set_userptr(obj)
   Set the panel’s user pointer to obj. This is used to associate an arbitrary piece of data with the panel, and can be any Python object.

Panel.show()
   Display the panel (which might have been hidden).

Panel.top()
   Push panel to the top of the stack.

Panel.userptr()
   Returns the user pointer for the panel. This might be any Python object.

Panel.window()
   Returns the window object associated with the panel.

16.15 platform — Access to underlying platform’s identifying data

Source code: Lib/platform.py
Note: Specific platforms listed alphabetically, with Linux included in the Unix section.

16.15.1 Cross Platform

platform.architecture(executable=sys.executable, bits='', linkage='')
Queries the given executable (defaults to the Python interpreter binary) for various architecture information.

Returns a tuple (bits, linkage) which contain information about the bit architecture and the linkage format used for the executable. Both values are returned as strings.

Values that cannot be determined are returned as given by the parameter presets. If bits is given as '', the sizeof(pointer) (or sizeof(long) on Python version < 1.5.2) is used as indicator for the supported pointer size.

The function relies on the system’s file command to do the actual work. This is available on most if not all Unix platforms and some non-Unix platforms and then only if the executable points to the Python interpreter. Reasonable defaults are used when the above needs are not met.

Note: On Mac OS X (and perhaps other platforms), executable files may be universal files containing multiple architectures.

To get at the “64-bitness” of the current interpreter, it is more reliable to query the sys.maxsize attribute:

is_64bits = sys.maxsize > 2**32

platform.machine()
Returns the machine type, e.g. ‘i386’. An empty string is returned if the value cannot be determined.

platform.node()
Returns the computer’s network name (may not be fully qualified!). An empty string is returned if the value cannot be determined.

platform.platform(aliased=0, terse=0)
Returns a single string identifying the underlying platform with as much useful information as possible.

The output is intended to be human readable rather than machine parseable. It may look different on different platforms and this is intended.

If aliased is true, the function will use aliases for various platforms that report system names which differ from their common names, for example SunOS will be reported as Solaris. The system_alias() function is used to implement this.

Setting terse to true causes the function to return only the absolute minimum information needed to identify the platform.

platform.processor()
Returns the (real) processor name, e.g. ‘amd64’.

An empty string is returned if the value cannot be determined. Note that many platforms do not provide this information or simply return the same value as for machine(). NetBSD does this.

platform.python_build()
Returns a tuple (buildno, builddate) stating the Python build number and date as strings.

platform.python_compiler()
Returns a string identifying the compiler used for compiling Python.

platform.python_branch()
Returns a string identifying the Python implementation SCM branch.
platform.python_implementation()
    Returns a string identifying the Python implementation. Possible return values are: ‘CPython’, ‘Iron-

platform.python_revision()
    Returns a string identifying the Python implementation SCM revision.

platform.python_version()
    Returns the Python version as string ‘major.minor.patchlevel’
    Note that unlike the Python sys.version, the returned value will always include the patchlevel (it de-
    faults to 0).

platform.python_version_tuple()
    Returns the Python version as tuple (major, minor, patchlevel) of strings.
    Note that unlike the Python sys.version, the returned value will always include the patchlevel (it de-
    faults to ‘0’).

platform.release()
    Returns the system’s release, e.g. ‘2.2.0’ or ‘NT’ An empty string is returned if the value cannot be
determined.

platform.system()
    Returns the system/OS name, e.g. ‘Linux’, ‘Windows’, or ‘Java’. An empty string is returned if the
value cannot be determined.

platform.system_alias (system, release, version)
    Returns (system, release, version) aliased to common marketing names used for some systems.
It also does some reordering of the information in some cases where it would otherwise cause confusion.

platform.version()
    Returns the system’s release version, e.g. ‘#3 on degas’. An empty string is returned if the value
cannot be determined.

platform.uname()
    Fairly portable uname interface. Returns a namedtuple() containing six attributes: system, node,
release, version, machine, and processor.

    Note that this adds a sixth attribute (processor) not present in the os.uname() result. Also,
the attribute names are different for the first two attributes; os.uname() names them sysname and
nodename.

    Entries which cannot be determined are set to ”. Changed in version 3.3: Result changed from a tuple to a
namedtuple.

16.15.2 Java Platform

platform.java_ver (release='', vendor='', vminfo=(''), osinfo=(''))
    Version interface for Jython.
    Returns a tuple (release, vendor, vminfo, osinfo) with vminfo being a tuple (vm_name, vm_release, vm_vendor) and osinfo being a tuple (os_name, os_version, os_arch).
    Values which cannot be determined are set to the defaults given as parameters (which all default to ”).

16.15.3 Windows Platform

platform.win32_ver (release='', version='', csd='', ptype='')
    Get additional version information from the Windows Registry and return a tuple (release, version,
csd, ptype) referring to OS release, version number, CSD level (service pack) and OS type
(multi/single processor).
As a hint: ptype is ‘Uniprocessor Free’ on single processor NT machines and ‘Multiprocessor Free’ on multi processor machines. The ‘Free’ refers to the OS version being free of debugging code. It could also state ‘Checked’ which means the OS version uses debugging code, i.e. code that checks arguments, ranges, etc.

**Note:** This function works best with Mark Hammond’s win32all package installed, but also on Python 2.3 and later (support for this was added in Python 2.6). It obviously only runs on Win32 compatible platforms.

### Win95/98 specific

```python
platform.popen(cmd, mode='r', bufsize=-1)
```

Portable popen() interface. Find a working popen implementation preferring win32pipe.popen(). On Windows NT, win32pipe.popen() should work; on Windows 9x it hangs due to bugs in the MS C library. Deprecated since version 3.3: This function is obsolete. Use the subprocess module. Check especially the Replacing Older Functions with the subprocess Module section.

### 16.15.4 Mac OS Platform

```python
platform.mac_ver(release='', versioninfo=('', '', ''), machine='')
```

Get Mac OS version information and return it as tuple (release, versioninfo, machine) with versioninfo being a tuple (version, dev_stage, non_release_version).

Entries which cannot be determined are set to ‘’. All tuple entries are strings.

### 16.15.5 Unix Platforms

```python
platform.dist(distname='', version='', id='', supported_dists=('SuSE', 'debian', 'redhat', 'mandrake', ...))
```

This is another name for linux_distribution().

```python
platform.linux_distribution(distname='', version='', id='', supported_dists=('SuSE', 'debian', 'redhat', 'mandrake', ...), full_distribution_name=1)
```

Tries to determine the name of the Linux OS distribution name.

supported_dists may be given to define the set of Linux distributions to look for. It defaults to a list of currently supported Linux distributions identified by their release file name.

If full_distribution_name is true (default), the full distribution read from the OS is returned. Otherwise the short name taken from supported_dists is used.

Returns a tuple (distname, version, id) which defaults to the args given as parameters. id is the item in parentheses after the version number. It is usually the version codename.

```python
platform.libc_ver(executable=sys.executable, lib='', version='', chunksize=2048)
```

Tries to determine the libc version against which the file executable (defaults to the Python interpreter) is linked. Returns a tuple of strings (lib, version) which default to the given parameters in case the lookup fails.

Note that this function has intimate knowledge of how different libc versions add symbols to the executable is probably only usable for executables compiled using gcc.

The file is read and scanned in chunks of chunksize bytes.
16.16  **errno — Standard errno system symbols**

This module makes available standard `errno` system symbols. The value of each symbol is the corresponding integer value. The names and descriptions are borrowed from `linux/include/errno.h`, which should be pretty all-inclusive.

`errno.errorcode`

Dictionary providing a mapping from the `errno` value to the string name in the underlying system. For instance, `errno.errorcode[errno.EPERM]` maps to ‘EPERM’.

To translate a numeric error code to an error message, use `os.strerror()`.

Of the following list, symbols that are not used on the current platform are not defined by the module. The specific list of defined symbols is available as `errno.errorcode.keys()`. Symbols available can include:

- `errno.EPERM`  
  Operation not permitted

- `errno.ENOENT`  
  No such file or directory

- `errno.ESRCH`  
  No such process

- `errno.EINTR`  
  Interrupted system call

- `errno.EIO`  
  I/O error

- `errno.ENXIO`  
  No such device or address

- `errno.E2BIG`  
  Arg list too long

- `errno.ENOEXEC`  
  Exec format error

- `errno.EBADF`  
  Bad file number

- `errno.ECHILD`  
  No child processes

- `errno.EAGAIN`  
  Try again

- `errno.ENOMEM`  
  Out of memory

- `errno.EACCES`  
  Permission denied

- `errno.EFAULT`  
  Bad address

- `errno.ENOTBLK`  
  Block device required

- `errno.EBUSY`  
  Device or resource busy

- `errno.EXIST`  
  File exists

- `errno.EXDEV`  
  Cross-device link
errno.\texttt{ENODEV}
   No such device
errno.\texttt{ENOTDIR}
   Not a directory
errno.\texttt{EISDIR}
   Is a directory
errno.\texttt{EINVAL}
   Invalid argument
errno.\texttt{ENFILE}
   File table overflow
errno.\texttt{EMFILE}
   Too many open files
errno.\texttt{ENOTTY}
   Not a typewriter
errno.\texttt{ETXTBSY}
   Text file busy
errno.\texttt{EFAULT}
   File too large
errno.\texttt{ENOSPC}
   No space left on device
errno.\texttt{ESPIPE}
   Broken pipe
errno.\texttt{EDOM}
   Math argument out of domain of func
errno.\texttt{ERANGE}
   Math result not representable
errno.\texttt{EDOM}
   Math argument out of domain of func
errno.\texttt{ERANGE}
   Math result not representable
errno.\texttt{EDEADLK}
   Resource deadlock would occur
errno.\texttt{ENOMEM}
   Not enough memory
errno.\texttt{ENOMEM}
   Not enough memory
errno.\texttt{ENOSYS}
   Function not implemented
errno.\texttt{ENOTEMPTY}
   Directory not empty
errno.\texttt{ELOOP}
   Too many links
errno.\texttt{EWOULDBLOCK}
   Operation would block
errno. **ENOMSG**
No message of desired type

errno. **EIDRM**
Identifier removed

errno. **ECHRNG**
Channel number out of range

errno. **EL2NSYNC**
Level 2 not synchronized

errno. **EL3HLT**
Level 3 halted

errno. **EL3RST**
Level 3 reset

errno. **ELNRNG**
Link number out of range

errno. **EUNATCH**
Protocol driver not attached

errno. **ENOCSI**
No CSI structure available

errno. **EL2HLT**
Level 2 halted

errno. **EBADE**
Invalid exchange

errno. **EBADR**
Invalid request descriptor

errno. **EXFULL**
Exchange full

errno. **ENOANO**
No anode

errno. **EBADRQC**
Invalid request code

errno. **EBADSLT**
Invalid slot

errno. **EDEADLOCK**
File locking deadlock error

errno. **EBFONT**
Bad font file format

errno. **ENOSTR**
Device not a stream

errno. **ENODATA**
No data available

errno. **ETIME**
Timer expired

errno. **ENOSR**
Out of streams resources

errno. **ENONET**
Machine is not on the network
The Python Library Reference, Release 3.3.3

errno. **ENOPKG**
Package not installed

errno. **EREMOTE**
Object is remote

errno. **ENOLINK**
Link has been severed

errno. **EADV**
Advertise error

errno. **ESRMNT**
Srsmount error

errno. **ECOMM**
Communication error on send

errno. **EPROTO**
Protocol error

errno. **EMULTIHOP**
Multihop attempted

errno. **EDOTDOT**
RFS specific error

errno. **EBADMSG**
Not a data message

errno. **EOVERFLOW**
Value too large for defined data type

errno. **ENOTUNIQ**
Name not unique on network

errno. **EBADFD**
File descriptor in bad state

errno. **EREMCHG**
Remote address changed

errno. **ELIBACC**
Can not access a needed shared library

errno. **ELIBBAD**
Accessing a corrupted shared library

errno. **ELIBSCN**
.lib section in a.out corrupted

errno. **ELIBMAX**
Attempting to link in too many shared libraries

errno. **ELIBEXEC**
Cannot exec a shared library directly

errno. **EILSEQ**
Illegal byte sequence

errno. **ERESTART**
Interrupted system call should be restarted

errno. **ESTRPIPE**
Streams pipe error

errno. **EUSERS**
Too many users

16.16. **errno** — Standard errno system symbols 551
errno.ENOTSOCK
    Socket operation on non-socket
errno.EDESTADDRREQ
    Destination address required
errno.EMSGSIZE
    Message too long
errno.EPROTOTYPE
    Protocol wrong type for socket
errno.ENOPROTOOPT
    Protocol not available
errno.EPROTONOSUPPORT
    Protocol not supported
errno.ESOCKTNOSUPPORT
    Socket type not supported
errno.EOPNOTSUPP
    Operation not supported on transport endpoint
errno.EPFNOSUPPORT
    Protocol family not supported
errno.EAFNOSUPPORT
    Address family not supported by protocol
errno.EADDRINUSE
    Address already in use
errno.EADDRNOTAVAIL
    Cannot assign requested address
errno.ENETDOWN
    Network is down
errno.ENETUNREACH
    Network is unreachable
errno.ENETRESET
    Network dropped connection because of reset
errno.ECONNABORTED
    Software caused connection abort
errno.ECONNRESET
    Connection reset by peer
errno.ENOBUFS
    No buffer space available
errno.EISCONN
    Transport endpoint is already connected
errno.ENOTCONN
    Transport endpoint is not connected
errno.ESHUTDOWN
    Cannot send after transport endpoint shutdown
errno.ETOOMANYREFS
    Too many references: cannot splice
errno.ETIMEDOUT
    Connection timed out
errno. **ECONNREFUSED**
Connection refused

errno. **EHOSTDOWN**
Host is down

errno. **EHOSTUNREACH**
No route to host

errno. **EALREADY**
Operation already in progress

errno. **EINPROGRESS**
Operation now in progress

errno. **ESTALE**
Stale NFS file handle

errno. **EUCLEAN**
Structure needs cleaning

errno. **ENOTNAM**
Not a XENIX named type file

errno. **ENAVAIL**
No XENIX semaphores available

errno. **EISNAM**
Is a named type file

errno. **EREMOTEIO**
Remote I/O error

errno. **EDQUOT**
Quota exceeded

### 16.17 ctypes — A foreign function library for Python

c**types** is a foreign function library for Python. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python.

#### 16.17.1 ctypes tutorial

Note: The code samples in this tutorial use doctest to make sure that they actually work. Since some code samples behave differently under Linux, Windows, or Mac OS X, they contain doctest directives in comments.

Note: Some code samples reference the ctypes c_int type. This type is an alias for the c_long type on 32-bit systems. So, you should not be confused if c_long is printed if you would expect c_int — they are actually the same type.

**Loading dynamic link libraries**

c**types** exports the c**dll**, and on Windows windll and oledll objects, for loading dynamic link libraries.

You load libraries by accessing them as attributes of these objects. c**dll** loads libraries which export functions using the standard cdecl calling convention, while windll libraries call functions using the stdcall calling convention. oledll also uses the stdcall calling convention, and assumes the functions return a Windows HRESULT error code. The error code is used to automatically raise a OSError exception when the function call fails. Changed in version 3.3: Windows errors used to raise WindowsError, which is now an alias of OSError. Here are some examples for Windows. Note that msvcr**t** is the MS standard C library containing most standard C functions, and uses the cdecl calling convention:
Windows appends the usual `.dll` file suffix automatically.

On Linux, it is required to specify the filename including the extension to load a library, so attribute access cannot be used to load libraries. Either the `LoadLibrary()` method of the dll loaders should be used, or you should load the library by creating an instance of `CDLL` by calling the constructor:

```python
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
```

### Accessing functions from loaded dlls

Functions are accessed as attributes of dll objects:

```python
>>> from ctypes import *
>>> libc.printf
<_FuncPtr object at 0x...>
```

```python
>>> print(windll.kernel32.GetModuleHandleA)
<_FuncPtr object at 0x...>
```

```python
>>> print(windll.kernel32.MyOwnFunction)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
  AttributeError: function 'MyOwnFunction' not found
```

Note that win32 system dlls like `kernel32` and `user32` often export ANSI as well as UNICODE versions of a function. The UNICODE version is exported with an `W` appended to the name, while the ANSI version is exported with an `A` appended to the name. The win32 `GetModuleHandle` function, which returns a module handle for a given module name, has the following C prototype, and a macro is used to expose one of them as `GetModuleHandle` depending on whether UNICODE is defined or not:

```c
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

`windll` does not try to select one of them by magic, you must access the version you need by specifying `GetModuleHandleA` or `GetModuleHandleW` explicitly, and then call it with bytes or string objects respectively.

Sometimes, dlls export functions with names which aren’t valid Python identifiers, like "??2@YAPAXI@Z". In this case you have to use `getattr()` to retrieve the function:

```python
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z")
<_FuncPtr object at 0x...>
```

On Windows, some dlls export functions not by name but by ordinal. These functions can be accessed by indexing the dll object with the ordinal number:
Calling functions

You can call these functions like any other Python callable. This example uses the `time()` function, which returns system time in seconds since the Unix epoch, and the `GetModuleHandleA()` function, which returns a win32 module handle.

This example calls both functions with a NULL pointer (None should be used as the NULL pointer):

```python
>>> print(libc.time(None))
1150640792
>>> print(hex(windll.kernel32.GetModuleHandleA(None)))
0x1d000000
```

`ctypes` tries to protect you from calling functions with the wrong number of arguments or the wrong calling convention. Unfortunately this only works on Windows. It does this by examining the stack after the function returns, so although an error is raised the function has been called:

```python
>>> windll.kernel32.GetModuleHandleA()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  ValueError: Procedure probably called with not enough arguments (4 bytes missing)
```

The same exception is raised when you call an stdcall function with the cdecl calling convention, or vice versa:

```python
>>> cdll.kernel32.GetModuleHandleA(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  ValueError: Procedure probably called with not enough arguments (4 bytes missing)
```

To find out the correct calling convention you have to look into the C header file or the documentation for the function you want to call.

On Windows, `ctypes` uses win32 structured exception handling to prevent crashes from general protection faults when functions are called with invalid argument values:

```python
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
```

16.17. `ctypes` — A foreign function library for Python
There are, however, enough ways to crash Python with `ctypes`, so you should be careful anyway. The `faulthandler` module can be helpful in debugging crashes (e.g., from segmentation faults produced by erroneous C library calls).

None, integers, bytes objects and (unicode) strings are the only native Python objects that can directly be used as parameters in these function calls. None is passed as a C NULL pointer, bytes objects and strings are passed as pointer to the memory block that contains their data (`char *` or `wchar_t *`). Python integers are passed as the platforms default C `int` type, their value is masked to fit into the C type.

Before we move on calling functions with other parameter types, we have to learn more about `ctypes` data types.

## Fundamental data types

`ctypes` defines a number of primitive C compatible data types:

<table>
<thead>
<tr>
<th>ctypes type</th>
<th>C type</th>
<th>Python type</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_bool</td>
<td>_Bool</td>
<td>bool (1)</td>
</tr>
<tr>
<td>c_char</td>
<td>char</td>
<td>1-character bytes object</td>
</tr>
<tr>
<td>c_wchar</td>
<td>wchar_t</td>
<td>1-character string</td>
</tr>
<tr>
<td>c_byte</td>
<td>char</td>
<td>int</td>
</tr>
<tr>
<td>c_ubyte</td>
<td>unsigned char</td>
<td>int</td>
</tr>
<tr>
<td>c_short</td>
<td>short</td>
<td>int</td>
</tr>
<tr>
<td>c_ushort</td>
<td>unsigned short</td>
<td>int</td>
</tr>
<tr>
<td>c_int</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>c_uint</td>
<td>unsigned int</td>
<td>int</td>
</tr>
<tr>
<td>c_ulong</td>
<td>unsigned long</td>
<td>int</td>
</tr>
<tr>
<td>c_ulonglong</td>
<td>__int64 or long long</td>
<td>int</td>
</tr>
<tr>
<td>c_wchar_p</td>
<td>wchar_t *</td>
<td>string or None</td>
</tr>
<tr>
<td>c_char_p</td>
<td>char * (NUL terminated)</td>
<td>int or None</td>
</tr>
</tbody>
</table>

1. The constructor accepts any object with a truth value.

All these types can be created by calling them with an optional initializer of the correct type and value:

```python
>>> c_int()
c_long(0)
>>> c_wchar_p("Hello, World")
c_wchar_p('Hello, World')
>>> c_ushort(-3)
c_ushort(65533)
```

Since these types are mutable, their value can also be changed afterwards:

```python
>>> i = c_int(42)
>>> print(i)
c_long(42)
>>> print(i.value)
42
>>> i.value = -99
>>> print(i.value)
```

1. The constructor accepts any object with a truth value.

All these types can be created by calling them with an optional initializer of the correct type and value:

```python
>>> c_int()
c_long(0)
>>> c_wchar_p("Hello, World")
c_wchar_p('Hello, World')
>>> c_ushort(-3)
c_ushort(65533)
```
Assigning a new value to instances of the pointer types `c_char_p`, `c_wchar_p`, and `c_void_p` changes the memory location they point to, not the contents of the memory block (of course not, because Python bytes objects are immutable):

```
>>> s = "Hello, World"
>>> c_s = c_wchar_p(s)
>>> print(c_s)
c_wchar_p('Hello, World')
>>> c_s.value = "Hi, there"
>>> print(c_s)
c_wchar_p('Hi, there')
>>> print(s)
# first object is unchanged
Hello, World
```

You should be careful, however, not to pass them to functions expecting pointers to mutable memory. If you need mutable memory blocks, ctypes has a `create_string_buffer()` function which creates these in various ways. The current memory block contents can be accessed (or changed) with the `raw` property; if you want to access it as NUL terminated string, use the `value` property:

```
>>> from ctypes import *
>>> p = create_string_buffer(3)  # create a 3 byte buffer, initialized to NUL
3 b'\x00\x00\x00'
>>> p = create_string_buffer(b"Hello")  # create a buffer containing a NUL terminated string
6 b'Hello\x00'
>>> print(repr(p.raw))
b'Hello'
>>> p = create_string_buffer(b"Hello", 10)  # create a 10 byte buffer
>>> print(sizeof(p), repr(p.raw))
10 b'Hello\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> p.value = b"Hi"
>>> print(sizeof(p), repr(p.raw))
10 b'Hi\x00lo\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

The `create_string_buffer()` function replaces the `c_buffer()` function (which is still available as an alias), as well as the `c_string()` function from earlier ctypes releases. To create a mutable memory block containing unicode characters of the C type `wchar_t` use the `create_unicode_buffer()` function.

**Calling functions, continued**

Note that printf prints to the real standard output channel, **not** to `sys.stdout`, so these examples will only work at the console prompt, not from within IDLE or PythonWin:

```
>>> printf = libc.printf
>>> printf(b"Hello, %s\n", b"World!")
Hello, World!
14
>>> printf(b"Hello, %S\n", "World!")
Hello, World!
14
>>> printf(b"%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf(b"%f bottles of beer\n", 42.5)
```
As has been mentioned before, all Python types except integers, strings, and bytes objects have to be wrapped in their corresponding `ctypes` type, so that they can be converted to the required C data type:

```python
>>> printf(b"An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
```

### Calling functions with your own custom data types

You can also customize `ctypes` argument conversion to allow instances of your own classes be used as function arguments. `ctypes` looks for an `_as_parameter_` attribute and uses this as the function argument. Of course, it must be one of integer, string, or bytes:

```python
>>> class Bottles:
...     def __init__(self, number):
...         self._as_parameter_ = number
... >>> bottles = Bottles(42)
>>> printf(b"%d bottles of beer\n", bottles)
42 bottles of beer
```

If you don’t want to store the instance’s data in the `_as_parameter_` instance variable, you could define a `property` which makes the attribute available on request.

### Specifying the required argument types (function prototypes)

It is possible to specify the required argument types of functions exported from DLLs by setting the `argtypes` attribute. `argtypes` must be a sequence of C data types (the `printf` function is probably not a good example here, because it takes a variable number and different types of parameters depending on the format string, on the other hand this is quite handy to experiment with this feature):

```python
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf(b"String '%s', Int %d, Double %f\n", b"Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
```

Specifying a format protects against incompatible argument types (just as a prototype for a C function), and tries to convert the arguments to valid types:

```python
>>> printf(b"%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ArgumentError: argument 2: exceptions.TypeError: wrong type
>>> printf(b"%s %d %f\n", b"X", 2, 3)
X 2 3.000000
```

If you have defined your own classes which you pass to function calls, you have to implement a `from_param()` class method for them to be able to use them in the `argtypes` sequence. The `from_param()` class method
receives the Python object passed to the function call, it should do a typecheck or whatever is needed to make sure this object is acceptable, and then return the object itself, its _as_parameter_ attribute, or whatever you want to pass as the C function argument in this case. Again, the result should be an integer, string, bytes, a ctypes instance, or an object with an _as_parameter_ attribute.

**Return types**

By default functions are assumed to return the C int type. Other return types can be specified by setting the restype attribute of the function object.

Here is a more advanced example, it uses the strchr function, which expects a string pointer and a char, and returns a pointer to a string:

```python
>>> strchr = libc.strchr
>>> strchr(b"abcdef", ord("d"))
8059983
>>> strchr.retype = c_char_p  # c_char_p is a pointer to a string
>>> strchr(b"abcdef", ord("d"))
'b'def'
>>> print(strchr(b"abcdef", ord("x")))
None
>>> 
If you want to avoid the ord("x") calls above, you can set the argtypes attribute, and the second argument will be converted from a single character Python bytes object into a C char:

```python
>>> strchr.retype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr(b"abcdef", b"d")
'def'
>>> strchr(b"abcdef", b"def")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ArgumentError: argument 2: exceptions.TypeError: one character string expected
>>> print(strchr(b"abcdef", b"x"))
None
>>> strchr(b"abcdef", b"d")
'def'
>>> 
You can also use a callable Python object (a function or a class for example) as the restype attribute, if the foreign function returns an integer. The callable will be called with the integer the C function returns, and the result of this call will be used as the result of your function call. This is useful to check for error return values and automatically raise an exception:

```python
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
...
>>> GetModuleHandle.retype = ValidHandle
>>> GetModuleHandle(1)
486539264
>>> GetModuleHandle("something silly")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in ValidHandle
16.17. ctypes — A foreign function library for Python
```
OSError: [Errno 126] The specified module could not be found.
```
>>> WinError is a function which will call Windows FormatMessage() api to get the string representation of an error code, and returns an exception. WinError takes an optional error code parameter, if no one is used, it calls GetLastError() to retrieve it.

Please note that a much more powerful error checking mechanism is available through the errcheck attribute; see the reference manual for details.

Passing pointers (or: passing parameters by reference)

Sometimes a C api function expects a pointer to a data type as parameter, probably to write into the corresponding location, or if the data is too large to be passed by value. This is also known as passing parameters by reference.

ctypes exports the byref() function which is used to pass parameters by reference. The same effect can be achieved with the pointer() function, although pointer() does a lot more work since it constructs a real pointer object, so it is faster to use byref() if you don’t need the pointer object in Python itself:
```
```python
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer(b'\000' * 32)
>>> print(i.value, f.value, repr(s.value))
0 0.0 b''
>>> libc.sscanf(b"1 3.14 Hello", b"%d %f %s",
...
        byref(i), byref(f), s)
3
>>> print(i.value, f.value, repr(s.value))
1 3.1400001049 b'Hello'
```n
Structures and unions

Structures and unions must derive from the Structure and Union base classes which are defined in the ctypes module. Each subclass must define a _fields_ attribute. _fields_ must be a list of 2-tuples, containing a field name and a field type.

The field type must be a ctypes type like c_int, or any other derived ctypes type: structure, union, array, pointer.

Here is a simple example of a POINT structure, which contains two integers named x and y, and also shows how to initialize a structure in the constructor:
```
```python
>>> from ctypes import *
>>> class POINT(Structure):
...    _fields_ = ["x", c_int],
...    ("y", c_int])
...
>>> point = POINT(10, 20)
>>> print(point.x, point.y)
10 20
>>> point = POINT(y=5)
>>> print(point.x, point.y)
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in 
ValueError: too many initializers
```
You can, however, build much more complicated structures. A structure can itself contain other structures by using a structure as a field type.

Here is a RECT structure which contains two POINTs named \textit{upperleft} and \textit{lowerright}:

\begin{verbatim}
>>> class RECT(Structure):
...     _fields_ = ["upperleft", POINT],
...                ("lowerright", POINT]
...
>>> rc = RECT(point)
>>> print(rc.upperleft.x, rc.upperleft.y)
0 5
>>> print(rc.lowerright.x, rc.lowerright.y)
0 0
>>> Nestsed structures can also be initialized in the constructor in several ways:

>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))

Field descriptors can be retrieved from the class, they are useful for debugging because they can provide useful information:

>>> print(POINT.x)
<Field type=c_long, ofs=0, size=4>
>>> print(POINT.y)
<Field type=c_long, ofs=4, size=4>

\textbf{Warning:} ctypes does not support passing unions or structures with bit-fields to functions by value. While this may work on 32-bit x86, it’s not guaranteed by the library to work in the general case. Unions and structures with bit-fields should always be passed to functions by pointer.

Structure/union alignment and byte order

By default, Structure and Union fields are aligned in the same way the C compiler does it. It is possible to override this behavior by specifying a \_pack\_ class attribute in the subclass definition. This must be set to a positive integer and specifies the maximum alignment for the fields. This is what \texttt{#pragma pack(n)} also does in MSVC.

c\texttt{types} uses the native byte order for Structures and Unions. To build structures with non-native byte order, you can use one of the BigEndianStructure, LittleEndianStructure, BigEndianUnion, and LittleEndianUnion base classes. These classes cannot contain pointer fields.

Bit fields in structures and unions

It is possible to create structures and unions containing bit fields. Bit fields are only possible for integer fields, the bit width is specified as the third item in the \_fields\_ tuples:

\begin{verbatim}
>>> class Int(Structure):
...     _fields_ = ["first_16", c_int, 16],
...                ("second_16", c_int, 16]
...
>>> print(Int.first_16)
<Field type=c_long, ofs=0, bits=16>
>>> print(Int.second_16)
<Field type=c_long, ofs=0, bits=16>
\end{verbatim}
Arrays

Arrays are sequences, containing a fixed number of instances of the same type.

The recommended way to create array types is by multiplying a data type with a positive integer:

```python
TenPointsArrayType = POINT * 10
```

Here is an example of an somewhat artificial data type, a structure containing 4 POINTs among other stuff:

```python
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
... >>>
>>> class MyStruct(Structure):
...     _fields_ = ["a", c_int], ("b", c_float), ("point_array", POINT * 4)]
>>>
>>> print(len(MyStruct().point_array))
4
>>> Instances are created in the usual way, by calling the class:

```python
arr = TenPointsArrayType()
for pt in arr:
    print(pt.x, pt.y)
```

The above code print a series of 0 0 lines, because the array contents is initialized to zeros.

Initializers of the correct type can also be specified:

```python
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print(ii)
<c_long_Array_10 object at 0x...>
>>> for i in ii: print(i, end=" ")
... 1 2 3 4 5 6 7 8 9 10
>>>```

Pointers

Pointer instances are created by calling the `pointer()` function on a `ctypes` type:

```python
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>```

Pointer instances have a `contents` attribute which returns the object to which the pointer points, the `i` object above:

```python
>>> pi.contents
c_long(42)
>>>```

Note that `ctypes` does not have OOR (original object return), it constructs a new, equivalent object each time you retrieve an attribute:

```python
>>> pi.contents is i
False
```
Assigning another `c_int` instance to the pointer’s contents attribute would cause the pointer to point to the memory location where this is stored:

```python
>>> i = c_int(99)
>>> pi.contents = i
```

```c
    c_long(99)
```

Pointer instances can also be indexed with integers:

```python
>>> pi[0]
99
```

Assigning to an integer index changes the pointed to value:

```python
>>> print(i)
    c_long(99)
>>> pi[0] = 22
>>> print(i)
    c_long(22)
```

It is also possible to use indexes different from 0, but you must know what you’re doing, just as in C: You can access or change arbitrary memory locations. Generally you only use this feature if you receive a pointer from a C function, and you know that the pointer actually points to an array instead of a single item.

Behind the scenes, the `pointer()` function does more than simply create pointer instances, it has to create `pointer` types first. This is done with the `POINTER()` function, which accepts any `ctypes` type, and returns a new type:

```python
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_long'>
```

Calling the pointer type without an argument creates a NULL pointer. NULL pointers have a `False` boolean value:

```python
>>> null_ptr = POINTER(c_int)()
>>> print(bool(null_ptr))
False
```

c`types` checks for NULL when dereferencing pointers (but dereferencing invalid non-NULL pointers would crash Python):

```python
>>> null_ptr[0]
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
```

```python
>>> null_ptr[0] = 1234
```

Traceback (most recent call last):
Type conversions

Usually, ctypes does strict type checking. This means, if you have\texttt{POINTER(c\_int)} in the \texttt{argtypes} list of a function or as the type of a member field in a structure definition, only instances of exactly the same type are accepted. There are some exceptions to this rule, where ctypes accepts other objects. For example, you can pass compatible array instances instead of pointer types. So, for \texttt{POINTER(c\_int)}, ctypes accepts an array of \texttt{c\_int}:

\begin{verbatim}
>>> class Bar(Structure):
...     _fields_ = ["count", c\_int], ("values", POINTER(c\_int))
...
>>> bar = Bar()
>>> bar.values = (c\_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print(bar.values[i])
1
2
3
...
\end{verbatim}

In addition, if a function argument is explicitly declared to be a pointer type (such as \texttt{POINTER(c\_int)}) in \texttt{argtypes}, an object of the pointed type (\texttt{c\_int} in this case) can be passed to the function. ctypes will apply the required \texttt{byref()} conversion in this case automatically.

To set a \texttt{POINTER} type field to \texttt{NULL}, you can assign \texttt{None}:

\begin{verbatim}
>>> bar.values = None
>>>...
\end{verbatim}

Sometimes you have instances of incompatible types. In C, you can cast one type into another type. ctypes provides a \texttt{cast()} function which can be used in the same way. The \texttt{Bar} structure defined above accepts \texttt{POINTER(c\_int)} pointers or \texttt{c\_int} arrays for its \texttt{values} field, but not instances of other types:

\begin{verbatim}
>>> bar.values = (c\_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: incompatible types, c\_byte\_Array\_4 instance instead of LP\_c\_long instance
>>>...
\end{verbatim}

For these cases, the \texttt{cast()} function is handy.

The \texttt{cast()} function can be used to cast a ctypes instance into a pointer to a different ctypes data type. \texttt{cast()} takes two parameters, a ctypes object that is or can be converted to a pointer of some kind, and a ctypes pointer type. It returns an instance of the second argument, which references the same memory block as the first argument:

\begin{verbatim}
>>> a = (c\_byte * 4)()
>>> cast(a, POINTER(c\_int))
<ctypes.LP\_c\_long object at ...>
>>>...
\end{verbatim}

So, \texttt{cast()} can be used to assign to the \texttt{values} field of \texttt{Bar} the structure:

\begin{verbatim}
>>> bar = Bar()
>>> bar.values = cast((c\_byte * 4)(), POINTER(c\_int))
>>> print(bar.values[0])
0
>>>...
\end{verbatim}
Incomplete Types

Incomplete Types are structures, unions or arrays whose members are not yet specified. In C, they are specified by forward declarations, which are defined later:

```c
struct cell; /* forward declaration */

struct cell {
    char *name;
    struct cell *next;
};
```

The straightforward translation into ctypes code would be this, but it does not work:

```python
>>> class cell(Structure):
...     _fields_ = [('name', c_char_p),
...                 ('next', POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
```

because the new class `cell` is not available in the class statement itself. In ctypes, we can define the `cell` class and set the `_fields_` attribute later, after the class statement:

```python
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [('name', c_char_p),
...                   ('next', POINTER(cell))]
```

Let's try it. We create two instances of `cell`, and let them point to each other, and finally follow the pointer chain a few times:

```python
>>> c1 = cell()
>>> c1.name = "foo"
>>> c2 = cell()
>>> c2.name = "bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print(p.name, end=" ")
...     p = p.next[0]
... foo bar foo bar foo bar foo bar
```

Callback functions

ctypes allows to create C callable function pointers from Python callables. These are sometimes called callback functions.

First, you must create a class for the callback function. The class knows the calling convention, the return type, and the number and types of arguments this function will receive.

The `CFUNCTYPE()` factory function creates types for callback functions using the cdecl calling convention. On Windows, the `WINFUNCTYPE()` factory function creates types for callback functions using the stdcall calling convention.
calling convention.

Both of these factory functions are called with the result type as first argument, and the callback functions expected argument types as the remaining arguments.

I will present an example here which uses the standard C library’s qsort() function, that is used to sort items with the help of a callback function. qsort() will be used to sort an array of integers:

```python
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None

qsort() must be called with a pointer to the data to sort, the number of items in the data array, the size of one item, and a pointer to the comparison function, the callback. The callback will then be called with two pointers to items, and it must return a negative integer if the first item is smaller than the second, a zero if they are equal, and a positive integer otherwise.

So our callback function receives pointers to integers, and must return an integer. First we create the type for the callback function:

```python
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>> def py_cmp_func(a, b):
...    print("py_cmp_func", a[0], b[0])
...    return 0
... qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
```

The result:

```
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
```

Now we can actually compare the two items and return a useful result:

```python
>>> def py_cmp_func(a, b):
...    print("py_cmp_func", a[0], b[0])
...    return a[0] - b[0]
... qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
```

As we can easily check, our array is sorted now:

```python
>>> for i in ia: print(i, end=" ")
1 5 7 33 99
>>> 
```
Important note for callback functions:
Make sure you keep references to `CFUNCTYPE()` objects as long as they are used from C code. `ctypes` doesn’t, and if you don’t, they may be garbage collected, crashing your program when a callback is made.

Accessing values exported from dlls

Some shared libraries not only export functions, they also export variables. An example in the Python library itself is the `Py_OptimizeFlag`, an integer set to 0, 1, or 2, depending on the `-O` or `-OO` flag given on startup.

`ctypes` can access values like this with the `in_dll()` class methods of the type. `pythonapi` is a predefined symbol giving access to the Python C api:

```python
>>> opt_flag = c_int.in_dll(pythonapi, "Py_OptimizeFlag")
>>> print(opt_flag)
c_long(0)
```

If the interpreter would have been started with `-O`, the sample would have printed `c_long(1)`, or `c_long(2)` if `-OO` would have been specified.

An extended example which also demonstrates the use of pointers accesses the `PyImport_FrozenModules` pointer exported by Python.

Quoting the docs for that value:

```
This pointer is initialized to point to an array of struct _frozen records, terminated by one whose members are all NULL or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.
```

So manipulating this pointer could even prove useful. To restrict the example size, we show only how this table can be read with `ctypes`:

```python
>>> from ctypes import *

>>> class struct_frozen(Structure):
...    _fields_ = [("name", c_char_p),
...                ("code", POINTER(c_ubyte)),
...                ("size", c_int)]
...

>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "PyImport_FrozenModules")

Since `table` is a pointer to the array of `struct_frozen` records, we can iterate over it, but we just have to make sure that our loop terminates, because pointers have no size. Sooner or later it would probably crash with an access violation or whatever, so it’s better to break out of the loop when we hit the NULL entry:

```python
>>> for item in table:
...    print(item.name, item.size)
...    if item.name is None:
...        break
...__hello__ 104
__phello__ -104
__phello__.spam 104
None 0
```
The fact that standard Python has a frozen module and a frozen package (indicated by the negative size member) is not well known, it is only used for testing. Try it out with `import __hello__` for example.

**Surprises**

There are some edges in `ctypes` where you might expect something other than what actually happens. Consider the following example:

```python
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
3 4 3 4
```

Hm. We certainly expected the last statement to print 3 4 1 2. What happened? Here are the steps of the `rc.a, rc.b = rc.b, rc.a` line above:

```python
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
```

Note that `temp0` and `temp1` are objects still using the internal buffer of the `rc` object above. So executing `rc.a = temp0` copies the buffer contents of `temp0` into `rc`'s buffer. This, in turn, changes the contents of `temp1`. So, the last assignment `rc.b = temp1` doesn't have the expected effect.

Keep in mind that retrieving sub-objects from Structure, Unions, and Arrays doesn't `copy` the sub-object, instead it retrieves a wrapper object accessing the root-object's underlying buffer.

Another example that may behave different from what one would expect is this:

```python
>>> s = c_char_p()
>>> s.value = "abc def ghi"
>>> s.value
'abc def ghi'
>>> s.value is s.value
False
```

Why is it printing `False`? `ctypes` instances are objects containing a memory block plus some descriptors accessing the contents of the memory. Storing a Python object in the memory block does not store the object itself, instead the contents of the object is stored. Accessing the contents again constructs a new Python object each time!

**Variable-sized data types**

`ctypes` provides some support for variable-sized arrays and structures.
The resize() function can be used to resize the memory buffer of an existing ctypes object. The function takes
the object as first argument, and the requested size in bytes as the second argument. The memory block cannot
be made smaller than the natural memory block specified by the objects type, a ValueError is raised if this is
tried:

```python
>>> short_array = (c_short * 4)()
>>> print(sizeof(short_array))
8
>>> resize(short_array, 4)
Traceback (most recent call last):
  ...:
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
```

This is nice and fine, but how would one access the additional elements contained in this array? Since the type
still only knows about 4 elements, we get errors accessing other elements:

```python
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
  ...:
IndexError: invalid index
```

Another way to use variable-sized data types with ctypes is to use the dynamic nature of Python, and (re-)define
the data type after the required size is already known, on a case by case basis.

### 16.17.2 ctypes reference

#### Finding shared libraries

When programming in a compiled language, shared libraries are accessed when compiling/linking a program, and
when the program is run.

The purpose of the find_library() function is to locate a library in a way similar to what the compiler does
(on platforms with several versions of a shared library the most recent should be loaded), while the ctypes library
loaders act like when a program is run, and call the runtime loader directly.

The ctypes.util module provides a function which can help to determine the library to load.

callctypes.util.find_library(name)

Try to find a library and return a pathname. name is the library name without any prefix like lib, suffix like
.so, .dylib or version number (this is the form used for the posix linker option -l). If no library can be
found, returns None.

The exact functionality is system dependent.

On Linux, find_library() tries to run external programs (/sbin/ldconfig, gcc, and objdump) to find
the library file. It returns the filename of the library file. Here are some examples:

```python
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
```
On OS X, `find_library()` tries several predefined naming schemes and paths to locate the library, and returns a full pathname if successful:

```python
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
```

On Windows, `find_library()` searches along the system search path, and returns the full pathname, but since there is no predefined naming scheme a call like `find_library("c")` will fail and return `None`. If wrapping a shared library with `ctypes`, it may be better to determine the shared library name at development time, and hardcode that into the wrapper module instead of using `find_library()` to locate the library at runtime.

### Loading shared libraries

There are several ways to load shared libraries into the Python process. One way is to instantiate one of the following classes:

```python
class ctypes.CDLL(name, mode=DEFAULT_MODE, handle=None,
                  use_errno=False, use_last_error=False)
```

Instances of this class represent loaded shared libraries. Functions in these libraries use the standard C calling convention, and are assumed to return `int`.

```python
class ctypes.OleDLL(name, mode=DEFAULT_MODE, handle=None,
                   use_errno=False, use_last_error=False)
```

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the stdcall calling convention, and are assumed to return the windows specific HRESULT code. HRESULT values contain information specifying whether the function call failed or succeeded, together with additional error code. If the return value signals a failure, an `OSError` is automatically raised. Changed in version 3.3: `WindowsError` used to be raised.

```python
class ctypes.WinDLL(name, mode=DEFAULT_MODE, handle=None,
                    use_errno=False, use_last_error=False)
```

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the stdcall calling convention, and are assumed to return `int` by default.

On Windows CE only the standard calling convention is used, for convenience the `WinDLL` and `OleDLL` use the standard calling convention on this platform.

The Python `global interpreter lock` is released before calling any function exported by these libraries, and reacquired afterwards.

```python
class ctypes.PyDLL(name, mode=DEFAULT_MODE, handle=None)
```

Instances of this class behave like `CDLL` instances, except that the Python GIL is not released during the function call, and after the function execution the Python error flag is checked. If the error flag is set, a Python exception is raised.

Thus, this is only useful to call Python C API functions directly.

All these classes can be instantiated by calling them with at least one argument, the pathname of the shared library. If you have an existing handle to an already loaded shared library, it can be passed as the `handle` named parameter, otherwise the underlying platforms `dlopen` or `LoadLibrary` function is used to load the library into the process, and to get a handle to it.
The `mode` parameter can be used to specify how the library is loaded. For details, consult the `dlopen(3)` manpage, on Windows, `mode` is ignored.

The `use_errno` parameter, when set to True, enables a ctypes mechanism that allows to access the system `errno` error number in a safe way. ctypes maintains a thread-local copy of the systems `errno` variable; if you call foreign functions created with `use_errno=True` then the `errno` value before the function call is swapped with the ctypes private copy, the same happens immediately after the function call.

The function `ctypes.get_errno()` returns the value of the ctypes private copy, and the function `ctypes.set_errno()` changes the ctypes private copy to a new value and returns the former value.

The `use_last_error` parameter, when set to True, enables the same mechanism for the Windows error code which is managed by the `GetLastError()` and ` SetLastError()` Windows API functions; `ctypes.get_last_error()` and `ctypes.set_last_error()` are used to request and change the ctypes private copy of the windows error code.

The following public attributes are available, their name starts with an underscore to not clash with exported function names:

---

### PyDLL

- **._handle**
  - The system handle used to access the library.

- **._name**
  - The name of the library passed in the constructor.

Shared libraries can also be loaded by using one of the prefabricated objects, which are instances of the `LibraryLoader` class, either by calling the `LoadLibrary()` method, or by retrieving the library as attribute of the loader instance.

```python
class ctypes.LibraryLoader(dlltype)
    Class which loads shared libraries. `dlltype` should be one of the CDLL, PyDLL, WinDLL, or OleDLL types.
    `__getattr__()` has special behavior: It allows to load a shared library by accessing it as attribute of a library loader instance. The result is cached, so repeated attribute accesses return the same library each time.

    `LoadLibrary(name)`
    Load a shared library into the process and return it. This method always returns a new instance of the library.
```

These prefabricated library loaders are available:

- **ctypes.cdll**
  - Creates CDLL instances.

- **ctypes.windll**
  - Windows only: Creates WinDLL instances.

- **ctypes.oledll**
  - Windows only: Creates OleDLL instances.

- **ctypes.pydll**
  - Creates PyDLL instances.
For accessing the C Python api directly, a ready-to-use Python shared library object is available:

```python
import ctypes

ctypes.pythonapi
```

An instance of `PyDLL` that exposes Python C API functions as attributes. Note that all these functions are assumed to return C `int`, which is of course not always the truth, so you have to assign the correct `restype` attribute to use these functions.

### Foreign functions

As explained in the previous section, foreign functions can be accessed as attributes of loaded shared libraries. The function objects created in this way by default accept any number of arguments, accept any ctypes data instances as arguments, and return the default result type specified by the library loader. They are instances of a private class:

```python
class ctypes._FuncPtr
```

Base class for C callable foreign functions.

Instances of foreign functions are also C compatible data types; they represent C function pointers.

This behavior can be customized by assigning to special attributes of the foreign function object.

- **restype**
  Assign a ctypes type to specify the result type of the foreign function. Use `None` for `void`, a function not returning anything.

  It is possible to assign a callable Python object that is not a ctypes type, in this case the function is assumed to return a C `int`, and the callable will be called with this integer, allowing to do further processing or error checking. Using this is deprecated, for more flexible post processing or error checking use a ctypes data type as `restype` and assign a callable to the `errcheck` attribute.

- **argtypes**
  Assign a tuple of ctypes types to specify the argument types that the function accepts. Functions using the `stdcall` calling convention can only be called with the same number of arguments as the length of this tuple; functions using the C calling convention accept additional, unspecified arguments as well.

  When a foreign function is called, each actual argument is passed to the `from_param()` class method of the items in the `argtypes` tuple, this method allows to adapt the actual argument to an object that the foreign function accepts. For example, a `c_char_p` item in the `argtypes` tuple will convert a string passed as argument into a bytes object using ctypes conversion rules.

  New: It is now possible to put items in `argtypes` which are not ctypes types, but each item must have a `from_param()` method which returns a value usable as argument (integer, string, ctypes instance). This allows to define adapters that can adapt custom objects as function parameters.

- **errcheck**
  Assign a Python function or another callable to this attribute. The callable will be called with three or more arguments:

  ```python
callable (result, func, arguments)
  ```

  `result` is what the foreign function returns, as specified by the `restype` attribute.

  `func` is the foreign function object itself, this allows to reuse the same callable object to check or post process the results of several functions.

  `arguments` is a tuple containing the parameters originally passed to the function call, this allows to specialize the behavior on the arguments used.

  The object that this function returns will be returned from the foreign function call, but it can also check the result value and raise an exception if the foreign function call failed.

### Exception

```python
ctypes.ArgumentError
```

This exception is raised when a foreign function call cannot convert one of the passed arguments.
**Function prototypes**

Foreign functions can also be created by instantiating function prototypes. Function prototypes are similar to function prototypes in C; they describe a function (return type, argument types, calling convention) without defining an implementation. The factory functions must be called with the desired result type and the argument types of the function.

```python
ctypes.CFUNCTYPE (restype, *argtypes, use_errno=False, use_last_error=False)
```

The returned function prototype creates functions that use the standard C calling convention. The function will release the GIL during the call. If `use_errno` is set to True, the ctypes private copy of the system `errno` variable is exchanged with the real `errno` value before and after the call; `use_last_error` does the same for the Windows error code.

```python
ctypes.WINFUNCTYPE (restype, *argtypes, use_errno=False, use_last_error=False)
```

Windows only: The returned function prototype creates functions that use the `stdcall` calling convention, except on Windows CE where `WINFUNCTYPE()` is the same as `CFUNCTYPE()`. The function will release the GIL during the call. `use_errno` and `use_last_error` have the same meaning as above.

```python
ctypes.PYFUNCTYPE (restype, *argtypes)
```

The returned function prototype creates functions that use the Python calling convention. The function will not release the GIL during the call.

Function prototypes created by these factory functions can be instantiated in different ways, depending on the type and number of the parameters in the call:

```python
prototype (address)
```

Returns a foreign function at the specified address which must be an integer.

```python
prototype (callable)
```

Create a C callable function (a callback function) from a Python `callable`.

```python
prototype (func_spec[, paramflags])
```

Returns a foreign function exported by a shared library. `func_spec` must be a 2-tuple `(name_or_ordinal, library)`. The first item is the name of the exported function as string, or the ordinal of the exported function as small integer. The second item is the shared library instance.

```python
prototype (vtbl_index, name[, paramflags[, iid]])
```

Returns a foreign function that will call a COM method. `vtbl_index` is the index into the virtual function table, a small non-negative integer. `name` is name of the COM method. `iid` is an optional pointer to the interface identifier which is used in extended error reporting.

COM methods use a special calling convention: They require a pointer to the COM interface as first argument, in addition to those parameters that are specified in the `argtypes` tuple.

The optional `paramflags` parameter creates foreign function wrappers with much more functionality than the features described above.

`paramflags` must be a tuple of the same length as `argtypes`.

Each item in this tuple contains further information about a parameter, it must be a tuple containing one, two, or three items.

The first item is an integer containing a combination of direction flags for the parameter:

1. Specifies an input parameter to the function.
2. Output parameter. The foreign function fills in a value.
4. Input parameter which defaults to the integer zero.

The optional second item is the parameter name as string. If this is specified, the foreign function can be called with named parameters.

The optional third item is the default value for this parameter.

This example demonstrates how to wrap the Windows `MessageBoxA` function so that it supports default parameters and named arguments. The C declaration from the windows header file is this:
WINUSERAPI int WINAPI
MessageBoxA(
    HWND hWnd ,
    LPCSTR lpText ,
    LPCSTR lpCaption ,
    UINT uType);

Here is the wrapping with ctypes:

```python
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCSTR, UINT

>>> prototype = WINFUNCTYPE(c_int, HWND, LPCSTR, LPCSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption", None), (1, "flags", 0)
>>> MessageBox = prototype(('MessageBoxA', windll.user32), paramflags)

The MessageBox foreign function can now be called in these ways:

```python
>>> MessageBox()
```
```python
>>> MessageBox(text="Spam, spam, spam")
```
```python
>>> MessageBox(flags=2, text="foo bar")
```

A second example demonstrates output parameters. The win32 GetWindowRect function retrieves the dimensions of a specified window by copying them into RECT structure that the caller has to supply. Here is the C declaration:

WINUSERAPI BOOL WINAPI
GetWindowRect(
    HWND hWnd,
    LPRECT lpRect);

Here is the wrapping with ctypes:

```python
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT

>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(('GetWindowRect', windll.user32), paramflags)

Functions with output parameters will automatically return the output parameter value if there is a single one, or a tuple containing the output parameter values when there are more than one, so the GetWindowRect function now returns a RECT instance, when called.

Output parameters can be combined with the errcheck protocol to do further output processing and error checking. The win32 GetWindowRect api function returns a BOOL to signal success or failure, so this function could do the error checking, and raises an exception when the api call failed:

```python
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...     return args

>>> GetWindowRect.errcheck = errcheck

If the errcheck function returns the argument tuple it receives unchanged, ctypes continues the normal processing it does on the output parameters. If you want to return a tuple of window coordinates instead of a RECT instance, you can retrieve the fields in the function and return them instead, the normal processing will no longer take place:

```python
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
```
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
... >>> GetWindowRect.errcheck = errcheck

Utility functions

ctypes.\texttt{addressof} (\texttt{obj})
Returns the address of the memory buffer as integer. \texttt{obj} must be an instance of a ctypes type.

ctypes.\texttt{alignment} (\texttt{obj\_or\_type})
Returns the alignment requirements of a ctypes type. \texttt{obj\_or\_type} must be a ctypes type or instance.

ctypes.\texttt{byref} (\texttt{obj[, offset]})
Returns a light-weight pointer to \texttt{obj}, which must be an instance of a ctypes type. \texttt{offset} defaults to zero, and must be an integer that will be added to the internal pointer value.

\texttt{byref(obj, offset)} corresponds to this C code:

\begin{verbatim}
((char *)\&obj) + offset
\end{verbatim}

The returned object can only be used as a foreign function call parameter. It behaves similar to \texttt{pointer(obj)}, but the construction is a lot faster.

ctypes.\texttt{cast} (\texttt{obj, type})
This function is similar to the cast operator in C. It returns a new instance of \texttt{type} which points to the same memory block as \texttt{obj}. \texttt{type} must be a pointer type, and \texttt{obj} must be an object that can be interpreted as a pointer.

ctypes.\texttt{create\_string\_buffer} (\texttt{init\_or\_size, size=None})
This function creates a mutable character buffer. The returned object is a ctypes array of \texttt{c\_char}.

\texttt{init\_or\_size} must be an integer which specifies the size of the array, or a bytes object which will be used to initialize the array items.

If a bytes object is specified as first argument, the buffer is made one item larger than its length so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows to specify the size of the array if the length of the bytes should not be used.

ctypes.\texttt{create\_unicode\_buffer} (\texttt{init\_or\_size, size=None})
This function creates a mutable unicode character buffer. The returned object is a ctypes array of \texttt{c\_wchar}.

\texttt{init\_or\_size} must be an integer which specifies the size of the array, or a string which will be used to initialize the array items.

If a string is specified as first argument, the buffer is made one item larger than the length of the string so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows to specify the size of the array if the length of the string should not be used.

ctypes.\texttt{DllCanUnloadNow}()
Windows only: This function is a hook which allows to implement in-process COM servers with ctypes. It is called from the DllCanUnloadNow function that the _ctypes extension dll exports.

ctypes.\texttt{DllGetClassObject}()
Windows only: This function is a hook which allows to implement in-process COM servers with ctypes. It is called from the DllGetClassObject function that the _ctypes extension dll exports.

ctypes.util.\texttt{find\_library} (\texttt{name})
Try to find a library and return a pathname. \texttt{name} is the library name without any prefix like lib, suffix like .so, .dylib or version number (this is the form used for the posix linker option \texttt{-l}). If no library can be found, returns \texttt{None}.

The exact functionality is system dependent.
The Python Library Reference, Release 3.3.3

```python
ctypes.util.find_msvcrt()

Windows only: return the filename of the VC runtype library used by Python, and by the extension modules. If the name of the library cannot be determined, None is returned.

If you need to free memory, for example, allocated by an extension module with a call to the free(void *), it is important that you use the function in the same library that allocated the memory.
```

```python
ctypes.FormatError([code])

Windows only: Returns a textual description of the error code code. If no error code is specified, the last error code is used by calling the Windows api function GetLastError.
```

```python
ctypes.GetLastError()

Windows only: Returns the last error code set by Windows in the calling thread. This function calls the Windows GetLastError() function directly, it does not return the ctypes-private copy of the error code.
```

```python
ctypes.get_errno()

Returns the current value of the ctypes-private copy of the system errno variable in the calling thread.
```

```python
ctypes.get_last_error()

Windows only: returns the current value of the ctypes-private copy of the system LastError variable in the calling thread.
```

```python
ctypes.memmove(dst, src, count)

Same as the standard C memmove library function: copies count bytes from src to dst. dst and src must be integers or ctypes instances that can be converted to pointers.
```

```python
ctypes.memset(dst, c, count)

Same as the standard C memset library function: fills the memory block at address dst with count bytes of value c. dst must be an integer specifying an address, or a ctypes instance.
```

```python
ctypes.POINTER(type)

This factory function creates and returns a new ctypes pointer type. Pointer types are cached and reused internally, so calling this function repeatedly is cheap. type must be a ctypes type.
```

```python
ctypes.pointer(obj)

This function creates a new pointer instance, pointing to obj. The returned object is of the type POINTER(type(obj)).

Note: If you just want to pass a pointer to an object to a foreign function call, you should use byref(obj) which is much faster.
```

```python
ctypes.resize(obj, size)

This function resizes the internal memory buffer of obj, which must be an instance of a ctypes type. It is not possible to make the buffer smaller than the native size of the objects type, as given by sizeof(type(obj)), but it is possible to enlarge the buffer.
```

```python
ctypes.set_errno(value)

Set the current value of the ctypes-private copy of the system errno variable in the calling thread to value and return the previous value.
```

```python
ctypes.set_last_error(value)

Windows only: set the current value of the ctypes-private copy of the system LastError variable in the calling thread to value and return the previous value.
```

```python
ctypes.sizeof(obj_or_type)

Returns the size in bytes of a ctypes type or instance memory buffer. Does the same as the C sizeof operator.
```

```python
ctypes.string_at(address, size=-1)

This function returns the C string starting at memory address address as a bytes object. If size is specified, it is used as size, otherwise the string is assumed to be zero-terminated.
```

```python
ctypes.WinError(code=None, descr=None)

Windows only: this function is probably the worst-named thing in ctypes. It creates an instance of OSErrord. If code is not specified, GetLastError is called to determine the error code. If descr is not specified,
FormatError() is called to get a textual description of the error. Changed in version 3.3: An instance of WindowsError used to be created.

c ctypes.wstring_at(address, size=-1)
This function returns the wide character string starting at memory address address as a string. If size is specified, it is used as the number of characters of the string, otherwise the string is assumed to be zero-terminated.

Data types

class ctypes._CData
This non-public class is the common base class of all ctypes data types. Among other things, all ctypes type instances contain a memory block that hold C compatible data; the address of the memory block is returned by the addressof() helper function. Another instance variable is exposed as _objects; this contains other Python objects that need to be kept alive in case the memory block contains pointers.

Common methods of ctypes data types, these are all class methods (to be exact, they are methods of the metaclass):

from_buffer(source[, offset])
This method returns a ctypes instance that shares the buffer of the source object. The source object must support the writeable buffer interface. The optional offset parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a ValueError is raised.

from_buffer_copy(source[, offset])
This method creates a ctypes instance, copying the buffer from the source object buffer which must be readable. The optional offset parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a ValueError is raised.

from_address(address)
This method returns a ctypes type instance using the memory specified by address which must be an integer.

from_param(obj)
This method adapts obj to a ctypes type. It is called with the actual object used in a foreign function call when the type is present in the foreign function’s argtypes tuple; it must return an object that can be used as a function call parameter.

All ctypes data types have a default implementation of this classmethod that normally returns obj if that is an instance of the type. Some types accept other objects as well.

in_dll(library, name)
This method returns a ctypes type instance exported by a shared library. name is the name of the symbol that exports the data, library is the loaded shared library.

Common instance variables of ctypes data types:

_b_base_
Sometimes ctypes data instances do not own the memory block they contain, instead they share part of the memory block of a base object. The _b_base_ read-only member is the root ctypes object that owns the memory block.

_b_needsfree_
This read-only variable is true when the ctypes data instance has allocated the memory block itself, false otherwise.

_objects
This member is either None or a dictionary containing Python objects that need to be kept alive so that the memory block contents is kept valid. This object is only exposed for debugging; never modify the contents of this dictionary.
Fundamental data types

class ctypes._SimpleCData

This non-public class is the base class of all fundamental ctypes data types. It is mentioned here because it contains the common attributes of the fundamental ctypes data types. _SimpleCData is a subclass of _CData, so it inherits their methods and attributes. ctypes data types that are not and do not contain pointers can now be pickled.

Instances have a single attribute:

value

This attribute contains the actual value of the instance. For integer and pointer types, it is an integer, for character types, it is a single character bytes object or string, for character pointer types it is a Python bytes object or string.

When the value attribute is retrieved from a ctypes instance, usually a new object is returned each time. ctypes does not implement original object return, always a new object is constructed. The same is true for all other ctypes object instances.

Fundamental data types, when returned as foreign function call results, or, for example, by retrieving structure field members or array items, are transparently converted to native Python types. In other words, if a foreign function has a restype of c_char_p, you will always receive a Python bytes object, not a c_char_p instance.

Subclasses of fundamental data types do not inherit this behavior. So, if a foreign functions restype is a subclass of c_void_p, you will receive an instance of this subclass from the function call. Of course, you can get the value of the pointer by accessing the value attribute.

These are the fundamental ctypes data types:

class ctypes.c_byte

Represents the C signed char datatype, and interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_char

Represents the C char datatype, and interprets the value as a single character. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.c_char_p

Represents the C char * datatype when it points to a zero-terminated string. For a general character pointer that may also point to binary data, POINTER(c_char) must be used. The constructor accepts an integer address, or a bytes object.

class ctypes.c_double

Represents the C double datatype. The constructor accepts an optional float initializer.

class ctypes.c_longdouble

Represents the C long double datatype. The constructor accepts an optional float initializer. On platforms where sizeof(long double) == sizeof(double) it is an alias to c_double.

class ctypes.c_float

Represents the C float datatype. The constructor accepts an optional float initializer.

class ctypes.c_int

Represents the C signed int datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where sizeof(int) == sizeof(long) it is an alias to c_long.

class ctypes.c_int8

Represents the C 8-bit signed int datatype. Usually an alias for c_byte.

class ctypes.c_int16

Represents the C 16-bit signed int datatype. Usually an alias for c_short.

class ctypes.c_int32

Represents the C 32-bit signed int datatype. Usually an alias for c_int.
class ctypes.c_int64
   Represents the C 64-bit signed int datatype. Usually an alias for c_longlong.

class ctypes.c_long
   Represents the C signed long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_longlong
   Represents the C signed long long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_short
   Represents the C signed short datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_size_t
   Represents the C size_t datatype.

class ctypes.c_ssize_t
   Represents the C ssize_t datatype. New in version 3.2.

class ctypes.c_ubyte
   Represents the C unsigned char datatype, it interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_uint
   Represents the C unsigned int datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where sizeof(int) == sizeof(long) it is an alias for c_ulong.

class ctypes.c_uint8
   Represents the C 8-bit unsigned int datatype. Usually an alias for c_ubyte.

class ctypes.c_uint16
   Represents the C 16-bit unsigned int datatype. Usually an alias for c_ushort.

class ctypes.c_uint32
   Represents the C 32-bit unsigned int datatype. Usually an alias for c_uint.

class ctypes.c_uint64
   Represents the C 64-bit unsigned int datatype. Usually an alias for c_ulonglong.

class ctypes.c_ulong
   Represents the C unsigned long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_ulonglong
   Represents the C unsigned long long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_ushort
   Represents the C unsigned short datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_void_p
   Represents the C void * type. The value is represented as integer. The constructor accepts an optional integer initializer.

class ctypes.c_wchar
   Represents the C wchar_t datatype, and interprets the value as a single character unicode string. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.c_wchar_p
   Represents the C wchar_t * datatype, which must be a pointer to a zero-terminated wide character string. The constructor accepts an integer address, or a string.
class ctypes.c_bool
    Represent the C bool datatype (more accurately, _Bool from C99). Its value can be True or False, and
    the constructor accepts any object that has a truth value.

class ctypes.HRESULT
    Windows only: Represents a HRESULT value, which contains success or error information for a function
    or method call.

class ctypes.py_object
    Represents the C PyObject * datatype. Calling this without an argument creates a NULL PyObject *
    pointer.

The ctypes.wintypes module provides quite some other Windows specific data types, for example HWND,
WPARAM, or DWORD. Some useful structures like MSG or RECT are also defined.

Structured data types

class ctypes.Union(*args, **kw)
    Abstract base class for unions in native byte order.

class ctypes.BigEndianStructure(*args, **kw)
    Abstract base class for structures in big endian byte order.

class ctypes.LittleEndianStructure(*args, **kw)
    Abstract base class for structures in little endian byte order.

Structures with non-native byte order cannot contain pointer type fields, or any other data types containing pointer
type fields.

class ctypes.Structure(*args, **kw)
    Abstract base class for structures in native byte order.

Concrete structure and union types must be created by subclassing one of these types, and at least define a
_fields_ class variable. ctypes will create descriptors which allow reading and writing the fields by
direct attribute accesses. These are the

_fields_

A sequence defining the structure fields. The items must be 2-tuples or 3-tuples. The first item is the
name of the field, the second item specifies the type of the field; it can be any ctypes data type.

For integer type fields like c_int, a third optional item can be given. It must be a small positive
integer defining the bit width of the field.

Field names must be unique within one structure or union. This is not checked, only one field can be
accessed when names are repeated.

It is possible to define the _fields_ class variable after the class statement that defines the Structure
subclass, this allows to create data types that directly or indirectly reference themselves:

class List(Structure):
    pass
List._fields_ = [("pnext", POINTER(List)),
                 ...]

The _fields_ class variable must, however, be defined before the type is first used (an instance is
created, sizeof() is called on it, and so on). Later assignments to the _fields_ class variable
will raise an AttributeError.

Structure and union subclass constructors accept both positional and named arguments. Positional
arguments are used to initialize the fields in the same order as they appear in the _fields_ definition,
named arguments are used to initialize the fields with the corresponding name.
It is possible to define sub-subclasses of structure types, they inherit the fields of the base class plus the _fields_ defined in the sub-subclass, if any.

__pack__

An optional small integer that allows to override the alignment of structure fields in the instance. __pack__ must already be defined when _fields_ is assigned, otherwise it will have no effect.

__anonymous__

An optional sequence that lists the names of unnamed (anonymous) fields. __anonymous__ must be already defined when _fields_ is assigned, otherwise it will have no effect.

The fields listed in this variable must be structure or union type fields. ctypes will create descriptors in the structure type that allows to access the nested fields directly, without the need to create the structure or union field.

Here is an example type (Windows):

```python
class _U(Union):
    _fields_ = ["lptdesc", POINTER(TYPEDESC)],
                "lpadesc", POINTER(ARRAYDESC)),
                "hreftype", HREFTYPE)

class TYPEDESC(Structure):
    _anonymous_ = ("u",)
    _fields_ = ["u", _U],
                "vt", VARTYPE]
```

The TYPEDESC structure describes a COM data type, the vt field specifies which one of the union fields is valid. Since the u field is defined as anonymous field, it is now possible to access the members directly off the TYPEDESC instance. td.lptdesc and td.u.lptdesc are equivalent, but the former is faster since it does not need to create a temporary union instance:

```python
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

It is possible to define sub-subclasses of structures, they inherit the fields of the base class. If the subclass definition has a separate _fields_ variable, the fields specified in this are appended to the fields of the base class.

Structure and union constructors accept both positional and keyword arguments. Positional arguments are used to initialize member fields in the same order as they are appear in _fields_. Keyword arguments in the constructor are interpreted as attribute assignments, so they will initialize _fields_ with the same name, or create new attributes for names not present in _fields_.

**Arrays and pointers**

Not yet written - please see the sections *Pointers* and section *Arrays* in the tutorial.
The modules described in this chapter provide support for concurrent execution of code. The appropriate choice of tool will depend on the task to be executed (CPU bound vs IO bound) and preferred style of development (event driven cooperative multitasking vs preemptive multitasking). Here’s an overview:

17.1 threading — Thread-based parallelism

Source code: Lib/threading.py

This module constructs higher-level threading interfaces on top of the lower level _thread module. See also the queue module.

The dummy_threading module is provided for situations where threading cannot be used because _thread is missing.

Note: While they are not listed below, the camelCase names used for some methods and functions in this module in the Python 2.x series are still supported by this module.

This module defines the following functions:

```python
def active_count():
    # Return the number of Thread objects currently alive. The returned count is equal to the length of the list returned by enumerate().

def current_thread():
    # Return the current Thread object, corresponding to the caller’s thread of control. If the caller’s thread of control was not created through the threading module, a dummy thread object with limited functionality is returned.

def get_ident():
    # Return the ‘thread identifier’ of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created. New in version 3.3.

def enumerate():
    # Return a list of all Thread objects currently alive. The list includes daemonic threads, dummy thread objects created by current_thread(), and the main thread. It excludes terminated threads and threads that have not yet been started.

def settrace(func):
    # Set a trace function for all threads started from the threading module. The func will be passed to sys.settrace() for each thread, before its run() method is called.

def setprofile(func):
    # Set a profile function for all threads started from the threading module. The func will be passed to sys.setprofile() for each thread, before its run() method is called.
```
threading.stack_size([size])

Return the thread stack size used when creating new threads. The optional size argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32 KiB). If changing the thread stack size is unsupported, a RuntimeError is raised. If the specified stack size is invalid, a ValueError is raised and the stack size is unmodified. 32 KiB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32 KiB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4 KiB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information). Availability: Windows, systems with POSIX threads.

This module also defines the following constant:

threading.TIMEOUT_MAX

The maximum value allowed for the timeout parameter of blocking functions (Lock.acquire(), RLock.acquire(), Condition.wait(), etc.). Specifying a timeout greater than this value will raise an OverflowError. New in version 3.2.

This module defines a number of classes, which are detailed in the sections below.

The design of this module is loosely based on Java’s threading model. However, where Java makes locks and condition variables basic behavior of every object, they are separate objects in Python. Python’s Thread class supports a subset of the behavior of Java’s Thread class; currently, there are no priorities, no thread groups, and threads cannot be destroyed, stopped, suspended, resumed, or interrupted. The static methods of Java’s Thread class, when implemented, are mapped to module-level functions.

All of the methods described below are executed atomically.

17.1.1 Thread-Local Data

Thread-local data is data whose values are thread specific. To manage thread-local data, just create an instance of local (or a subclass) and store attributes on it:

```python
mydata = threading.local()
mydata.x = 1
```

The instance’s values will be different for separate threads.

```python
class threading.local
    A class that represents thread-local data.

    For more details and extensive examples, see the documentation string of the _threading_local module.
```

17.1.2 Thread Objects

The Thread class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the run() method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, only override the __init__() and run() methods of this class.

Once a thread object is created, its activity must be started by calling the thread’s start() method. This invokes the run() method in a separate thread of control.

Once the thread’s activity is started, the thread is considered ‘alive’. It stops being alive when its run() method terminates – either normally, or by raising an unhandled exception. The is_alive() method tests whether the thread is alive.

Other threads can call a thread’s join() method. This blocks the calling thread until the thread whose join() method is called is terminated.

A thread has a name. The name can be passed to the constructor, and read or changed through the name attribute.
A thread can be flagged as a “daemon thread”. The significance of this flag is that the entire Python program exits when only daemon threads are left. The initial value is inherited from the creating thread. The flag can be set through the `daemon` property or the `daemon` constructor argument.

Note: Daemon threads are abruptly stopped at shutdown. Their resources (such as open files, database transactions, etc.) may not be released properly. If you want your threads to stop gracefully, make them non-daemonic and use a suitable signalling mechanism such as an `Event`.

There is a “main thread” object; this corresponds to the initial thread of control in the Python program. It is not a daemon thread.

There is the possibility that “dummy thread objects” are created. These are thread objects corresponding to “alien threads”, which are threads of control started outside the threading module, such as directly from C code. Dummy thread objects have limited functionality; they are always considered alive and daemonic, and cannot be `join()`ed. They are never deleted, since it is impossible to detect the termination of alien threads.

```python
class threading.Thread(
        group=None, target=None, name=None, args=(), kwvars={}, *, daemon=None)
```

This constructor should always be called with keyword arguments. Arguments are:

- `group` should be `None`; reserved for future extension when a `ThreadGroup` class is implemented.
- `target` is the callable object to be invoked by the `run()` method. Defaults to `None`, meaning nothing is called.
- `name` is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number.
- `args` is the argument tuple for the target invocation. Defaults to `()`.
- `kwvars` is a dictionary of keyword arguments for the target invocation. Defaults to `{}`.

If not `None`, `daemon` explicitly sets whether the thread is daemonic. If `None` (the default), the daemonic property is inherited from the current thread.

If the subclass overrides the constructor, it must make sure to invoke the base class constructor (`Thread.__init__()`)) before doing anything else to the thread. Changed in version 3.3: Added the `daemon` argument.

```start()
```

Start the thread’s activity.

It must be called at most once per thread object. It arranges for the object’s `run()` method to be invoked in a separate thread of control.

This method will raise a `RuntimeError` if called more than once on the same thread object.

```run()
```

Method representing the thread’s activity.

You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object’s constructor as the `target` argument, if any, with sequential and keyword arguments taken from the `args` and `kwvars` arguments, respectively.

```join(timeout=None)
```

Wait until the thread terminates. This blocks the calling thread until the thread whose `join()` method is called terminates – either normally or through an unhandled exception –, or until the optional timeout occurs.

When the `timeout` argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As `join()` always returns `None`, you must call `is_alive()` after `join()` to decide whether a timeout happened – if the thread is still alive, the `join()` call timed out.

When the `timeout` argument is not present or `None`, the operation will block until the thread terminates. A thread can be `join()`ed many times.
**join()** raises a **RuntimeError** if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to **join()** a thread before it has been started and attempts to do so raise the same exception.

**name**
A string used for identification purposes only. It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

```python
getName()
setName()
```
Old getter/setter API for `name`; use it directly as a property instead.

**ident**
The ‘thread identifier’ of this thread or `None` if the thread has not been started. This is a nonzero integer. See the `_thread.get_ident()` function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

```python
is_alive()
```
Return whether the thread is alive.

This method returns `True` just before the `run()` method starts until just after the `run()` method terminates. The module function `enumerate()` returns a list of all alive threads.

**daemon**
A boolean value indicating whether this thread is a daemon thread (True) or not (False). This must be set before `start()` is called, otherwise **RuntimeError** is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to `daemon = False`.

The entire Python program exits when no alive non-daemon threads are left.

```python
isDaemon()
setDaemon()
```
Old getter/setter API for `daemon`; use it directly as a property instead.

**CPython implementation detail:** In CPython, due to the **Global Interpreter Lock**, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use `multiprocessing` or `concurrent.futures.ProcessPoolExecutor`. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

## 17.1.3 Lock Objects

A primitive lock is a synchronization primitive that is not owned by a particular thread when locked. In Python, it is currently the lowest level synchronization primitive available, implemented directly by the `_thread` extension module.

A primitive lock is in one of two states, “locked” or “unlocked”. It is created in the unlocked state. It has two basic methods, `acquire()` and `release()`. When the state is unlocked, `acquire()` changes the state to locked and returns immediately. When the state is locked, `acquire()` blocks until a call to `release()` in another thread changes it to unlocked, then the `acquire()` call resets it to locked and returns. The `release()` method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, a **RuntimeError** will be raised.

Locks also support the **context manager protocol**.

When more than one thread is blocked in `acquire()` waiting for the state to turn to unlocked, only one thread proceeds when a `release()` call resets the state to unlocked; which one of the waiting threads proceeds is not defined, and may vary across implementations.

All methods are executed atomically.

```python
class threading.Lock
```
The class implementing primitive lock objects. Once a thread has acquired a lock, subsequent attempts to
acquire it block, until it is released; any thread may release it. Changed in version 3.3: Changed from a factory function to a class.

\texttt{acquire}(\texttt{blocking=True, timeout=-1})

Acquire a lock, blocking or non-blocking.

When invoked with the \texttt{blocking} argument set to \texttt{True} (the default), block until the lock is unlocked, then set it to locked and return \texttt{True}.

When invoked with the \texttt{blocking} argument set to \texttt{False}, do not block. If a call with \texttt{blocking} set to \texttt{True} would block, return \texttt{False} immediately; otherwise, set the lock to locked and return \texttt{True}.

When invoked with the floating-point \texttt{timeout} argument set to a positive value, block for at most the number of seconds specified by \texttt{timeout} and as long as the lock cannot be acquired. A \texttt{timeout} argument of \texttt{-1} specifies an unbounded wait. It is forbidden to specify a \texttt{timeout} when \texttt{blocking} is false.

The return value is \texttt{True} if the lock is acquired successfully, \texttt{False} if not (for example if the \texttt{timeout} expired). Changed in version 3.2: The \texttt{timeout} parameter is new. Changed in version 3.2: Lock acquires can now be interrupted by signals on POSIX.

\texttt{release}()

Release a lock. This can be called from any thread, not only the thread which has acquired the lock.

When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

When invoked on an unlocked lock, a \texttt{RuntimeError} is raised.

There is no return value.

### 17.1.4 RLock Objects

A reentrant lock is a synchronization primitive that may be acquired multiple times by the same thread. Internally, it uses the concepts of “owning thread” and “recursion level” in addition to the locked/unlocked state used by primitive locks. In the locked state, some thread owns the lock; in the unlocked state, no thread owns it.

To lock the lock, a thread calls its \texttt{acquire()} method; this returns once the thread owns the lock. To unlock the lock, a thread calls its \texttt{release()} method. \texttt{acquire()}/\texttt{release()} call pairs may be nested; only the final \texttt{release()} (the \texttt{release()} of the outermost pair) resets the lock to unlocked and allows another thread blocked in \texttt{acquire()} to proceed.

Reentrant locks also support the context manager protocol.

\textbf{class} \texttt{threading.RLock}

This class implements reentrant lock objects. A reentrant lock must be released by the thread that acquired it. Once a thread has acquired a reentrant lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

Note that \texttt{RLock} is actually a factory function which returns an instance of the most efficient version of the concrete \texttt{RLock} class that is supported by the platform.

\texttt{acquire}(\texttt{blocking=True, timeout=-1})

Acquire a lock, blocking or non-blocking.

When invoked without arguments: if this thread already owns the lock, increment the recursion level by one, and return immediately. Otherwise, if another thread owns the lock, block until the lock is unlocked. Once the lock is unlocked (not owned by any thread), then grab ownership, set the recursion level to one, and return. If more than one thread is blocked waiting until the lock is unlocked, only one at a time will be able to grab ownership of the lock. There is no return value in this case.

When invoked with the \texttt{blocking} argument set to \texttt{true}, do the same thing as when called without arguments, and return true.
When invoked with the `blocking` argument set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

When invoked with the floating-point `timeout` argument set to a positive value, block for at most the number of seconds specified by `timeout` and as long as the lock cannot be acquired. Return true if the lock has been acquired, false if the timeout has elapsed. Changed in version 3.2: The `timeout` parameter is new.

```python
release()
```
Release a lock, decrementing the recursion level. If after the decrement it is zero, reset the lock to unlocked (not owned by any thread), and if any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling thread.

Only call this method when the calling thread owns the lock. A `RuntimeError` is raised if this method is called when the lock is unlocked.

There is no return value.

### 17.1.5 Condition Objects

A condition variable is always associated with some kind of lock; this can be passed in or one will be created by default. Passing one in is useful when several condition variables must share the same lock. The lock is part of the condition object: you don’t have to track it separately.

A condition variable obeys the context manager protocol: using the `with` statement acquires the associated lock for the duration of the enclosed block. The `acquire()` and `release()` methods also call the corresponding methods of the associated lock.

Other methods must be called with the associated lock held. The `wait()` method releases the lock, and then blocks until another thread awakens it by calling `notify()` or `notify_all()`. Once awakened, `wait()` re-acquires the lock and returns. It is also possible to specify a timeout.

The `notify()` method wakes up one of the threads waiting for the condition variable, if any are waiting. The `notify_all()` method wakes up all threads waiting for the condition variable.

Note: the `notify()` and `notify_all()` methods don’t release the lock; this means that the thread or threads awakened will not return from their `wait()` call immediately, but only when the thread that called `notify()` or `notify_all()` finally relinquishes ownership of the lock.

The typical programming style using condition variables uses the lock to synchronize access to some shared state; threads that are interested in a particular change of state call `wait()` repeatedly until they see the desired state, while threads that modify the state call `notify()` or `notify_all()` when they change the state in such a way that it could possibly be a desired state for one of the waiters. For example, the following code is a generic producer-consumer situation with unlimited buffer capacity:

```python
# Consume one item
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# Produce one item
with cv:
    make_an_item_available()
    cv.notify()
```

The `while` loop checking for the application’s condition is necessary because `wait()` can return after an arbitrary long time, and the condition which prompted the `notify()` call may no longer hold true. This is inherent to multi-threaded programming. The `wait_for()` method can be used to automate the condition checking, and eases the computation of timeouts.
# Consume an item

```python
with cv:
    cv.wait_for(an_item_is_available)
    get_an_available_item()
```

To choose between `notify()` and `notify_all()`, consider whether one state change can be interesting for only one or several waiting threads. E.g. in a typical producer-consumer situation, adding one item to the buffer only needs to wake up one consumer thread.

```python
class threading.Condition(lock=None)
```

This class implements condition variable objects. A condition variable allows one or more threads to wait until they are notified by another thread.

If the `lock` argument is given and not `None`, it must be a `Lock` or `RLock` object, and it is used as the underlying lock. Otherwise, a new `RLock` object is created and used as the underlying lock. Changed in version 3.3: changed from a factory function to a class.

```python
acquire(*args)
```

Acquire the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.

```python
release()
```

Release the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.

```python
wait(timeout=None)
```

Wait until notified or until a timeout occurs. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.

This method releases the underlying lock, and then blocks until it is awakened by a `notify()` or `notify_all()` call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.

When the `timeout` argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

When the underlying lock is an `RLock`, it is not released using its `release()` method, since this may not actually unlock the lock when it was acquired multiple times recursively. Instead, an internal interface of the `RLock` class is used, which really unlocks it even when it has been recursively acquired several times. Another internal interface is then used to restore the recursion level when the lock is reacquired.

The return value is `True` unless a given `timeout` expired, in which case it is `False`. Changed in version 3.2: Previously, the method always returned `None`.

```python
wait_for(predicate, timeout=None)
```

Wait until a condition evaluates to `True`. `predicate` should be a callable which result will be interpreted as a boolean value. A `timeout` may be provided giving the maximum time to wait.

This utility method may call `wait()` repeatedly until the predicate is satisfied, or until a timeout occurs. The return value is the last return value of the predicate and will evaluate to `False` if the method timed out.

Ignoring the timeout feature, calling this method is roughly equivalent to writing:

```python
while not predicate():
    cv.wait()
```

Therefore, the same rules apply as with `wait()`: The lock must be held when called and is re-acquired on return. The predicate is evaluated with the lock held. New in version 3.2.

```python
notify(n=1)
```

By default, wake up one thread waiting on this condition, if any. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.
This method wakes up at most \( n \) of the threads waiting for the condition variable; it is a no-op if no threads are waiting.

The current implementation wakes up exactly \( n \) threads, if at least \( n \) threads are waiting. However, it’s not safe to rely on this behavior. A future, optimized implementation may occasionally wake up more than \( n \) threads.

Note: an awakened thread does not actually return from its \( \text{wait()} \) call until it can reacquire the lock. Since \( \text{notify()} \) does not release the lock, its caller should.

\( \text{notify\_all()} \)

Wake up all threads waiting on this condition. This method acts like \( \text{notify()} \), but wakes up all waiting threads instead of one. If the calling thread has not acquired the lock when this method is called, a \( \text{RuntimeError} \) is raised.

### 17.1.6 Semaphore Objects

This is one of the oldest synchronization primitives in the history of computer science, invented by the early Dutch computer scientist Edsger W. Dijkstra (he used the names \( P() \) and \( V() \) instead of \( \text{acquire()} \) and \( \text{release()} \)).

A semaphore manages an internal counter which is decremented by each \( \text{acquire()} \) call and incremented by each \( \text{release()} \) call. The counter can never go below zero; when \( \text{acquire()} \) finds that it is zero, it blocks, waiting until some other thread calls \( \text{release()} \).

Semaphores also support the \( \text{context manager protocol} \).

#### class \texttt{threading.\_Semaphore}(value=1)

This class implements semaphore objects. A semaphore manages a counter representing the number of \( \text{release()} \) calls minus the number of \( \text{acquire()} \) calls, plus an initial value. The \( \text{acquire()} \) method blocks if necessary until it can return without making the counter negative. If not given, \( value \) defaults to 1.

The optional argument gives the initial \( value \) for the internal counter; it defaults to 1. If the \( value \) given is less than 0, \( \text{ValueError} \) is raised. Changed in version 3.3: changed from a factory function to a class.

\( \text{acquire}(\text{blocking}=\text{True}, \text{timeout}=\text{None}) \)

Acquire a semaphore.

When invoked without arguments: if the internal counter is larger than zero on entry, decrement it by one and return immediately. If it is zero on entry, block, waiting until some other thread has called \( \text{release()} \) to make it larger than zero. This is done with proper interlocking so that if multiple \( \text{acquire()} \) calls are blocked, \( \text{release()} \) will wake exactly one of them up. The implementation may pick one at random, so the order in which blocked threads are awakened should not be relied on. Returns true (or blocks indefinitely).

When invoked with \( \text{blocking} \) set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

When invoked with a \( \text{timeout} \) other than None, it will block for at most \( \text{timeout} \) seconds. If \( \text{acquire()} \) does not complete successfully in that interval, return false. Return true otherwise. Changed in version 3.2: The \( \text{timeout} \) parameter is new.

\( \text{release()} \)

Release a semaphore, incrementing the internal counter by one. When it was zero on entry and another thread is waiting for it to become larger than zero again, wake up that thread.

#### class \texttt{threading.\_BoundedSemaphore}(value=1)

Class implementing bounded semaphore objects. A bounded semaphore checks to make sure its current value doesn’t exceed its initial value. If it does, \( \text{ValueError} \) is raised. In most situations semaphores are used to guard resources with limited capacity. If the semaphore is released too many times it’s a sign of a bug. If not given, \( value \) defaults to 1. Changed in version 3.3: changed from a factory function to a class.
Semaphore Example

Semaphores are often used to guard resources with limited capacity, for example, a database server. In any situation where the size of the resource is fixed, you should use a bounded semaphore. Before spawning any worker threads, your main thread would initialize the semaphore:

```python
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
```

Once spawned, worker threads call the semaphore’s acquire and release methods when they need to connect to the server:

```python
with pool_sema:
    conn = connectdb()
    try:
        # ... use connection ...
    finally:
        conn.close()
```

The use of a bounded semaphore reduces the chance that a programming error which causes the semaphore to be released more than it’s acquired will go undetected.

17.1.7 Event Objects

This is one of the simplest mechanisms for communication between threads: one thread signals an event and other threads wait for it.

An event object manages an internal flag that can be set to true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true.

```python
class threading.Event
    Class implementing event objects. An event manages a flag that can be set to true with the set() method and reset to false with the clear() method. The wait() method blocks until the flag is true. The flag is initially false. Changed in version 3.3: changed from a factory function to a class.

    is_set()
        Return true if and only if the internal flag is true.

    set()
        Set the internal flag to true. All threads waiting for it to become true are awakened. Threads that call wait() once the flag is true will not block at all.

    clear()
        Reset the internal flag to false. Subsequently, threads calling wait() will block until set() is called to set the internal flag to true again.

    wait(timeout=None)
        Block until the internal flag is true. If the internal flag is true on entry, return immediately. Otherwise, block until another thread calls set() to set the flag to true, or until the optional timeout occurs.

        When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

        This method returns true if and only if the internal flag has been set to true, either before the wait call or after the wait starts, so it will always return True except if a timeout is given and the operation times out. Changed in version 3.1: Previously, the method always returned None.
```

17.1.8 Timer Objects

This class represents an action that should be run only after a certain amount of time has passed — a timer. Timer is a subclass of Thread and as such also functions as an example of creating custom threads.
Timers are started, as with threads, by calling their `start()` method. The timer can be stopped (before its action has begun) by calling the `cancel()` method. The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user.

For example:

```python
def hello():
    print("hello, world")

# create a timer that will run hello() after 30 seconds
# t is the timer object
T = Timer(30.0, hello)
T.start() # after 30 seconds, "hello, world" will be printed
```

class threading.Timer(interval, function, args=None, kwargs=None)

Create a timer that will run `function` with arguments `args` and keyword arguments `kwargs`, after `interval` seconds have passed. If `args` is None (the default) then an empty list will be used. If `kwargs` is None (the default) then an empty dict will be used. Changed in version 3.3: changed from a factory function to a class.

```python
cancel()
```

Stop the timer, and cancel the execution of the timer’s action. This will only work if the timer is still in its waiting stage.

### 17.1.9 Barrier Objects

New in version 3.2. This class provides a simple synchronization primitive for use by a fixed number of threads that need to wait for each other. Each of the threads tries to pass the barrier by calling the `wait()` method and will block until all of the threads have made the call. At this points, the threads are released simultaneously.

The barrier can be reused any number of times for the same number of threads.

As an example, here is a simple way to synchronize a client and server thread:

```python
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
        process_server_connection(connection)

def client():
    b.wait()
    while True:
        connection = make_connection()
        process_client_connection(connection)
```

class threading.Barrier(parties, action=None, timeout=None)

Create a barrier object for `parties` number of threads. An `action`, when provided, is a callable to be called by one of the threads when they are released. `timeout` is the default timeout value if none is specified for the `wait()` method.

```python
wait(timeout=None)
```

Pass the barrier. When all the threads party to the barrier have called this function, they are all released simultaneously. If a `timeout` is provided, it is used in preference to any that was supplied to the class constructor.

The return value is an integer in the range 0 to `parties` – 1, different for each thread. This can be used to select a thread to do some special housekeeping, e.g.:

```python
i = barrier.wait()
if i == 0:
```
# Only one thread needs to print this
print("passed the barrier")

If an action was provided to the constructor, one of the threads will have called it prior to being released. Should this call raise an error, the barrier is put into the broken state.

If the call times out, the barrier is put into the broken state.

This method may raise a BrokenBarrierError exception if the barrier is broken or reset while a thread is waiting.

reset()
Return the barrier to the default, empty state. Any threads waiting on it will receive the BrokenBarrierError exception.

Note that using this function may require some external synchronization if there are other threads whose state is unknown. If a barrier is broken it may be better to just leave it and create a new one.

abort()
Put the barrier into a broken state. This causes any active or future calls to wait() to fail with the BrokenBarrierError. Use this for example if one of the needs to abort, to avoid deadlocking the application.

It may be preferable to simply create the barrier with a sensible timeout value to automatically guard against one of the threads going awry.

parties
The number of threads required to pass the barrier.

n_waiting
The number of threads currently waiting in the barrier.

broken
A boolean that is True if the barrier is in the broken state.

exception threading.BrokenBarrierError
This exception, a subclass of RuntimeError, is raised when the Barrier object is reset or broken.

17.1.10 Using locks, conditions, and semaphores in the with statement

All of the objects provided by this module that have acquire() and release() methods can be used as context managers for a with statement. The acquire() method will be called when the block is entered, and release() will be called when the block is exited. Hence, the following snippet:

with some_lock:
    # do something...

is equivalent to:

some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()

Currently, Lock, RLock, Condition, Semaphore, and BoundedSemaphore objects may be used as with statement context managers.
17.2 multiprocessing — Process-based parallelism

17.2.1 Introduction

`multiprocessing` is a package that supports spawning processes using an API similar to the `threading` module. The `multiprocessing` package offers both local and remote concurrency, effectively side-stepping the `Global Interpreter Lock` by using subprocesses instead of threads. Due to this, the `multiprocessing` module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows.

Note: Some of this package’s functionality requires a functioning shared semaphore implementation on the host operating system. Without one, the `multiprocessing.synchronize` module will be disabled, and attempts to import it will result in an `ImportError`. See issue 3770 for additional information.

Note: Functionality within this package requires that the `__main__` module be importable by the children. This is covered in Programming guidelines however it is worth pointing out here. This means that some examples, such as the `multiprocessing.pool.Pool` examples will not work in the interactive interpreter. For example:

```python
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
... >>> p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
  File "", line 1, in <listcomp>
AttributeError: 'module' object has no attribute 'f'
```

(If you try this it will actually output three full tracebacks interleaved in a semi-random fashion, and then you may have to stop the master process somehow.)

The Process class

In `multiprocessing`, processes are spawned by creating a `Process` object and then calling its `start()` method. `Process` follows the API of `threading.Thread`. A trivial example of a multiprocess program is

```python
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

To show the individual process IDs involved, here is an expanded example:

```python
from multiprocessing import Process
import os

def info(title):
    print(title)
```
print('module name:', __name__)
if hasattr(os, 'getppid'):  # only available on Unix
    print('parent process:', os.getppid())
print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
p = Process(target=f, args=('bob',))
p.start()
p.join()

For an explanation of why (on Windows) the if __name__ == '__main__' part is necessary, see Programming guidelines.

Exchanging objects between processes

multiprocessing supports two types of communication channel between processes:

Queues

The Queue class is a near clone of queue.Queue. For example:

```python
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
p = Process(target=f, args=(q,))
p.start()
print(q.get())  # prints "[42, None, 'hello']"
p.join()
```

Queues are thread and process safe.

Pipes

The Pipe() function returns a pair of connection objects connected by a pipe which by default is duplex (two-way). For example:

```python
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
p = Process(target=f, args=(child_conn,))
p.start()
print(parent_conn.recv())  # prints "[42, None, 'hello']"
p.join()
```

The two connection objects returned by Pipe() represent the two ends of the pipe. Each connection object has send() and recv() methods (among others). Note that data in a pipe may become corrupted if two processes (or threads) try to read from or write to the same end of the pipe at the same time.
same time. Of course there is no risk of corruption from processes using different ends of the pipe at the same time.

Synchronization between processes

`multiprocessing` contains equivalents of all the synchronization primitives from `threading`. For instance one can use a lock to ensure that only one process prints to standard output at a time:

```python
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    print('hello world', i)
    l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()

Without using the lock output from the different processes is liable to get all mixed up.

Sharing state between processes

As mentioned above, when doing concurrent programming it is usually best to avoid using shared state as far as possible. This is particularly true when using multiple processes.

However, if you really do need to use some shared data then `multiprocessing` provides a couple of ways of doing so.

Shared memory

Data can be stored in a shared memory map using `Value` or `Array`. For example, the following code

```python
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

will print

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```
The ‘d’ and ‘i’ arguments used when creating `num` and `arr` are typecodes of the kind used by
the `array` module: ‘d’ indicates a double precision float and ‘i’ indicates a signed integer. These
shared objects will be process and thread-safe.

For more flexibility in using shared memory one can use the `multiprocessing.sharedctypes` module which supports the creation of arbitrary `ctypes` objects allocated from shared memory.

Server process

A manager object returned by `Manager()` controls a server process which holds Python objects and
allows other processes to manipulate them using proxies.

A manager returned by `Manager()` will support types `list`, `dict`, `Namespace`, `Lock`, `RLock`,
`Semaphore`, `BoundedSemaphore`, `Condition`, `Event`, `Barrier`, `Queue`, `Value` and
`Array`. For example,

```python
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))

        p = Process(target=f, args=(d, l))
        p.start()
        p.join()

    print(d)
    print(l)
```

will print

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Server process managers are more flexible than using shared memory objects because they can be
made to support arbitrary object types. Also, a single manager can be shared by processes on different
computers over a network. They are, however, slower than using shared memory.

Using a pool of workers

The `Pool` class represents a pool of worker processes. It has methods which allows tasks to be offloaded to the
worker processes in a few different ways.

For example:

```python
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(processes=4) as pool:
        result = pool.apply_async(f, [10])
```

Server process managers are more flexible than using shared memory objects because they can be
made to support arbitrary object types. Also, a single manager can be shared by processes on different
computers over a network. They are, however, slower than using shared memory.

Using a pool of workers

The `Pool` class represents a pool of worker processes. It has methods which allows tasks to be offloaded to the
worker processes in a few different ways.

For example:

```python
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(processes=4) as pool:
        result = pool.apply_async(f, [10])
```
Note that the methods of a pool should only ever be used by the process which created it.

### 17.2.2 Reference

The `multiprocessing` package mostly replicates the API of the `threading` module.

#### Process and exceptions

The `Process` class has equivalents of all the methods of `threading.Thread`.

The constructor should always be called with keyword arguments. `group` should always be `None`; it exists solely for compatibility with `threading.Thread`. `target` is the callable object to be invoked by the `run()` method. It defaults to `None`, meaning nothing is called. `name` is the process name (see `name` for more details). `args` is the argument tuple for the target invocation. `kwargs` is a dictionary of keyword arguments for the target invocation. If provided, the keyword-only `daemon` argument sets the process `daemon` flag to `True` or `False`. If `None` (the default), this flag will be inherited from the creating process.

By default, no arguments are passed to `target`.

If a subclass overrides the constructor, it must make sure it invokes the base class constructor `(Process.__init__())` before doing anything else to the process. Changed in version 3.3: Added the `daemon` argument.

- **run()**
  Method representing the process’s activity.

  You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the `args` and `kwargs` arguments, respectively.

- **start()**
  Start the process’s activity.

  This must be called at most once per process object. It arranges for the object’s `run()` method to be invoked in a separate process.

- **join([timeout])**
  If the optional argument `timeout` is `None` (the default), the method blocks until the process whose `join()` method is called terminates. If `timeout` is a positive number, it blocks at most `timeout` seconds.

  A process can be joined many times.

  A process cannot join itself because this would cause a deadlock. It is an error to attempt to join a process before it has been started.

- **name**
  The process’s name. The name is a string used for identification purposes only. It has no semantics. Multiple processes may be given the same name.

  The initial name is set by the constructor. If no explicit name is provided to the constructor, a name of the form ‘Process-N1:N2:...:Nk’ is constructed, where each Ni is the N-th child of its parent.

- **is_alive()**
  Return whether the process is alive.

  Roughly, a process object is alive from the moment the `start()` method returns until the child process terminates.
daemon

The process’s daemon flag, a Boolean value. This must be set before start() is called.

The initial value is inherited from the creating process.

When a process exits, it attempts to terminate all of its daemonic child processes.

Note that a daemonic process is not allowed to create child processes. Otherwise a daemonic process
would leave its children orphaned if it gets terminated when its parent process exits. Additionally,
these are not Unix daemons or services, they are normal processes that will be terminated (and not
joined) if non-daemonic processes have exited.

In addition to the threading.Thread API, Process objects also support the following attributes and
methods:

pid

Return the process ID. Before the process is spawned, this will be None.

exitcode

The child’s exit code. This will be None if the process has not yet terminated. A negative value -N
indicates that the child was terminated by signal N.

authkey

The process’s authentication key (a byte string).

When multiprocessing is initialized the main process is assigned a random string using
os.urandom().

When a Process object is created, it will inherit the authentication key of its parent process, although
this may be changed by setting authkey to another byte string.

See Authentication keys.

sentinel

A numeric handle of a system object which will become “ready” when the process ends.

You can use this value if you want to wait on several events at once using
multiprocessing.connection.wait(). Otherwise calling join() is simpler.

On Windows, this is an OS handle usable with the WaitForSingleObject and
WaitForMultipleObjects family of API calls. On Unix, this is a file descriptor usable with
primitives from the select module. New in version 3.3.

terminate()

Terminate the process. On Unix this is done using the SIGTERM signal; on Windows
TerminateProcess() is used. Note that exit handlers and finally clauses, etc., will not be ex-
ecuted.

Note that descendant processes of the process will not be terminated – they will simply become or-
phaned.

Warning: If this method is used when the associated process is using a pipe or queue then the
pipe or queue is liable to become corrupted and may become unusable by other process. Similarly,
if the process has acquired a lock or semaphore etc. then terminating it is liable to cause other
processes to deadlock.

Note that the start(), join(), is_alive(), terminate() and exitcode methods should only
be called by the process that created the process object.

Example usage of some of the methods of Process:

```python
>>> import multiprocessing, time, signal
>>> p = multiprocessing.Process(target=time.sleep, args=(1000,))
>>> print(p, p.is_alive())
<Process(Process-1, initial)> False
>>> p.start()
```

---

17.2. multiprocessing — Process-based parallelism 599
>>> print(p, p.is_alive())
<Process(Process-1, started)> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print(p, p.is_alive())
<Process(Process-1, stopped[SIGTERM])> False
>>> p.exitcode == -signal.SIGTERM
True

exception multiprocessing.ProcessError
The base class of all multiprocessing exceptions.

exception multiprocessing.BufferTooShort
Exception raised by Connection.recv_bytes_into() when the supplied buffer object is too small for the message read.

  If e is an instance of BufferTooShort then e.args[0] will give the message as a byte string.

exception multiprocessing.AuthenticationError
Raised when there is an authentication error.

exception multiprocessing.TimeoutError
Raised by methods with a timeout when the timeout expires.

Pipes and Queues

When using multiple processes, one generally uses message passing for communication between processes and avoids having to use any synchronization primitives like locks.

For passing messages one can use Pipe() (for a connection between two processes) or a queue (which allows multiple producers and consumers).

The Queue, SimpleQueue and JoinableQueue types are multi-producer, multi-consumer FIFO queues modelled on the queue.Queue class in the standard library. They differ in that Queue lacks the task_done() and join() methods introduced into Python 2.5’s queue.Queue class.

If you use JoinableQueue then you must call JoinableQueue.task_done() for each task removed from the queue or else the semaphore used to count the number of unfinished tasks may eventually overflow, raising an exception.

Note that one can also create a shared queue by using a manager object – see Managers.

Note: multiprocessing uses the usual queue.Empty and queue.Full exceptions to signal a timeout. They are not available in the multiprocessing namespace so you need to import them from queue.

Note: When an object is put on a queue, the object is pickled and a background thread later flushes the pickled data to an underlying pipe. This has some consequences which are a little surprising, but should not cause any practical difficulties – if they really bother you then you can instead use a queue created with a manager.

  1. After putting an object on an empty queue there may be an infinitesimal delay before the queue’s empty() method returns False and get_nowait() can return without raising queue.Empty.
  2. If multiple processes are enqueuing objects, it is possible for the objects to be received at the other end out-of-order. However, objects enqueued by the same process will always be in the expected order with respect to each other.

Warning: If a process is killed using Process.terminate() or os.kill() while it is trying to use a Queue, then the data in the queue is likely to become corrupted. This may cause any other process to get an exception when it tries to use the queue later on.
Warning: As mentioned above, if a child process has put items on a queue (and it has not used \texttt{JoinableQueue.cancel\_join\_thread}), then that process will not terminate until all buffered items have been flushed to the pipe. This means that if you try joining that process you may get a deadlock unless you are sure that all items which have been put on the queue have been consumed. Similarly, if the child process is non-daemonic then the parent process may hang on exit when it tries to join all its non-daemonic children. Note that a queue created using a manager does not have this issue. See \textit{Programming guidelines}. 

For an example of the usage of queues for interprocess communication see \textit{Examples}. 

\texttt{multiprocessing.Pipe([\textit{duplex}])} 

Returns a pair \texttt{(conn1, conn2)} of \texttt{Connection} objects representing the ends of a pipe. 

If \texttt{duplex} is True (the default) then the pipe is bidirectional. If \texttt{duplex} is False then the pipe is unidirectional: \texttt{conn1} can only be used for receiving messages and \texttt{conn2} can only be used for sending messages. 

\texttt{class multiprocessing.Queue([\textit{maxsize}])} 

Returns a process shared queue implemented using a pipe and a few locks/semaphores. When a process first puts an item on the queue a feeder thread is started which transfers objects from a buffer into the pipe. 

The usual \texttt{queue.Empty} and \texttt{queue.Full} exceptions from the standard library’s \texttt{queue} module are raised to signal timeouts. 

\texttt{Queue} implements all the methods of \texttt{queue.Queue} except for \texttt{task\_done()} and \texttt{join()}. 

\texttt{qsize()} 

Return the approximate size of the queue. Because of multithreading/multiprocessing semantics, this number is not reliable. 

Note that this may raise \texttt{NotImplementedError} on Unix platforms like Mac OS X where \texttt{sem\_getvalue()} is not implemented. 

\texttt{empty()} 

Return True if the queue is empty, False otherwise. Because of multithreading/multiprocessing semantics, this is not reliable. 

\texttt{full()} 

Return True if the queue is full, False otherwise. Because of multithreading/multiprocessing semantics, this is not reliable. 

\texttt{put (obj[, block[, timeout]])} 

Put obj into the queue. If the optional argument \texttt{block} is True (the default) and \texttt{timeout} is None (the default), block if necessary until a free slot is available. If \texttt{timeout} is a positive number, it blocks at most \texttt{timeout} seconds and raises the \texttt{queue.Full} exception if no free slot was available within that time. Otherwise (\texttt{block} is False), put an item on the queue if a free slot is immediately available, else raise the \texttt{queue.Full} exception (\texttt{timeout} is ignored in that case). 

\texttt{put\_nowait (obj)} 

Equivalent to \texttt{put(obj, False)}. 

\texttt{get ([block[, timeout]])} 

Remove and return an item from the queue. If optional args \texttt{block} is True (the default) and \texttt{timeout} is None (the default), block if necessary until an item is available. If \texttt{timeout} is a positive number, it blocks at most \texttt{timeout} seconds and raises the \texttt{queue.Empty} exception if no item was available within that time. Otherwise (\texttt{block} is False), return an item if one is immediately available, else raise the \texttt{queue.Empty} exception (\texttt{timeout} is ignored in that case). 

\texttt{get\_nowait ()} 

Equivalent to \texttt{get(False)}. 

\texttt{multiprocessing.Queue} has a few additional methods not found in \texttt{queue.Queue}. These methods are usually unnecessary for most code: 

\texttt{close()} 

Indicate that no more data will be put on this queue by the current process. The background thread
will quit once it has flushed all buffered data to the pipe. This is called automatically when the queue is garbage collected.

\texttt{join\_thread()}

Join the background thread. This can only be used after \texttt{close()} has been called. It blocks until the background thread exits, ensuring that all data in the buffer has been flushed to the pipe.

By default if a process is not the creator of the queue then on exit it will attempt to join the queue’s background thread. The process can call \texttt{cancel\_join\_thread()} to make \texttt{join\_thread()} do nothing.

\texttt{cancel\_join\_thread()}

Prevent \texttt{join\_thread()} from blocking. In particular, this prevents the background thread from being joined automatically when the process exits – see \texttt{join\_thread()}.

A better name for this method might be \texttt{allow\_exit\_without\_flush()}. It is likely to cause enqueued data to be lost, and you almost certainly will not need to use it. It is really only there if you need the current process to exit immediately without waiting to flush enqueued data to the underlying pipe, and you don’t care about lost data.

\textbf{class} \texttt{multiprocessing\.SimpleQueue}

It is a simplified \texttt{Queue} type, very close to a locked \texttt{Pipe}.

\texttt{empty()}

Return \texttt{True} if the queue is empty, \texttt{False} otherwise.

\texttt{get()}

Remove and return an item from the queue.

\texttt{put(item)}

Put \texttt{item} into the queue.

\textbf{class} \texttt{multiprocessing\.JoinableQueue([maxsize])}

\texttt{JoinableQueue}, a \texttt{Queue} subclass, is a queue which additionally has \texttt{task\_done()} and \texttt{join()} methods.

\texttt{task\_done()}

Indicate that a formerly enqueued task is complete. Used by queue consumers. For each \texttt{get()} used to fetch a task, a subsequent call to \texttt{task\_done()} tells the queue that the processing on the task is complete.

If a \texttt{join()} is currently blocking, it will resume when all items have been processed (meaning that a \texttt{task\_done()} call was received for every item that had been \texttt{put()} into the queue).

Raises a \texttt{ValueError} if called more times than there were items placed in the queue.

\texttt{join()}

Block until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer calls \texttt{task\_done()} to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, \texttt{join()} unblocks.

\textbf{Miscellaneous}

\texttt{multiprocessing\.active\_children()}

Return list of all live children of the current process.

Calling this has the side affect of “joining” any processes which have already finished.

\texttt{multiprocessing\.cpu\_count()}

Return the number of CPUs in the system. May raise \texttt{NotImplementedError}.

\texttt{multiprocessing\.current\_process()}

Return the \texttt{Process} object corresponding to the current process.
An analogue of `threading.current_thread()`.

`multiprocessing.freeze_support()`
Add support for when a program which uses `multiprocessing` has been frozen to produce a Windows executable. (Has been tested with `py2exe`, `PyInstaller` and `cx_Freeze`.)

One needs to call this function straight after the `if __name__ == '__main__'` line of the main module. For example:

```python
from multiprocessing import Process, freeze_support

def f():
    print('hello world!')

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

If the `freeze_support()` line is omitted then trying to run the frozen executable will raise `RuntimeError`.

If the module is being run normally by the Python interpreter then `freeze_support()` has no effect.

`multiprocessing.set_executable()`
Sets the path of the Python interpreter to use when starting a child process. (By default `sys.executable` is used). Embedders will probably need to do something like

```python
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

before they can create child processes. (Windows only)

---

**Note:** `multiprocessing` contains no analogues of `threading.active_count()`, `threading.enumerate()`, `threading.settrace()`, `threading.setprofile()`, `threading.Timer`, or `threading.local`.

---

### Connection Objects

Connection objects allow the sending and receiving of picklable objects or strings. They can be thought of as message oriented connected sockets.

Connection objects are usually created using `Pipe()` – see also *Listeners and Clients*.

**class** `multiprocessing.Connection`

**send**(obj)
Send an object to the other end of the connection which should be read using `recv()`.

The object must be picklable. Very large pickles (approximately 32 MB+, though it depends on the OS) may raise a `ValueError` exception.

**recv**( )
Return an object sent from the other end of the connection using `send()`. Blocks until there is something to receive. Raises `EOFError` if there is nothing left to receive and the other end was closed.

**fileno**( )
Return the file descriptor or handle used by the connection.

**close**( )
Close the connection.
This is called automatically when the connection is garbage collected.

**poll**([`timeout `])

Return whether there is any data available to be read.

If `timeout` is not specified then it will return immediately. If `timeout` is a number then this specifies the maximum time in seconds to block. If `timeout` is `None` then an infinite timeout is used.

Note that multiple connection objects may be polled at once by using `multiprocessing.connection.wait()`.

**send_bytes**([`buffer`, `offset`, `size `])

Send byte data from a `bytes-like object` as a complete message.

If `offset` is given then data is read from that position in `buffer`. If `size` is given then that many bytes will be read from buffer. Very large buffers (approximately 32 MB+, though it depends on the OS) may raise a `ValueError` exception.

**recv_bytes**([`maxlength `])

Return a complete message of byte data sent from the other end of the connection as a string. Blocks until there is something to receive. Raises `EOFError` if there is nothing left to receive and the other end has closed.

If `maxlength` is specified and the message is longer than `maxlength` then `OSError` is raised and the connection will no longer be readable. Changed in version 3.3: This function used to raise a `IOError`, which is now an alias of `OSError`.

**recv_bytes_into**([`buffer`, `offset `])

Read into `buffer` a complete message of byte data sent from the other end of the connection and return the number of bytes in the message. Blocks until there is something to receive. Raises `EOFError` if there is nothing left to receive and the other end was closed.

`buffer` must be a writable `bytes-like object`. If `offset` is given then the message will be written into the buffer from that position. Offset must be a non-negative integer less than the length of `buffer` (in bytes).

If the buffer is too short then a `BufferTooShort` exception is raised and the complete message is available as `e.args[0]` where `e` is the exception instance.

Changed in version 3.3: Connection objects themselves can now be transferred between processes using `Connection.send()` and `Connection.recv()`. New in version 3.3: Connection objects now support the context manager protocol – see `Context Manager Types`. **__enter__()** returns the connection object, and **__exit__()** calls `close()`.

For example:

```python
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes(b'thank you')
>>> a.recv_bytes()
'b\'thank you\'

>>> import array
>>> arr1 = array.array('i', range(5))
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```
Warning: The `Connection.recv()` method automatically unpickles the data it receives, which can be a security risk unless you can trust the process which sent the message. Therefore, unless the connection object was produced using `Pipe()` you should only use the `recv()` and `send()` methods after performing some sort of authentication. See `Authentication keys`.

Warning: If a process is killed while it is trying to read or write to a pipe then the data in the pipe is likely to become corrupted, because it may become impossible to be sure where the message boundaries lie.

**Synchronization primitives**

Generally synchronization primitives are not as necessary in a multiprocess program as they are in a multithreaded program. See the documentation for `threading` module.

Note that one can also create synchronization primitives by using a manager object – see `Managers`.

```python
class multiprocessing.Barrier(parties[, action[, timeout]])

class multiprocessing.BoundedSemaphore([value])
    (On Mac OS X, this is indistinguishable from `Semaphore` because `sem_getvalue()` is not implemented on that platform).

class multiprocessing.Condition([lock])
    If `lock` is specified then it should be a `Lock` or `RLock` object from `multiprocessing`. Changed in version 3.3: The `wait_for()` method was added.

class multiprocessing.Event
    A clone of `threading.Event`.

class multiprocessing.Lock
    A non-recursive lock object: a clone of `threading.Lock`.

class multiprocessing.RLock
    A recursive lock object: a clone of `threading.RLock`.

class multiprocessing.Semaphore([value])
    A semaphore object: a clone of `threading.Semaphore`.
```

Note: The `acquire()` and `wait()` methods of each of these types treat negative timeouts as zero timeouts. This differs from `threading` where, since version 3.2, the equivalent `acquire()` methods treat negative timeouts as infinite timeouts.

On Mac OS X, `sem_timedwait` is unsupported, so calling `acquire()` with a timeout will emulate that function’s behavior using a sleeping loop.

Note: If the SIGINT signal generated by Ctrl-C arrives while the main thread is blocked by a call to `BoundedSemaphore.acquire()`, `Lock.acquire()`, `RLock.acquire()`, `Semaphore.acquire()`, `Condition.acquire()` or `Condition.wait()` then the call will be immediately interrupted and `KeyboardInterrupt` will be raised. This differs from the behaviour of `threading` where SIGINT will be ignored while the equivalent blocking calls are in progress.
Shared ctypes Objects

It is possible to create shared objects using shared memory which can be inherited by child processes.

```python
multiprocessing.Value(typecode_or_type, *args, lock=True)
```

Return a ctypes object allocated from shared memory. By default the return value is actually a synchronized wrapper for the object. The object itself can be accessed via the value attribute of a Value.

typecode_or_type determines the type of the returned object: it is either a ctypes type or a one character typecode of the kind used by the array module. *args is passed on to the constructor for the type.

If lock is True (the default) then a new lock object is created to synchronize access to the value. If lock is a Lock or RLock object then that will be used to synchronize access to the value. If lock is False then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Note that lock is a keyword-only argument.

```python
multiprocessing:Array(typecode_or_type, size_or_initializer, *, lock=True)
```

Return a ctypes array allocated from shared memory. By default the return value is actually a synchronized wrapper for the array.

typecode_or_type determines the type of the elements of the returned array: it is either a ctypes type or a one character typecode of the kind used by the array module. If size_or_initializer is an integer, then it determines the length of the array, and the array will be initially zeroed. Otherwise, size_or_initializer is a sequence which is used to initialize the array and whose length determines the length of the array.

If lock is True (the default) then a new lock object is created to synchronize access to the value. If lock is a Lock or RLock object then that will be used to synchronize access to the value. If lock is False then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Note that lock is a keyword-only argument.

Note that an array of ctypes.c_char has value and raw attributes which allow one to use it to store and retrieve strings.

The multiprocessing.sharedctypes module

The multiprocessing.sharedctypes module provides functions for allocating ctypes objects from shared memory which can be inherited by child processes.

Note: Although it is possible to store a pointer in shared memory remember that this will refer to a location in the address space of a specific process. However, the pointer is quite likely to be invalid in the context of a second process and trying to dereference the pointer from the second process may cause a crash.

```python
multiprocessing.sharedctypes.RawArray(typecode_or_type, size_or_initializer)
```

Return a ctypes array allocated from shared memory.

typecode_or_type determines the type of the elements of the returned array: it is either a ctypes type or a one character typecode of the kind used by the array module. If size_or_initializer is an integer then it determines the length of the array, and the array will be initially zeroed. Otherwise size_or_initializer is a sequence which is used to initialize the array and whose length determines the length of the array.

Note that setting and getting an element is potentially non-atomic – use Array() instead to make sure that access is automatically synchronized using a lock.

```python
multiprocessing.sharedctypes.RawValue(typecode_or_type, *args)
```

Return a ctypes object allocated from shared memory.

typecode_or_type determines the type of the returned object: it is either a ctypes type or a one character typecode of the kind used by the array module. *args is passed on to the constructor for the type.
Note that setting and getting the value is potentially non-atomic – use Value() instead to make sure that access is automatically synchronized using a lock.

Note that an array of ctypes.c_char has value and raw attributes which allow one to use it to store and retrieve strings – see documentation for ctypes.

multiprocessing.sharedctypes.Array (typecode_or_type, size_or_initializer, *, lock=True)
The same as RawArray() except that depending on the value of lock a process-safe synchronization wrapper may be returned instead of a raw ctypes array.

If lock is True (the default) then a new lock object is created to synchronize access to the value. If lock is a Lock or RLock object then that will be used to synchronize access to the value. If lock is False then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Note that lock is a keyword-only argument.

multiprocessing.sharedctypes.Value (typecode_or_type, *args, lock=True)
The same as RawValue() except that depending on the value of lock a process-safe synchronization wrapper may be returned instead of a raw ctypes object.

If lock is True (the default) then a new lock object is created to synchronize access to the value. If lock is a Lock or RLock object then that will be used to synchronize access to the value. If lock is False then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Note that lock is a keyword-only argument.

multiprocessing.sharedctypes.copy (obj)
Return a ctypes object allocated from shared memory which is a copy of the ctypes object obj.

multiprocessing.sharedctypes.synchronized (obj[.lock])
Return a process-safe wrapper object for a ctypes object which uses lock to synchronize access. If lock is None (the default) then a multiprocessing.RLock object is created automatically.

A synchronized wrapper will have two methods in addition to those of the object it wraps: get_obj() returns the wrapped object and get_lock() returns the lock object used for synchronization.

Note that accessing the ctypes object through the wrapper can be a lot slower than accessing the raw ctypes object.

The table below compares the syntax for creating shared ctypes objects from shared memory with the normal ctypes syntax. (In the table MyStruct is some subclass of ctypes.Structure.)

<table>
<thead>
<tr>
<th>ctypes</th>
<th>sharedctypes using type</th>
<th>sharedctypes using typecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_double(2.4)</td>
<td>RawValue(c_double, 2.4)</td>
<td>RawValue('d', 2.4)</td>
</tr>
<tr>
<td>MyStruct(4, 6)</td>
<td>RawValue(MyStruct, 4, 6)</td>
<td>RawArray('h', 7)</td>
</tr>
<tr>
<td>(c_short * 7)</td>
<td>RawArray(c_short, 7)</td>
<td>RawArray('i', 9, 2, 8)</td>
</tr>
<tr>
<td>(c_int * 3)</td>
<td>RawArray(c_int, 9, 2, 8)</td>
<td></td>
</tr>
</tbody>
</table>

Below is an example where a number of ctypes objects are modified by a child process:

```python
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
        a.y **= 2
```

17.2. multiprocessing — Process-based parallelism

607
```python
if __name__ == '__main__':
    lock = Lock()
    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', b'hello world', lock=lock)
    A = Array(Point, [(1.875,-6.25), (-5.75,2.0), (2.375,9.5)], lock=lock)
    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()
    print(n.value)
    print(x.value)
    print(s.value)
    print([(a.x, a.y) for a in A])
```

The results printed are:

```
49
0.1111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
```

Managers

Managers provide a way to create data which can be shared between different processes, including sharing over a network between processes running on different machines. A manager object controls a server process which manages shared objects. Other processes can access the shared objects by using proxies.

```python
multiprocessing.Manager()

Returns a started SyncManager object which can be used for sharing objects between processes. The returned manager object corresponds to a spawned child process and has methods which will create shared objects and return corresponding proxies.
```

Manager processes will be shutdown as soon as they are garbage collected or their parent process exits. The manager classes are defined in the `multiprocessing.managers` module:

```python
class multiprocessing.managers.BaseManager([address, authkey])

Create a BaseManager object.
```

Once created one should call `start()` or `get_server().serve_forever()` to ensure that the manager object refers to a started manager process.

`address` is the address on which the manager process listens for new connections. If `address` is `None` then an arbitrary one is chosen.

`authkey` is the authentication key which will be used to check the validity of incoming connections to the server process. If `authkey` is `None` then `current_process().authkey` is used. Otherwise `authkey` is used and it must be a byte string.

```python
start([initializer[, initargs]])
```

Start a subprocess to start the manager. If `initializer` is not `None` then the subprocess will call `initializer(*initargs)` when it starts.

```python
get_server()
```

Returns a Server object which represents the actual server under the control of the Manager. The Server object supports the `serve_forever()` method:

```python
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=('__', 50000), authkey=b'abc')
```
>>> server = manager.get_server()
>>> server.serve_forever()

Server additionally has an `address` attribute.

`connect()`  
Connect a local manager object to a remote manager process:

```python
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 5000), authkey=b'abc')
>>> m.connect()
```

`shutdown()`  
Stop the process used by the manager. This is only available if `start()` has been used to start the server process.

This can be called multiple times.

`register(typeid[, callable[, proxytype[, exposed[, method_to_typeid[, create_method]]]]]])`
A classmethod which can be used for registering a type or callable with the manager class.

- `typeid` is a “type identifier” which is used to identify a particular type of shared object. This must be a string.
- `callable` is a callable used for creating objects for this type identifier. If a manager instance will be connected to the server using the `connect()` method, or if the `create_method` argument is `False` then this can be left as `None`.
- `proxytype` is a subclass of `BaseProxy` which is used to create proxies for shared objects with this `typeid`. If `None` then a proxy class is created automatically.
- `exposed` is used to specify a sequence of method names which proxies for this typeid should be allowed to access using `BaseProxy._callMethod()`. (If `exposed` is `None` then `proxytype._exposed_` is used instead if it exists.) In the case where no exposed list is specified, all “public methods” of the shared object will be accessible. (Here a “public method” means any attribute which has a `__call__()` method and whose name does not begin with `_`).
- `method_to_typeid` is a mapping used to specify the return type of those exposed methods which should return a proxy. It maps method names to typeid strings. (If `method_to_typeid` is `None` then `proxytype._method_to_typeid_` is used instead if it exists.) If a method’s name is not a key of this mapping or if the mapping is `None` then the object returned by the method will be copied by value.
- `create_method` determines whether a method should be created with name `typeid` which can be used to tell the server process to create a new shared object and return a proxy for it. By default it is `True`.

`BaseManager` instances also have one read-only property:

`address`
The address used by the manager.

Changed in version 3.3: Manager objects support the context manager protocol – see `Context Manager Types`. `__enter__()` starts the server process (if it has not already started) and then returns the manager object. `__exit__()` calls `shutdown()`. In previous versions `__enter__()` did not start the manager’s server process if it was not already started.

`class multiprocessing.managers.SyncManager`  
A subclass of `BaseManager` which can be used for the synchronization of processes. Objects of this type are returned by `multiprocessing.Manager()`.

It also supports creation of shared lists and dictionaries.

`Barrier(parties[, action[, timeout]])`
Create a shared `threading.Barrier` object and return a proxy for it. New in version 3.3.
BoundedSemaphore([value])
Create a shared threading.BoundedSemaphore object and return a proxy for it.

Condition([lock])
Create a shared threading.Condition object and return a proxy for it.

If lock is supplied then it should be a proxy for a threading.Lock or threading.RLock object.
Changed in version 3.3: The wait_for() method was added.

Event()
Create a shared threading.Event object and return a proxy for it.

Lock()
Create a shared threading.Lock object and return a proxy for it.

Namespace()
Create a shared Namespace object and return a proxy for it.

Queue([maxsize])
Create a shared queue.Queue object and return a proxy for it.

RLock()
Create a shared threading.RLock object and return a proxy for it.

Semaphore([value])
Create a shared threading.Semaphore object and return a proxy for it.

Array(typecode, sequence)
Create an array and return a proxy for it.

Value(typecode, value)
Create an object with a writable value attribute and return a proxy for it.

dict()
dict(mapping)
dict(sequence)
Create a shared dict object and return a proxy for it.

list()
list(sequence)
Create a shared list object and return a proxy for it.

Note: Modifications to mutable values or items in dict and list proxies will not be propagated through the manager, because the proxy has no way of knowing when its values or items are modified. To modify such an item, you can re-assign the modified object to the container proxy:

```python
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
d = lproxy[0]
d[‘a’] = 1
d[‘b’] = 2
# at this point, the changes to d are not yet synced, but by
# reassigning the dictionary, the proxy is notified of the change
lproxy[0] = d
```

Namespace objects

A namespace object has no public methods, but does have writable attributes. Its representation shows the values of its attributes.
However, when using a proxy for a namespace object, an attribute beginning with ‘_’ will be an attribute of the proxy and not an attribute of the referent:

```python
>>> manager = multiprocessing.Manager()
>>> Global = manager.Namespace()
>>> Global.x = 10
>>> Global.y = 'hello'
>>> Global._z = 12.3  # this is an attribute of the proxy
>>> print(Global)
Namespace(x=10, y='hello')
```

**Customized managers**

To create one’s own manager, one creates a subclass of `BaseManager` and uses the `register()` classmethod to register new types or callables with the manager class. For example:

```python
from multiprocessing.managers import BaseManager

class MathsClass:
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
    pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    with MyManager() as manager:
        maths = manager.Maths()
        print(maths.add(4, 3))  # prints 7
        print(maths.mul(7, 8))  # prints 56
```

**Using a remote manager**

It is possible to run a manager server on one machine and have clients use it from other machines (assuming that the firewalls involved allow it).

Running the following commands creates a server for a single shared queue which remote clients can access:

```python
>>> from multiprocessing.managers import BaseManager
>>> import queue

queue = queue.Queue()

class QueueManager(BaseManager): pass
QueueManager.register('get_queue', callable=lambda:queue)
m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
s = m.get_server()
s.serve_forever()
```

One client can access the server as follows:

```python
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
m.connect()
```
import multiprocessing

queue = m.get_queue()
queue.put('hello')

Another client can also use it:

from multiprocessing.managers import BaseManager

class QueueManager(BaseManager): pass

QueueManager.register('get_queue')

m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
m.connect()
queue = m.get_queue()
queue.get()  # 'hello'

Local processes can also access that queue, using the code from above on the client to access it remotely:

import multiprocessing

class Worker(Process):
    def __init__(self, q):
        self.q = q
    def run(self):
        self.q.put('local hello')

queue = Queue()
w = Worker(queue)
w.start()

class QueueManager(BaseManager): pass

QueueManager.register('get_queue', callable=lambda: queue)
m = QueueManager(address=('', 50000), authkey=b'abracadabra')
s = m.get_server()
s.serve_forever()

Proxy Objects

A proxy is an object which refers to a shared object which lives (presumably) in a different process. The shared object is said to be the referent of the proxy. Multiple proxy objects may have the same referent.

A proxy object has methods which invoke corresponding methods of its referent (although not every method of the referent will necessarily be available through the proxy). A proxy can usually be used in most of the same ways that its referent can:

manager = Manager()
l = manager.list([i*i for i in range(10)])
print(l)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
print(repr(l))<ListProxy object, typeid 'list' at 0x...>
l[4]
16
l[2:5][4, 9, 16]

Notice that applying str() to a proxy will return the representation of the referent, whereas applying repr() will return the representation of the proxy.

An important feature of proxy objects is that they are pickleable so they can be passed between processes. Note, however, that if a proxy is sent to the corresponding manager’s process then unpickling it will produce the referent...
itself. This means, for example, that one shared object can contain a second:

```python
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)  # referent of a now contains referent of b
[['']]
>>> b.append('hello')
>>> print(a, b)
[['hello']] ['hello']
```

Note: The proxy types in `multiprocessing` do nothing to support comparisons by value. So, for instance, we have:

```python
>>> manager.list([1,2,3]) == [1,2,3]
False
```

One should just use a copy of the referent instead when making comparisons.

class `multiprocessing.managers.BaseProxy`
Proxy objects are instances of subclasses of `BaseProxy`.

_**callmethod**(methodname[, args[, kwds]])
Call and return the result of a method of the proxy’s referent.

If `proxy` is a proxy whose referent is `obj` then the expression

```
proxy._callmethod(methodname, args, kwds)
```

will evaluate the expression

```
getattr(obj, methodname)(*args, **kwds)
```

in the manager’s process.

The returned value will be a copy of the result of the call or a proxy to a new shared object – see documentation for the `method_to_typeid` argument of `BaseManager.register()`.

If an exception is raised by the call, then it is re-raised by `_callmethod()`. If some other exception is raised in the manager’s process then this is converted into a `RemoteError` exception and is raised by `_callmethod()`.

Note in particular that an exception will be raised if `methodname` has not been exposed

An example of the usage of `_callmethod()`:

```python
>>> l = manager.list(range(10))
>>> l._callmethod('___len___')
10
>>> l._callmethod('___getslice___', (2, 7))  # equiv to 'l[2:7]'
[2, 3, 4, 5, 6]
>>> l._callmethod('___getitem___', (20,))  # equiv to 'l[20]'  
Traceback (most recent call last):
...  
IndexError: list index out of range
```

__**getvalue**__()

Return a copy of the referent.

If the referent is unpicklable then this will raise an exception.

__**repr**__()

Return a representation of the proxy object.
The Python Library Reference, Release 3.3.3

__str__()

Return the representation of the referent.

Cleanup

A proxy object uses a weakref callback so that when it gets garbage collected it deregisters itself from the manager which owns its referent.

A shared object gets deleted from the manager process when there are no longer any proxies referring to it.

Process Pools

One can create a pool of processes which will carry out tasks submitted to it with the Pool class.

```python
class multiprocessing.pool.Pool([processes[, initializer[, initargs[, maxtasksperchild]]]])
```

A process pool object which controls a pool of worker processes to which jobs can be submitted. It supports asynchronous results with timeouts and callbacks and has a parallel map implementation.

processes is the number of worker processes to use. If processes is None then the number returned by cpu_count() is used. If initializer is not None then each worker process will call initializer(*initargs) when it starts.

Note that the methods of the pool object should only be called by the process which created the pool. New in version 3.2: maxtasksperchild is the number of tasks a worker process can complete before it will exit and be replaced with a fresh worker process, to enable unused resources to be freed. The default maxtasksperchild is None, which means worker processes will live as long as the pool.

Note: Worker processes within a Pool typically live for the complete duration of the Pool’s work queue. A frequent pattern found in other systems (such as Apache, mod_wsgi, etc) to free resources held by workers is to allow a worker within a pool to complete only a set amount of work before being cleaned up and a new process spawned to replace the old one. The maxtasksperchild argument to the Pool exposes this ability to the end user.

```python
apply(func[, args[, kwds]])
```

Call func with arguments args and keyword arguments kwds. It blocks until the result is ready. Given this blocks, apply_async() is better suited for performing work in parallel. Additionally, func is only executed in one of the workers of the pool.

```python
apply_async(func[, args[, kwds[, callback[, error_callback]]]])
```

A variant of the apply() method which returns a result object.

If callback is specified then it should be a callable which accepts a single argument. When the result becomes ready callback is applied to it, that is unless the call failed, in which case the error_callback is applied instead.

If error_callback is specified then it should be a callable which accepts a single argument. If the target function fails, then the error_callback is called with the exception instance.

Callbacks should complete immediately since otherwise the thread which handles the results will get blocked.

```python
map(func, iterable[, chunksize])
```

A parallel equivalent of the map() built-in function (it supports only one iterable argument though). It blocks until the result is ready.

This method chops the iterable into a number of chunks which it submits to the process pool as separate tasks. The (approximate) size of these chunks can be specified by setting chunksize to a positive integer.

```python
map_async(func, iterable[, chunksize[, callback[, error_callback]]])
```

A variant of the map() method which returns a result object.
If `callback` is specified then it should be a callable which accepts a single argument. When the result becomes ready `callback` is applied to it, that is unless the call failed, in which case the `error_callback` is applied instead.

If `error_callback` is specified then it should be a callable which accepts a single argument. If the target function fails, then the `error_callback` is called with the exception instance.

Callbacks should complete immediately since otherwise the thread which handles the results will get blocked.

```python
imap (func, iterable[, chunksize ])
```

A lazier version of `map()`. The `chunksize` argument is the same as the one used by the `map()` method. For very long iterables using a large value for `chunksize` can make the job complete much faster than using the default value of 1.

Also if `chunksize` is 1 then the `next()` method of the iterator returned by the `imap()` method has an optional `timeout` parameter: `next(timeout)` will raise `multiprocessing.TimeoutError` if the result cannot be returned within `timeout` seconds.

```python
imap_unordered (func, iterable[, chunksize ])
```

The same as `imap()` except that the ordering of the results from the returned iterator should be considered arbitrary. (Only when there is only one worker process is the order guaranteed to be “correct”.)

```python
starmap (func, iterable[, chunksize ])
```

Like `map()` except that the elements of the `iterable` are expected to be iterables that are unpacked as arguments.

Hence an `iterable` of `[(1,2), (3, 4)]` results in `[func(1,2), func(3,4)]`. New in version 3.3.

```python
starmap_async (func, iterable[, chunksize[, callback[, error_back ]]]])
```

A combination of `starmap()` and `map_async()` that iterates over `iterable` of iterables and calls `func` with the iterables unpacked. Returns a result object. New in version 3.3.

```python
close ()
```

Prevents any more tasks from being submitted to the pool. Once all the tasks have been completed the worker processes will exit.

```python
terminate ()
```

Stops the worker processes immediately without completing outstanding work. When the pool object is garbage collected `terminate()` will be called immediately.

```python
join ()
```

Wait for the worker processes to exit. One must call `close()` or `terminate()` before using `join()`.

New in version 3.3: Pool objects now support the context manager protocol – see `Context Manager Types. __enter__()` returns the pool object, and __exit__()` calls `terminate()`.

```python
class multiprocessing.poolAsyncResult
```

The class of the result returned by `Pool.apply_async()` and `Pool.map_async()`.

```python
get ([timeout ])
```

Return the result when it arrives. If `timeout` is not `None` and the result does not arrive within `timeout` seconds then `multiprocessing.TimeoutError` is raised. If the remote call raised an exception then that exception will be reraised by `get()`.

```python
wait ([timeout ])
```

Wait until the result is available or until `timeout` seconds pass.

```python
ready ()
```

Return whether the call has completed.
successful()
Return whether the call completed without raising an exception. Will raise AssertionError if the result is not ready.

The following example demonstrates the use of a pool:

```python
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(processes=4) as pool:
        result = pool.apply_async(f, (10,))  # evaluate "f(10)" asynchronously
        print(result.get(timeout=1))  # prints "100" unless your computer is *very* slow

    print(pool.map(f, range(10)))  # prints "[0, 1, 4,..., 81]"

    it = pool.imap(f, range(10))
    print(next(it))  # prints "0"
    print(next(it))  # prints "1"
    print(it.next(timeout=1))  # prints "4" unless your computer is *very* slow

    import time
    result = pool.apply_async(time.sleep, (10,))
    print(result.get(timeout=1))  # raises TimeoutError
```

### Listeners and Clients

Usually message passing between processes is done using queues or by using Connection objects returned by Pipe().

However, the multiprocessing.connection module allows some extra flexibility. It basically gives a high level message oriented API for dealing with sockets or Windows named pipes. It also has support for digest authentication using the hmac module, and for polling multiple connections at the same time.

```python
multiprocessing.connection.deliver_challenge(connection, authkey)
    Send a randomly generated message to the other end of the connection and wait for a reply.
    If the reply matches the digest of the message using authkey as the key then a welcome message is sent to the other end of the connection. Otherwise AuthenticationError is raised.

multiprocessing.connection.answer_challenge(connection, authkey)
    Receive a message, calculate the digest of the message using authkey as the key, and then send the digest back.
    If a welcome message is not received, then AuthenticationError is raised.

multiprocessing.connection.Client(address[, family[, authenticate[, authkey]]]])
    Attempt to set up a connection to the listener which is using address address, returning a Connection.
    The type of the connection is determined by family argument, but this can generally be omitted since it can usually be inferred from the format of address. (See Address Formats)
    If authenticate is True or authkey is a byte string then digest authentication is used. The key used for authentication will be either authkey or current_process().authkey if authkey is None. If authentication fails then AuthenticationError is raised. See Authentication keys.

class multiprocessing.connection.Listener([address[, family[, backlog[, authenticate[, authkey]]]]])
    A wrapper for a bound socket or Windows named pipe which is ‘listening’ for connections.
    address is the address to be used by the bound socket or named pipe of the listener object.
```
Note: If an address of ‘0.0.0.0’ is used, the address will not be a connectable end point on Windows. If you require a connectable end-point, you should use ‘127.0.0.1’.

family is the type of socket (or named pipe) to use. This can be one of the strings ‘AF_INET’ (for a TCP socket), ‘AF_UNIX’ (for a Unix domain socket) or ‘AF_PIPE’ (for a Windows named pipe). Of these only the first is guaranteed to be available. If family is None then the family is inferred from the format of address. If address is also None then a default is chosen. This default is the family which is assumed to be the fastest available. See Address Formats. Note that if family is ‘AF_UNIX’ and address is None then the socket will be created in a private temporary directory created using tempfile.mkstemp().

If the listener object uses a socket then backlog (1 by default) is passed to the listen() method of the socket once it has been bound.

If authenticate is True (False by default) or authkey is not None then digest authentication is used.

If authkey is a byte string then it will be used as the authentication key; otherwise it must be None.

If authkey is None and authenticate is True then current_process().authkey is used as the authentication key. If authkey is None and authenticate is False then no authentication is done. If authentication fails then AuthenticationError is raised. See Authentication keys.

accept()  
Accept a connection on the bound socket or named pipe of the listener object and return a Connection object. If authentication is attempted and fails, then AuthenticationError is raised.

close()  
Close the bound socket or named pipe of the listener object. This is called automatically when the listener is garbage collected. However it is advisable to call it explicitly.

Listener objects have the following read-only properties:

address  
The address which is being used by the Listener object.

last_accepted  
The address from which the last accepted connection came. If this is unavailable then it is None.

New in version 3.3: Listener objects now support the context manager protocol – see Context Manager Types. __enter__() returns the listener object, and __exit__() calls close().

multiprocessing.connection.wait(object_list, timeout=None)  
Wait till an object in object_list is ready. Returns the list of those objects in object_list which are ready. If timeout is a float then the call blocks for at most that many seconds. If timeout is None then it will block for an unlimited period. A negative timeout is equivalent to a zero timeout.

For both Unix and Windows, an object can appear in object_list if it is

• a readable Connection object;

• a connected and readable socket.socket object; or

• the sentinel attribute of a Process object.

A connection or socket object is ready when there is data available to be read from it, or the other end has been closed.

Unix: wait(object_list, timeout) almost equivalent select.select(object_list, [], [], timeout). The difference is that, if select.select() is interrupted by a signal, it can raise OSError with an error number of EINTR, whereas wait() will not.

Windows: An item in object_list must either be an integer handle which is waitable (according to the definition used by the documentation of the Win32 function WaitForMultipleObjects()) or it can be an object with a fileno() method which returns a socket handle or pipe handle. (Note that pipe handles and socket handles are not waitable handles.) New in version 3.3.
Examples

The following server code creates a listener which uses ‘secret password’ as an authentication key. It then waits for a connection and sends some data to the client:

```python
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)  # family is deduced to be ‘AF_INET’

with Listener(address, authkey=b'secret password') as listener:
    with listener.accept() as conn:
        print('connection accepted from', listener.last_accepted)
        conn.send([2.25, None, 'junk', float])
        conn.send_bytes(b'hello')
        conn.send_bytes(array('i', [42, 1729]))
```

The following code connects to the server and receives some data from the server:

```python
from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)

with Client(address, authkey=b'secret password') as conn:
    print(conn.recv())  # => [2.25, None, 'junk', float]
    print(conn.recv_bytes())  # => 'hello'
    arr = array('i', [0, 0, 0, 0, 0])
    print(conn.recv_bytes_into(arr))  # => 8
    print(arr)  # => array('i', [42, 1729, 0, 0, 0])
```

The following code uses `wait()` to wait for messages from multiple processes at once:

```python
import time, random
from multiprocessing import Process, Pipe, current_process
from multiprocessing.connection import wait

def foo(w):
    for i in range(10):
        w.send((i, current_process().name))
    w.close()

if __name__ == '__main__':
    readers = []

    for i in range(4):
        r, w = Pipe(duplex=False)
        readers.append(r)
        p = Process(target=foo, args=(w,))
        p.start()
        # We close the writable end of the pipe now to be sure that
        # p is the only process which owns a handle for it. This
        # ensures that when p closes its handle for the writable end,
        # wait() will promptly report the readable end as being ready.
        w.close()
while readers:
    for r in wait(readers):
        try:
            msg = r.recv()
        except EOFError:
            readers.remove(r)
        else:
            print(msg)

Address Formats

- An ‘AF_INET’ address is a tuple of the form \((hostname, port)\) where \(hostname\) is a string and \(port\) is an integer.
- An ‘AF_UNIX’ address is a string representing a filename on the filesystem.
- An ‘AF_PIPE’ address is a string of the form \(\texttt{r'\\.\pipe\PipeName'}\). To use \(\texttt{Client()}\) to connect to a named pipe on a remote computer called \(ServerName\) one should use an address of the form \(\texttt{r'\\ServerName\pipe\PipeName'}\) instead.

Note that any string beginning with two backslashes is assumed by default to be an ‘AF_PIPE’ address rather than an ‘AF_UNIX’ address.

Authentication keys

When one uses \(\texttt{Connection.recv()}\), the data received is automatically unpickled. Unfortunately unpickling data from an untrusted source is a security risk. Therefore \(\texttt{Listener} and \(\texttt{Client()}\) use the \(\texttt{hmac}\) module to provide digest authentication.

An authentication key is a byte string which can be thought of as a password: once a connection is established both ends will demand proof that the other knows the authentication key. (Demonstrating that both ends are using the same key does \textbf{not} involve sending the key over the connection.)

If authentication is requested but no authentication key is specified then the return value of \(\texttt{current_process().authkey}\) is used (see \(\texttt{Process}\)). This value will automatically inherited by any \(\texttt{Process}\) object that the current process creates. This means that (by default) all processes of a multi-process program will share a single authentication key which can be used when setting up connections between themselves.

Suitable authentication keys can also be generated by using \(\texttt{os.urandom()}\).

Logging

Some support for logging is available. Note, however, that the \(\texttt{logging}\) package does not use process shared locks so it is possible (depending on the handler type) for messages from different processes to get mixed up.

\begin{verbatim}
multiprocessing.get_logger()  
Returns the logger used by \(\texttt{multiprocessing}\). If necessary, a new one will be created.

When first created the logger has level \(\texttt{logging.NOTSET}\) and no default handler. Messages sent to this logger will not by default propagate to the root logger.

Note that on Windows child processes will only inherit the level of the parent process’s logger – any other customization of the logger will not be inherited.

multiprocessing.log_to_stderr()  
This function performs a call to \(\texttt{get_logger()}\) but in addition to returning the logger created by \(\texttt{get_logger()}\), it adds a handler which sends output to \(\texttt{sys.stderr}\) using format \(\texttt{'[\%(levelname)s/%(processName)s] \%(message)s'}\).

Below is an example session with logging turned on:
\end{verbatim}
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed

>>> m = multiprocessing.Manager()
[INFO/SyncManager-... ] child process calling self.run()
[INFO/SyncManager-... ] created temp directory /.../pymp-...
[INFO/SyncManager-... ] manager serving at '/.../listener-...

>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-... ] manager exiting with exitcode 0

In addition to having these two logging functions, the multiprocessing also exposes two additional logging level attributes. These are SUBWARNING and SUBDEBUG. The table below illustrates where theses fit in the normal level hierarchy.

<table>
<thead>
<tr>
<th>Level</th>
<th>Numeric value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBWARNING</td>
<td>25</td>
</tr>
<tr>
<td>SUBDEBUG</td>
<td>5</td>
</tr>
</tbody>
</table>

For a full table of logging levels, see the logging module.

These additional logging levels are used primarily for certain debug messages within the multiprocessing module. Below is the same example as above, except with SUBDEBUG enabled:

>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(multiprocessing.SUBDEBUG)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed

>>> m = multiprocessing.Manager()
[INFO/SyncManager-... ] child process calling self.run()
[INFO/SyncManager-... ] created temp directory /.../pymp-...
[INFO/SyncManager-... ] manager serving at '/.../pymp-djGBXN/listener-...

>>> del m
[SUBDEBUG/MainProcess] finalizer calling ...
[INFO/MainProcess] sending shutdown message to manager
[DEBUG/SyncManager-... ] manager received shutdown message
[SUBDEBUG/SyncManager-... ] calling <Finalize object, callback=unlink, ...]
[SUBDEBUG/SyncManager-... ] finalizer calling <built-in function unlink> ...
[SUBDEBUG/SyncManager-... ] calling <Finalize object, dead>
[SUBDEBUG/SyncManager-... ] finalizer calling <function rmtree at 0x5aa730> ...
[INFO/SyncManager-... ] manager exiting with exitcode 0

The multiprocessing.dummy module

multiprocessing.dummy replicates the API of multiprocessing but is no more than a wrapper around the threading module.

17.2.3 Programming guidelines

There are certain guidelines and idioms which should be adhered to when using multiprocessing.

All platforms

Avoid shared state
As far as possible one should try to avoid shifting large amounts of data between processes.

It is probably best to stick to using queues or pipes for communication between processes rather than using the lower level synchronization primitives.

Picklability

Ensure that the arguments to the methods of proxies are picklable.

Thread safety of proxies

Do not use a proxy object from more than one thread unless you protect it with a lock.

(There is never a problem with different processes using the same proxy.)

Joining zombie processes

On Unix when a process finishes but has not been joined it becomes a zombie. There should never be very many because each time a new process starts (or active_children() is called) all completed processes which have not yet been joined will be joined. Also calling a finished process’s Process.is_alive will join the process. Even so it is probably good practice to explicitly join all the processes that you start.

Better to inherit than pickle/unpickle

On Windows many types from multiprocessing need to be picklable so that child processes can use them. However, one should generally avoid sending shared objects to other processes using pipes or queues. Instead you should arrange the program so that a process which needs access to a shared resource created elsewhere can inherit it from an ancestor process.

Avoid terminating processes

Using the Process.terminate method to stop a process is liable to cause any shared resources (such as locks, semaphores, pipes and queues) currently being used by the process to become broken or unavailable to other processes.

Therefore it is probably best to only consider using Process.terminate on processes which never use any shared resources.

Joining processes that use queues

Bear in mind that a process that has put items in a queue will wait before terminating until all the buffered items are fed by the “feeder” thread to the underlying pipe. (The child process can call the Queue.cancel_join_thread method of the queue to avoid this behaviour.)

This means that whenever you use a queue you need to make sure that all items which have been put on the queue will eventually be removed before the process is joined. Otherwise you cannot be sure that processes which have put items on the queue will terminate. Remember also that non-daemonic processes will be automatically be joined.

An example which will deadlock is the following:

```python
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()  # this deadlocks
    obj = queue.get()
```

A fix here would be to swap the last two lines round (or simply remove the p.join() line).

Explicitly pass resources to child processes
On Unix a child process can make use of a shared resource created in a parent process using a global resource. However, it is better to pass the object as an argument to the constructor for the child process.

Apart from making the code (potentially) compatible with Windows this also ensures that as long as the child process is still alive the object will not be garbage collected in the parent process. This might be important if some resource is freed when the object is garbage collected in the parent process.

So for instance

```python
from multiprocessing import Process, Lock
def f():
    ... do something using "lock" ...
if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

should be rewritten as

```python
from multiprocessing import Process, Lock
def f(l):
    ... do something using "l" ...
if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

Beware of replacing `sys.stdin` with a “file like object”

```python
import sys
os.close(sys.stdin.fileno())
in the multiprocessing.Process._bootstrap() method — this resulted in issues with processes-in-processes. This has been changed to:
sys.stdin.close()
sys.stdin = open(os.devnull)
```

Which solves the fundamental issue of processes colliding with each other resulting in a bad file descriptor error, but introduces a potential danger to applications which replace `sys.stdin()` with a “file-like object” with output buffering. This danger is that if multiple processes call `close()` on this file-like object, it could result in the same data being flushed to the object multiple times, resulting in corruption.

If you write a file-like object and implement your own caching, you can make it fork-safe by storing the pid whenever you append to the cache, and discarding the cache when the pid changes. For example:

```python
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

For more information, see issue 5155, issue 5313 and issue 5331
Windows

Since Windows lacks `os.fork()` it has a few extra restrictions:

More picklability

Ensure that all arguments to `Process.__init__()` are picklable. This means, in particular, that bound or unbound methods cannot be used directly as the `target` argument on Windows — just define a function and use that instead.

Also, if you subclass `Process` then make sure that instances will be picklable when the `Process.start` method is called.

Global variables

Bear in mind that if code run in a child process tries to access a global variable, then the value it sees (if any) may not be the same as the value in the parent process at the time that `Process.start` was called.

However, global variables which are just module level constants cause no problems.

Safe importing of main module

Make sure that the main module can be safely imported by a new Python interpreter without causing unintended side effects (such a starting a new process).

For example, under Windows running the following module would fail with a `RuntimeError`:

```python
from multiprocessing import Process

def foo():
    print('hello')

p = Process(target=foo)
p.start()
```

Instead one should protect the “entry point” of the program by using `if __name__ == '__main__':` as follows:

```python
from multiprocessing import Process, freeze_support

def foo():
    print('hello')

if __name__ == '__main__':
    freeze_support()
    p = Process(target=foo)
    p.start()
```

(The `freeze_support()` line can be omitted if the program will be run normally instead of frozen.)

This allows the newly spawned Python interpreter to safely import the module and then run the module’s `foo()` function.

Similar restrictions apply if a pool or manager is created in the main module.

17.2.4 Examples

Demonstration of how to create and use customized managers and proxies:

```python
# This module shows how to use arbitrary callables with a subclass of 'BaseManager'.
```

17.2. multiprocessing — Process-based parallelism 623
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

class Foo:
    def f(self):
        print('you called Foo.f()')
    def g(self):
        print('you called Foo.g()')
    def _h(self):
        print('you called Foo._h()')

# A simple generator function
def baz():
    for i in range(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ('next', '__next__')
    def __iter__(self):
        return self
    def __next__(self):
        return self._callmethod('next')
    def __next__(self):
        return self._callmethod('__next__')

# Function to return the operator module
def get_operator_module():
    return operator

class MyManager(BaseManager):
    pass

# register the Foo class; make 'f()' and 'g()' accessible via proxy
MyManager.register('Foo1', Foo)

# register the Foo class; make 'g()' and '_h()' accessible via proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

# register the generator function baz; use 'GeneratorProxy' to make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions accessible via proxy
MyManager.register('operator', get_operator_module())

# def test():
#     manager = MyManager()
manager.start()

print('-' * 20)

f1 = manager.Foo1()
f1.f()
f1.g()
assert not hasattr(f1, '_h')
assert sorted(f1._exposed_) == sorted(['f', 'g'])

print('-' * 20)

f2 = manager.Foo2()
f2.g()
f2._h()
assert not hasattr(f2, 'f')
assert sorted(f2._exposed_) == sorted(['g', '_h'])

print('-' * 20)

it = manager.baz()
for i in it:
    print('<%d>' % i, end=' ') print()

print('-' * 20)

op = manager.operator()
print('op.add(23, 45) =', op.add(23, 45))
print('op.pow(2, 94) =', op.pow(2, 94))
print('op.getslice(range(10), 2, 6) =', op.getslice(list(range(10)), 2, 6))
print('op.repeat(range(5), 3) =', op.repeat(list(range(5)), 3))
print('op._exposed_ =', op._exposed_)

##

if __name__ == '__main__':
    freeze_support()
test()

Using Pool:
#
# A test of 'multiprocessing.Pool' class
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#
import multiprocessing
import time
import random
import sys

#
# Functions used by test code
#

def calculate(func, args):
result = func(*args)
return '%%s says that %%s = %%s' % (
multiprocessing.current_process().name,
func.__name__, args, result)

def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5 * random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5 * random.random())
    return a + b

def f(x):
    return 1.0 / (x - 5.0)

def pow3(x):
    return x ** 3

def noop(x):
    pass

# # Test code #
#
def test():
    print('cpu_count() = %d

' % multiprocessing.cpu_count())

    # # Create pool #

    PROCESSES = 4
    print('Creating pool with %d processes

' % PROCESSES)
    pool = multiprocessing.Pool(PROCESSES)
    print('pool = %s

' % pool)
    print()

    # # Tests #

    TASKS = [(mul, (i, 7)) for i in range(10)] + 
          [(plus, (i, 8)) for i in range(10)]

    results = [pool.apply_async(calculate, t) for t in TASKS]
    imap_it = pool.imap(calculatestar, TASKS)
    imap_unordered_it = pool.imap_unordered(calculatestar, TASKS)

    print('Ordered results using pool.apply_async():

for r in results:
    print('	', r.get())

print()
**Ordered results using pool.imap():**

```python
for x in imap_it:
    print('  ', x)
print()
```

**Unordered results using pool.imap_unordered():**

```python
for x in imap_unordered_it:
    print('  ', x)
print()
```

**Ordered results using pool.map() --- will block till complete:**

```python
for x in pool.map(calculatestar, TASKS):
    print('  ', x)
print()
```

---

**Simple benchmarks**

```python
# Simple benchmarks
#

N = 100000
print('def pow3(x): return x**3')

t = time.time()
A = list(map(pow3, range(N)))
print('  map(pow3, range(%d)): %d seconds' % (N, time.time() - t))

t = time.time()
B = pool.map(pow3, range(N))
print('  pool.map(pow3, range(%d)): %d seconds' % (N, time.time() - t))

t = time.time()
C = list(pool.imap(pow3, range(N), chunksize=N//8))
print('  list(pool.imap(pow3, range(%d), chunksize=%d)): %d seconds' % (N, N//8, time.time() - t))

assert A == B == C, (len(A), len(B), len(C))
print()
```

```python
L = [None] * 1000000
print('def noop(x): pass')
print('L = [None] * 1000000')

t = time.time()
A = list(map(noop, L))
print('  map(noop, L): %d seconds' % (time.time() - t))

t = time.time()
B = pool.map(noop, L)
print('  pool.map(noop, L): %d seconds' % (time.time() - t))

t = time.time()
C = list(pool.imap(noop, L, chunksize=len(L)//8))
print('  list(pool.imap(noop, L, chunksize=%d)): %d seconds' % (time.time() - t))
```
assert A == B == C, (len(A), len(B), len(C))
print()

del A, B, C, L

# # Test error handling #
#
print('Testing error handling:

try:
    print(pool.apply(f, (5,)))
except ZeroDivisionError:
    print('	Got ZeroDivisionError as expected from pool.apply()')
else:
    raise AssertionError('expected ZeroDivisionError')

try:
    print(pool.map(f, list(range(10))))
except ZeroDivisionError:
    print('	Got ZeroDivisionError as expected from pool.map()')
else:
    raise AssertionError('expected ZeroDivisionError')

try:
    print(list(pool.imap(f, list(range(10)))))
except ZeroDivisionError:
    print('	Got ZeroDivisionError as expected from list(pool.imap())')
else:
    raise AssertionError('expected ZeroDivisionError')

it = pool.imap(f, list(range(10)))
for i in range(10):
    try:
        x = next(it)
    except ZeroDivisionError:
        if i == 5:
            pass
        except StopIteration:
            break
    else:
        if i == 5:
            raise AssertionError('expected ZeroDivisionError')

assert i == 9
print('	Got ZeroDivisionError as expected from IMapIterator.next()')
print()

# # Testing timeouts #
#
print('Testing ApplyResult.get() with timeout: ', end=' ')
res = pool.apply_async(calculate, TASKS[0])
while 1:
sys.stdout.flush()
try:
    sys.stdout.write('
\t%s' % res.get(0.02))
    break
except multiprocessing.TimeoutError:
    sys.stdout.write('.')
print()
print()

print('Testing IMapIterator.next() with timeout:', end=' ')
it = pool.imap(calculatestar, TASKS)
while 1:
    sys.stdout.flush()
    try:
        sys.stdout.write('
\t%s' % it.next(0.02))
    except StopIteration:
        break
    except multiprocessing.TimeoutError:
        sys.stdout.write('.')
    print()
    print()

# # Testing callback
#

print('Testing callback:')
A = []
B = [56, 0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
r = pool.apply_async(mul, (7, 8), callback=A.append)
r.wait()
r = pool.map_async(pow3, list(range(10)), callback=A.extend)
r.wait()
if A == B:
    print('	callbacks succeeded\n')
else:
    print('	*** callbacks failed\n\t\t\s% != %s
    % (A, B))

# # Check there are no outstanding tasks
#
assert not pool._cache, 'cache = %r' % pool._cache

# # Check close() methods
#
print('Testing close():')
for worker in pool._pool:
    assert worker.is_alive()

result = pool.apply_async(time.sleep, [0.5])
pool.close()
pool.join()

assert result.get() is None

for worker in pool._pool:
    assert not worker.is_alive()

print('close() succeeded

# # Check terminate() method
#

print('Testing terminate():

pool = multiprocessing.Pool(2)
DELTA = 0.1
ignore = pool.apply(pow3, [2])
results = [pool.apply_async(time.sleep, [DELTA]) for i in range(100)]
pool.terminate()
pool.join()

for worker in pool._pool:
    assert not worker.is_alive()

print('terminate() succeeded

# # Check garbage collection
#

print('Testing garbage collection:

pool = multiprocessing.Pool(2)
DELTA = 0.1
processes = pool._pool
ignore = pool.apply(pow3, [2])
results = [pool.apply_async(time.sleep, [DELTA]) for i in range(100)]
results = pool = None
time.sleep(DELTA * 2)

for worker in processes:
    assert not worker.is_alive()

print('garbage collection succeeded

if __name__ == '__main__':
multiprocessing.freeze_support()

assert len(sys.argv) in (1, 2)

if len(sys.argv) == 1 or sys.argv[1] == 'processes':
    print(' Using processes '.center(79, '-'))
elif sys.argv[1] == 'threads':
print(' Using threads '.center(79, '-'))
import multiprocessing.dummy as multiprocessing
else:
    print('Usage:
	%sm [processes | threads]
raise SystemExit(2)

Synchronization types like locks, conditions and queues:
#
# A test file for the 'multiprocessing' package
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#
import time
import sys
import random
from queue import Empty

import multiprocessing  # may get overwritten

#### TEST_VALUE

def value_func(running, mutex):
    random.seed()
    time.sleep(random.random()*4)
    mutex.acquire()
    print('
			' + str(multiprocessing.current_process()) + ' has finished')
    running.value -= 1
    mutex.release()

def test_value():
    TASKS = 10
    running = multiprocessing.Value('i', TASKS)
    mutex = multiprocessing.Lock()

    for i in range(TASKS):
        p = multiprocessing.Process(target=value_func, args=(running, mutex))
        p.start()

    while running.value > 0:
        time.sleep(0.08)
        mutex.acquire()
        print(running.value, end=' ')
        sys.stdout.flush()
        mutex.release()

    print()
    print('No more running processes')

#### TEST_QUEUE

def queue_func(queue):

17.2. multiprocessing — Process-based parallelism 631
for i in range(30):
    time.sleep(0.5 * random.random())
    queue.put(i*i)
queue.put('STOP')

def test_queue():
    q = multiprocessing.Queue()
    p = multiprocessing.Process(target=queue_func, args=(q,))
    p.start()
    o = None
    while o != 'STOP':
        try:
            o = q.get(timeout=0.3)
            print(o, end=' ')
            sys.stdout.flush()
        except Empty:
            print('TIMEOUT')
        print()

#### TEST_CONDITION

def condition_func(cond):
    cond.acquire()
    print('	' + str(cond))
    time.sleep(2)
    print('
child is notifying')
    print('	' + str(cond))
    cond.notify()
    cond.release()

def test_condition():
    cond = multiprocessing.Condition()
    p = multiprocessing.Process(target=condition_func, args=(cond,))
    print(cond)
    cond.acquire()
    print(cond)
    cond.acquire()
    print(cond)
    p.start()
    print('main is waiting')
    cond.wait()
    print('main has woken up')
    print(cond)
    cond.release()
    print(cond)
    cond.release()
    p.join()
    print(cond)

632 Chapter 17. Concurrent Execution
### TEST_SEMAPHORE

def semaphore_func(sema, mutex, running):
    sema.acquire()
    mutex.acquire()
    running.value += 1
    print(running.value, 'tasks are running')
    mutex.release()

    random.seed()
    time.sleep(random.random() * 2)

    mutex.acquire()
    running.value -= 1
    print('%s has finished' % multiprocessing.current_process())
    mutex.release()
    sema.release()

def test_semaphore():
    sema = multiprocessing.Semaphore(3)
    mutex = multiprocessing.RLock()
    running = multiprocessing.Value('i', 0)

    processes = [
        multiprocessing.Process(target=semaphore_func,
                                args=(sema, mutex, running))
            for i in range(10)
    ]

    for p in processes:
        p.start()

    for p in processes:
        p.join()

### TEST_JOIN_TIMEOUT

def join_timeout_func():
    print('	child sleeping')
    time.sleep(5.5)
    print('
	child terminating')

def test_join_timeout():
    p = multiprocessing.Process(target=join_timeout_func)
    p.start()

    print('waiting for process to finish')

    while 1:
        p.join(timeout=1)
        if not p.is_alive():
            break
        print('.', end=' ')
#### TEST_EVENT

def event_func(event):
    print(' %r is waiting' % multiprocessing.current_process())
    event.wait()
    print(' %r has woken up' % multiprocessing.current_process())

def test_event():
    event = multiprocessing.Event()

    processes = [multiprocessing.Process(target=event_func, args=(event,))
                 for i in range(5)]

    for p in processes:
        p.start()

    print('main is sleeping')
    time.sleep(2)

    print('main is setting event')
event.set()

    for p in processes:
        p.join()

#### TEST_SHAREDVALUES

def sharedvalues_func(values, arrays, shared_values, shared_arrays):
    for i in range(len(values)):
        v = values[i][1]
        sv = shared_values[i].value
        assert v == sv

    for i in range(len(values)):
        a = arrays[i][1]
        sa = list(shared_arrays[i][:])
        assert a == sa

    print('Tests passed')

def test_sharedvalues():
    values = [('i', 10),
              ('h', -2),
              ('d', 1.25)]

    arrays = [('i', list(range(100))),
              ('d', [0.25 * i for i in range(100)]),
              ('H', list(range(1000)))]

    shared_values = [multiprocessing.Value(id, v) for id, v in values]
    shared_arrays = [multiprocessing.Array(id, a) for id, a in arrays]
p = multiprocessing.Process(
    target=sharedvalues_func,
    args=(values, arrays, shared_values, shared_arrays)
)
p.start()
p.join()

assert p.exitcode == 0

####

def test(namespace=multiprocessing):
    global multiprocessing

    multiprocessing = namespace

    for func in [test_value, test_queue, test_condition,
                 test_semaphore, test_join_timeout, test_event,
                 test_sharedvalues]:

        print('

        ######## %s

        ' % func.__name__)
    func()

    ignore = multiprocessing.active_children()  # cleanup any old processes
    if hasattr(multiprocessing, '_debug_info'):
        info = multiprocessing._debug_info()
        if info:
            print(info)
            raise ValueError('there should be no positive refcounts left')

if __name__ == '__main__':
    multiprocessing.freeze_support()

    assert len(sys.argv) in (1, 2)

    if len(sys.argv) == 1 or sys.argv[1] == 'processes':
        print(' Using processes '.center(79, '-'))
        namespace = multiprocessing
    elif sys.argv[1] == 'manager':
        print(' Using processes and a manager '.center(79, '-'))
        namespace = multiprocessing.Manager()
        namespace.Process = multiprocessing.Process
        namespace.current_process = multiprocessing.current_process
        namespace.active_children = multiprocessing.active_children
    elif sys.argv[1] == 'threads':
        print(' Using threads '.center(79, '-'))
        import multiprocessing.dummy as namespace
    else:
        print('Usage:

        %s [processes | manager | threads]' % sys.argv[0])
        raise SystemExit(2)

    test(namespace)

An example showing how to use queues to feed tasks to a collection of worker processes and collect the results:

17.2. multiprocessing — Process-based parallelism
# Simple example which uses a pool of workers to carry out some tasks.
# Notice that the results will probably not come out of the output
# queue in the same order as the corresponding tasks were
# put on the input queue. If it is important to get the results back
# in the original order then consider using 'Pool.map()' or
# 'Pool.imap()' (which will save on the amount of code needed anyway).
#
# Copyright (c) 2006-2008, R Oudkerk
# All rights reserved.
#
import time
import random

from multiprocessing import Process, Queue, current_process, freeze_support

# Function run by worker processes
#
def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

# Function used to calculate result
#
def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' %
        (current_process().name, func.__name__, args, result)

# Functions referenced by tasks
#
def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

# functions to run

def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # Create queues

636 Chapter 17. Concurrent Execution
An example of how a pool of worker processes can each run a SimpleHTTPRequestHandler instance while sharing a single listening socket.

```python
import os
import sys
from multiprocessing import Process, current_process, freeze_support
from http.server import HTTPServer
from http.server import SimpleHTTPRequestHandler
if sys.platform == 'win32':
    import multiprocessing.reduction
    # make sockets pickable/inheritable

    # Example where a pool of http servers share a single listening socket
    # On Windows this module depends on the ability to pickle a socket
    # object so that the worker processes can inherit a copy of the server
    # object. (We import 'multiprocessing.reduction' to enable this pickling.)
    # Not sure if we should synchronize access to 'socket.accept()' method by
    # using a process-shared lock -- does not seem to be necessary.
    # Copyright (c) 2006-2008, R Oudkerk
    # All rights reserved.

    import os
    import sys

    from multiprocessing import Process, current_process, freeze_support
    from http.server import HTTPServer
    from http.server import SimpleHTTPRequestHandler

    if sys.platform == 'win32':
        import multiprocessing.reduction
        # make sockets pickable/inheritable

    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)

    # Start worker processes
    for i in range(NUMBER_OF_PROCESSES):
        Process(target=worker, args=(task_queue, done_queue)).start()

    # Get and print results
    print('Unordered results:"
    for i in range(len(TASKS1)):
        print('	', done_queue.get())

    # Add more tasks using 'put()' 
    for task in TASKS2:
        task_queue.put(task)

    # Get and print some more results
    for i in range(len(TASKS2)):
        print('	', done_queue.get())

    # Tell child processes to stop
    for i in range(NUMBER_OF_PROCESSES):
        task_queue.put('STOP')

if __name__ == '__main__':
    freeze_support()
test()
```
```python
def note(format, *args):
    sys.stderr.write('%s\t%s\n' % (current_process().name, format % args))

class RequestHandler(SimpleHTTPRequestHandler):
    # we override log_message() to show which process is handling the request
    def log_message(self, format, *args):
        note(format, *args)

def serve_forever(server):
    note('starting server')
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        pass

def runpool(address, number_of_processes):
    # create a single server object -- children will each inherit a copy
    server = HTTPServer(address, RequestHandler)

    # create child processes to act as workers
    for i in range(number_of_processes - 1):
        Process(target=serve_forever, args=(server,)).start()

    # main process also acts as a worker
    serve_forever(server)

def test():
    DIR = os.path.join(os.path.dirname(__file__), '..')
    ADDRESS = (‘localhost’, 8000)
    NUMBER_OF_PROCESSES = 4

    print(‘Serving at http://%s:%d using %d worker processes’ % 
          (ADDRESS[0], ADDRESS[1], NUMBER_OF_PROCESSES))
    print(‘To exit press Ctrl-‘ + [‘C’, ‘Break’][sys.platform==’win32’])

    os.chdir(DIR)
    runpool(ADDRESS, NUMBER_OF_PROCESSES)

if __name__ == '__main__':
    freeze_support()
    test()```
import threading
import queue
import gc

_timer = time.perf_counter
delta = 1

#### TEST_QUEUESPEED

def queuespeed_func(q, c, iterations):
a = '0' * 256
c.acquire()
c.notify()
c.release()

    for i in range(iterations):
        q.put(a)
        q.put('STOP')

def test_queuespeed(Process, q, c):
elapsed = 0
iterations = 1

    while elapsed < delta:
        iterations *= 2

        p = Process(target=queuespeed_func, args=(q, c, iterations))
c.acquire()
p.start()
c.wait()
c.release()

        result = None
        t = _timer()

        while result != 'STOP':
            result = q.get()
            elapsed = _timer() - t

        p.join()

    print(iterations, 'objects passed through the queue in', elapsed, 'seconds')
    print('average number/sec:', iterations/elapsed)

#### TEST_PIPESPEED

def pipe_func(c, cond, iterations):
a = '0' * 256
cond.acquire()
cond.notify()
cond.release()

    for i in range(iterations):
The Python Library Reference, Release 3.3.3

c.send(a)
c.send('STOP')

```python
def test_pipespeed():
c, d = multiprocessing.Pipe()
cond = multiprocessing.Condition()
elapsed = 0
iterations = 1

while elapsed < delta:
    iterations *= 2

    p = multiprocessing.Process(target=pipe_func,
                                args=(d, cond, iterations))
    cond.acquire()
p.start()
    cond.wait()
    cond.release()

    result = None
    t = _timer()

    while result != 'STOP':
        result = c.recv()

    elapsed = _timer() - t
    p.join()

    print(iterations, 'objects passed through connection in', elapsed, 'seconds')
    print('average number/sec:', iterations/elapsed)

### TEST_SEQSPEED

```python
def test_seqspeed(seq):
elapsed = 0
iterations = 1

while elapsed < delta:
    iterations *= 2

    t = _timer()

    for i in range(iterations):
        a = seq[5]

    elapsed = _timer() - t

    print(iterations, 'iterations in', elapsed, 'seconds')
    print('average number/sec:', iterations/elapsed)

### TEST_LOCK

```python
def test_lockspeed(l):
elapsed = 0
iterations = 1
while elapsed < delta:
    iterations *= 2
    t = _timer()
    for i in range(iterations):
        l.acquire()
        l.release()
    elapsed = _timer() - t
    print(iterations, ‘iterations in’, elapsed, ‘seconds’)
    print(‘average number/sec:’, iterations/elapsed)

#### TEST_CONDITION

def conditionspeed_func(c, N):
    c.acquire()
    c.notify()
    for i in range(N):
        c.wait()
        c.notify()
    c.release()

def test_conditionspeed(Process, c):
    elapsed = 0
    iterations = 1
    while elapsed < delta:
        iterations *= 2
        c.acquire()
        p = Process(target=conditionspeed_func, args=(c, iterations))
        p.start()
        c.wait()
        t = _timer()
        for i in range(iterations):
            c.notify()
            c.wait()
        elapsed = _timer() - t
        c.release()
        p.join()
        print(iterations * 2, ‘waits in’, elapsed, ‘seconds’)
        print(‘average number/sec:’, iterations * 2 / elapsed)

####
def test():

17.2. multiprocessing — Process-based parallelism
manager = multiprocessing.Manager()

gc.disable()

print('
\t
######## testing Queue.Queue
')
test_queuespeed(threading.Thread, queue.Queue(),
    threading.Condition())
print('
\t
######## testing multiprocessing.Queue
')
test_queuespeed(multiprocessing.Process, multiprocessing.Queue(),
    multiprocessing.Condition())
print('
\t
######## testing Queue managed by server process
')
test_queuespeed(multiprocessing.Process, manager.Queue(),
    manager.Condition())
print('
\t
######## testing multiprocessing.Pipe
')
test_pipespeed()

print()

print('
\t
######## testing list
')
test_seqspeed(list(range(10)))
print('
\t
######## testing list managed by server process
')
test_seqspeed(manager.list(list(range(10))))
print('
\t
######## testing Array("i", ..., lock=False)
')
test_seqspeed(multiprocessing.Array('i', list(range(10)), lock=False))
print('
\t
######## testing Array("i", ..., lock=True)
')
test_seqspeed(multiprocessing.Array('i', list(range(10)), lock=True))

print()

print('
\t
######## testing threading.Lock
')
test_lockspeed(threading.Lock())
print('
\t
######## testing threading.RLock
')
test_lockspeed(threading.RLock())
print('
\t
######## testing multiprocessing.Lock
')
test_lockspeed(multiprocessing.Lock())
print('
\t
######## testing multiprocessing.RLock
')
test_lockspeed(multiprocessing.RLock())
print('
\t
######## testing lock managed by server process
')
test_lockspeed(manager.Lock())
print('
\t
######## testing rlock managed by server process
')
test_lockspeed(manager.RLock())

print()

print('
\t
######## testing threading.Condition
')
test_conditionspeed(threading.Thread, threading.Condition())
print('
\t
######## testing multiprocessing.Condition
')
test_conditionspeed(multiprocessing.Process, multiprocessing.Condition())
print('
\t
######## testing condition managed by a server process
')
test_conditionspeed(multiprocessing.Process, manager.Condition())

gc.enable()

if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()
17.3 The concurrent package

Currently, there is only one module in this package:

- concurrent.futures – Launching parallel tasks

17.4 concurrent.futures — Launching parallel tasks

New in version 3.2. Source code: Lib/concurrent/futures/thread.py and Lib/concurrent/futures/process.py

The concurrent.futures module provides a high-level interface for asynchronously executing callables. The asynchronous execution can be performed with threads, using ThreadPoolExecutor, or separate processes, using ProcessPoolExecutor. Both implement the same interface, which is defined by the abstract Executor class.

17.4.1 Executor Objects

class concurrent.futures.Executor

An abstract class that provides methods to execute calls asynchronously. It should not be used directly, but through its concrete subclasses.

submit (fn, *args, **kwargs)

Schedules the callable, fn, to be executed as fn(*args **kwargs) and returns a Future object representing the execution of the callable.

with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())

map (func, *iterables, timeout=None)

Equivalent to map(func, *iterables) except func is executed asynchronously and several calls to func may be made concurrently. The returned iterator raises a TimeoutError if __next__() is called and the result isn’t available after timeout seconds from the original call to Executor.map(). timeout can be an int or a float. If timeout is not specified or None, there is no limit to the wait time. If a call raises an exception, then that exception will be raised when its value is retrieved from the iterator.

shutdown (wait=True)

Signal the executor that it should free any resources that it is using when the currently pending futures are done executing. Calls to Executor.submit() and Executor.map() made after shutdown will raise RuntimeError.

If wait is True then this method will not return until all the pending futures are done executing and the resources associated with the executor have been freed. If wait is False then this method will return immediately and the resources associated with the executor will be freed when all pending futures are done executing. Regardless of the value of wait, the entire Python program will not exit until all pending futures are done executing.

You can avoid having to call this method explicitly if you use the with statement, which will shutdown the Executor (waiting as if Executor.shutdown() were called with wait set to True):

import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest4.txt')
17.4.2 ThreadPoolExecutor

ThreadPoolExecutor is a Executor subclass that uses a pool of threads to execute calls asynchronously.

Deadlocks can occur when the callable associated with a Future waits on the results of another Future. For example:

```python
import time

def wait_on_b():
    time.sleep(5)
    print(b.result())  # b will never complete because it is waiting on a.
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result())  # a will never complete because it is waiting on b.
    return 6

executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)

And:

def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # This will never complete because there is only one worker thread and
    # it is executing this function.
    print(f.result())

executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```

class concurrent.futures.ThreadPoolExecutor(max_workers)

An Executor subclass that uses a pool of at most max_workers threads to execute calls asynchronously.

ThreadPoolExecutor Example

```python
import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://some-made-up-domain.com/']

# Retrieve a single page and report the url and contents
def load_url(url, timeout):
    conn = urllib.request.urlopen(url, timeout=timeout)
    return conn.readall()

# We can use a with statement to ensure threads are cleaned up promptly
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    # Start the load operations and mark each future with its URL
    future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
```
```python
data = future.result()
except Exception as exc:
    print('%r generated an exception: %s' % (url, exc))
else:
    print('%r page is %d bytes' % (url, len(data)))
```

### 17.4.3 ProcessPoolExecutor

The `ProcessPoolExecutor` class is an `Executor` subclass that uses a pool of processes to execute calls asynchronously. `ProcessPoolExecutor` uses the `multiprocessing` module, which allows it to side-step the `Global Interpreter Lock` but also means that only picklable objects can be executed and returned.

Calling `Executor` or `Future` methods from a callable submitted to a `ProcessPoolExecutor` will result in deadlock.

```python
class concurrent.futures.ProcessPoolExecutor(max_workers=None)
```

An `Executor` subclass that executes calls asynchronously using a pool of at most `max_workers` processes. If `max_workers` is `None` or not given, it will default to the number of processors on the machine. Changed in version 3.3: When one of the worker processes terminates abruptly, a `BrokenProcessPool` error is now raised. Previously, behaviour was undefined but operations on the executor or its futures would often freeze or deadlock.

#### ProcessPoolExecutor Example

```python
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419
]

def is_prime(n):
    if n % 2 == 0:
        return False
    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print('%d is prime: %s' % (number, prime))

if __name__ == '__main__':
    main()
```

### 17.4.4 Future Objects

The `Future` class encapsulates the asynchronous execution of a callable. `Future` instances are created by `Executor.submit()`.

17.4. concurrent.futures — Launching parallel tasks

---

The Python Library Reference, Release 3.3.3
The Python Library Reference, Release 3.3.3

class concurrent.futures.Future
Encapsulates the asynchronous execution of a callable. Future instances are created by Executor.submit() and should not be created directly except for testing.

cancel()
Attempt to cancel the call. If the call is currently being executed and cannot be cancelled then the method will return False, otherwise the call will be cancelled and the method will return True.

cancelled()
Return True if the call was successfully cancelled.

running()
Return True if the call is currently being executed and cannot be cancelled.

done()
Return True if the call was successfully cancelled or finished running.

result (timeout=None)
Return the value returned by the call. If the call hasn’t yet completed then this method will wait up to timeout seconds. If the call hasn’t completed in timeout seconds, then a TimeoutError will be raised. timeout can be an int or float. If timeout is not specified or None, there is no limit to the wait time.

If the future is cancelled before completing then CancelledError will be raised.

If the call raised, this method will raise the same exception.

exception (timeout=None)
Return the exception raised by the call. If the call hasn’t yet completed then this method will wait up to timeout seconds. If the call hasn’t completed in timeout seconds, then a TimeoutError will be raised. timeout can be an int or float. If timeout is not specified or None, there is no limit to the wait time.

If the future is cancelled before completing then CancelledError will be raised.

If the call completed without raising, None is returned.

add_done_callback (fn)
Attaches the callable fn to the future. fn will be called, with the future as its only argument, when the future is cancelled or finishes running.

Added callables are called in the order that they were added and are always called in a thread belonging to the process that added them. If the callable raises a Exception subclass, it will be logged and ignored. If the callable raises a BaseException subclass, the behavior is undefined.

If the future has already completed or been cancelled, fn will be called immediately.

The following Future methods are meant for use in unit tests and Executor implementations.

set_running_or_notify_cancel()
This method should only be called by Executor implementations before executing the work associated with the Future and by unit tests.

If the method returns False then the Future was cancelled, i.e. Future.cancel() was called and returned True. Any threads waiting on the Future completing (i.e. through as_completed() or wait()) will be woken up.

If the method returns True then the Future was not cancelled and has been put in the running state, i.e. calls to Future.running() will return True.

This method can only be called once and cannot be called after Future.set_result() or Future.set_exception() have been called.

set_result (result)
Sets the result of the work associated with the Future to result.
This method should only be used by Executor implementations and unit tests.

```python
set_exception(exception)
```

Sets the result of the work associated with the Future to the Exception exception.

This method should only be used by Executor implementations and unit tests.

### 17.4.5 Module Functions

```python
concurrent.futures.wait(fs, timeout=None, return_when=ALL_COMPLETED)
```

Wait for the Future instances (possibly created by different Executor instances) given by fs to complete. Returns a named 2-tuple of sets. The first set, named done, contains the futures that completed (finished or were cancelled) before the wait completed. The second set, named not_done, contains uncompleted futures.

*timeout* can be used to control the maximum number of seconds to wait before returning. *timeout* can be an int or float. If *timeout* is not specified or None, there is no limit to the wait time.

*return_when* indicates when this function should return. It must be one of the following constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST_COMPLETED</td>
<td>The function will return when any future finishes or is cancelled.</td>
</tr>
<tr>
<td>FIRST_EXCEPTION</td>
<td>The function will return when any future finishes by raising an exception.</td>
</tr>
<tr>
<td>ALL_COMPLETED</td>
<td>If no future raises an exception then it is equivalent to ALL_COMPLETED.</td>
</tr>
</tbody>
</table>

```python
concurrent.futures.as_completed(fs, timeout=None)
```

Returns an iterator over the Future instances (possibly created by different Executor instances) given by fs that yields futures as they complete (finished or were cancelled). Any futures that completed before as_completed() is called will be yielded first. The returned iterator raises a TimeoutError if __next__() is called and the result isn’t available after *timeout* seconds from the original call to as_completed(). *timeout* can be an int or float. If *timeout* is not specified or None, there is no limit to the wait time.

See Also:

- PEP 3148 – futures - execute computations asynchronously The proposal which described this feature for inclusion in the Python standard library.

### 17.4.6 Exception classes

```python
exception concurrent.futures.BrokenProcessPool
```

Derived from RuntimeError, this exception class is raised when one of the workers of a ProcessPoolExecutor has terminated in a non-clean fashion (for example, if it was killed from the outside). New in version 3.3.

### 17.5 subprocess — Subprocess management

The subprocess module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This module intends to replace several other, older modules and functions, such as:

```python
os.system
os.spawn*
```

Information about how the subprocess module can be used to replace these modules and functions can be found in the following sections.

See Also:

- PEP 324 – PEP proposing the subprocess module
17.5.1 Using the subprocess Module

The recommended approach to invoking subprocesses is to use the following convenience functions for all use cases they can handle. For more advanced use cases, the underlying Popen interface can be used directly.

```
subprocess.call(args, *, stdin=None, stdout=None, stderr=None, shell=False, timeout=None)
```

Run the command described by `args`. Wait for command to complete, then return the `returncode` attribute.

The arguments shown above are merely the most common ones, described below in `Frequently Used Arguments` (hence the use of keyword-only notation in the abbreviated signature). The full function signature is largely the same as that of the Popen constructor - this function passes all supplied arguments other than `timeout` directly through to that interface.

The `timeout` argument is passed to Popen.wait(). If the timeout expires, the child process will be killed and then waited for again. The TimeoutExpired exception will be re-raised after the child process has terminated.

Examples:

```python
>>> subprocess.call(["ls", "-l"])
0

>>> subprocess.call("exit 1", shell=True)
1
```

**Warning:** Invoking the system shell with `shell=True` can be a security hazard if combined with untrusted input. See the warning under `Frequently Used Arguments` for details.

**Note:** Do not use `stdout=PIPE` or `stderr=PIPE` with this function. As the pipes are not being read in the current process, the child process may block if it generates enough output to a pipe to fill up the OS pipe buffer.

Changed in version 3.3: `timeout` was added.

```
subprocess.check_call(args, *, stdin=None, stdout=None, stderr=None, shell=False, timeout=None)
```

Run command with arguments. Wait for command to complete. If the return code was zero then return, otherwise raise `CalledProcessError`. The `CalledProcessError` object will have the return code in the `returncode` attribute.

The arguments shown above are merely the most common ones, described below in `Frequently Used Arguments` (hence the use of keyword-only notation in the abbreviated signature). The full function signature is largely the same as that of the Popen constructor - this function passes all supplied arguments other than `timeout` directly through to that interface.

The `timeout` argument is passed to Popen.wait(). If the timeout expires, the child process will be killed and then waited for again. The TimeoutExpired exception will be re-raised after the child process has terminated.

Examples:

```python
>>> subprocess.check_call(["ls", "-l"])
0

>>> subprocess.check_call("exit 1", shell=True)
Traceback (most recent call last):
...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1
Warning: Invoking the system shell with `shell=True` can be a security hazard if combined with untrusted input. See the warning under Frequently Used Arguments for details.

**Note:** Do not use `stdout=PIPE` or `stderr=PIPE` with this function. As the pipes are not being read in the current process, the child process may block if it generates enough output to a pipe to fill up the OS pipe buffer.

Changed in version 3.3: `timeout` was added.

```python
subprocess.check_output(args, *, stdin=None, stderr=None, shell=False, universal_newlines=False, timeout=None)
```

Run command with arguments and return its output.

If the return code was non-zero it raises a `CalledProcessError`. The `CalledProcessError` object will have the return code in the `returncode` attribute and any output in the `output` attribute.

The arguments shown above are merely the most common ones, described below in Frequently Used Arguments (hence the use of keyword-only notation in the abbreviated signature). The full function signature is largely the same as that of the `Popen` constructor - this functions passes all supplied arguments other than `timeout` directly through to that interface. In addition, `stdout` is not permitted as an argument, as it is used internally to collect the output from the subprocess.

The `timeout` argument is passed to `Popen.wait()`. If the timeout expires, the child process will be killed and then waited for again. The `TimeoutExpired` exception will be re-raised after the child process has terminated.

Examples:

```python
>>> subprocess.check_output(['echo', 'Hello World!'])
b'Hello World!'\n'

>>> subprocess.check_output(['echo', 'Hello World!'], universal_newlines=True)
'Hello World!'\n'

>>> subprocess.check_output(['exit 1'], shell=True)
Traceback (most recent call last):
...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1
```

By default, this function will return the data as encoded bytes. The actual encoding of the output data may depend on the command being invoked, so the decoding to text will often need to be handled at the application level.

This behaviour may be overridden by setting `universal_newlines` to `True` as described below in Frequently Used Arguments.

To also capture standard error in the result, use `stderr=subprocess.STDOUT`:

```python
>>> subprocess.check_output(...
"ls non_existent_file; exit 0",
... stderr=subprocess.STDOUT,
... shell=True)
'ls: non_existent_file: No such file or directory\n'
```

New in version 3.1.

Warning: Invoking the system shell with `shell=True` can be a security hazard if combined with untrusted input. See the warning under Frequently Used Arguments for details.
Note: Do not use `stderr=PIPE` with this function. As the pipe is not being read in the current process, the child process may block if it generates enough output to the pipe to fill up the OS pipe buffer.

 Changed in version 3.3: `timeout` was added.

`subprocess.DEVNULL`

Special value that can be used as the `stdin`, `stdout` or `stderr` argument to `Popen` and indicates that the special file `os.devnull` will be used. New in version 3.3.

`subprocess.PIPE`

Special value that can be used as the `stdin`, `stdout` or `stderr` argument to `Popen` and indicates that a pipe to the standard stream should be opened.

`subprocess.STDOUT`

Special value that can be used as the `stderr` argument to `Popen` and indicates that standard error should go into the same handle as standard output.

`exception subprocess.SubprocessError`

Base class for all other exceptions from this module. New in version 3.3.

`exception subprocess.TimeoutExpired`

Subclass of `SubprocessError`, raised when a timeout expires while waiting for a child process.

```
cmd
Command that was used to spawn the child process.

timeout
Timeout in seconds.

output
Output of the child process if this exception is raised by `check_output()`. Otherwise, None.
```

New in version 3.3.

`exception subprocess.CalledProcessError`

Subclass of `SubprocessError`, raised when a process run by `check_call()` or `check_output()` returns a non-zero exit status.

```
returncode
Exit status of the child process.

cmd
Command that was used to spawn the child process.

output
Output of the child process if this exception is raised by `check_output()`. Otherwise, None.
```

Frequently Used Arguments

To support a wide variety of use cases, the `Popen` constructor (and the convenience functions) accept a large number of optional arguments. For most typical use cases, many of these arguments can be safely left at their default values. The arguments that are most commonly needed are:

- `args` is required for all calls and should be a string, or a sequence of program arguments. Providing a sequence of arguments is generally preferred, as it allows the module to take care of any required escaping and quoting of arguments (e.g., to permit spaces in file names). If passing a single string, either `shell` must be `True` (see below) or else the string must simply name the program to be executed without specifying any arguments.

- `stdin`, `stdout` and `stderr` specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are `PIPE`, `DEVNULL`, an existing file descriptor (a positive integer), an existing file object, and `None`. `PIPE` indicates that a new pipe to the child should be created. `DEVNULL` indicates that the special file `os.devnull` will be used. With the default

---

Chapter 17. Concurrent Execution
settings of `None`, no redirection will occur; the child’s file handles will be inherited from the parent. Additionally, `stderr` can be `STDOUT`, which indicates that the stderr data from the child process should be captured into the same file handle as for `stdout`.

If `universal_newlines` is `False` the file objects `stdin`, `stdout` and `stderr` will be opened as binary streams, and no line ending conversion is done.

If `universal_newlines` is `True`, these file objects will be opened as text streams in `universal newlines` mode using the encoding returned by `locale.getpreferredencoding(False)`. For `stdin`, line ending characters `\n` in the input will be converted to the default line separator `os.linesep`. For `stdout` and `stderr`, all line endings in the output will be converted to `\n`. For more information see the documentation of the `io.TextIOWrapper` class when the `newline` argument to its constructor is `None`.

**Note:** The newlines attribute of the file objects `Popen.stdin`, `Popen.stdout` and `Popen.stderr` are not updated by the `Popen.communicate()` method.

If `shell` is `True`, the specified command will be executed through the shell. This can be useful if you are using Python primarily for the enhanced control flow it offers over most system shells and still want convenient access to other shell features such as shell pipes, filename wildcards, environment variable expansion, and expansion of `~` to a user’s home directory. However, note that Python itself offers implementations of many shell-like features (in particular, `glob`, `fnmatch`, `os.walk()`, `os.path.expandvars()`, `os.path.expanduser()`, and `shutil`). Changed in version 3.3: When `universal_newlines` is `True`, the class uses the encoding `locale.getpreferredencoding(False)` instead of `locale.getpreferredencoding()`. See the `io.TextIOWrapper` class for more information on this change.

**Warning:** Executing shell commands that incorporate unsanitized input from an untrusted source makes a program vulnerable to shell injection, a serious security flaw which can result in arbitrary command execution. For this reason, the use of `shell=True` is strongly discouraged in cases where the command string is constructed from external input:

```python
>>> from subprocess import call
>>> filename = input("What file would you like to display?\n")
What file would you like to display?
non_existent; rm -rf / #
>>> call("cat " + filename, shell=True) # Uh-oh. This will end badly...
shell=False disables all shell based features, but does not suffer from this vulnerability; see the Note in the Popen constructor documentation for helpful hints in getting shell=False to work.
When using shell=True, `shlex.quote()` can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

These options, along with all of the other options, are described in more detail in the `Popen` constructor documentation.

**Popen Constructor**

The underlying process creation and management in this module is handled by the `Popen` class. It offers a lot of flexibility so that developers are able to handle the less common cases not covered by the convenience functions.

```python
class subprocess.Popen(args, bufsize=-1, executable=None, stdin=None, stdout=None, stderr=None, preexec_fn=None, close_fds=True, shell=False, cwd=None, env=None, universal_newlines=False, startupinfo=None, creationflags=0, restore_signals=True, start_new_session=False, pass_fds=())
```

Execute a child program in a new process. On Unix, the class uses `os.execvp()`-like behavior to execute the child program. On Windows, the class uses the Windows `CreateProcess()` function. The arguments to `Popen` are as follows.
args should be a sequence of program arguments or else a single string. By default, the program to execute is the first item in args if args is a sequence. If args is a string, the interpretation is platform-dependent and described below. See the shell and executable arguments for additional differences from the default behavior. Unless otherwise stated, it is recommended to pass args as a sequence.

On Unix, if args is a string, the string is interpreted as the name or path of the program to execute. However, this can only be done if not passing arguments to the program. Note: shlex.split() can be useful when determining the correct tokenization for args, especially in complex cases:

```python
>>> import shlex, subprocess
>>> command_line = input()
/bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
>>> args = shlex.split(command_line)
>>> print(args)
['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd', 'echo '$MONEY'
>>> p = subprocess.Popen(args)  # Success!
```

Note in particular that options (such as -input) and arguments (such as eggs.txt) that are separated by whitespace in the shell go in separate list elements, while arguments that need quoting or backslash escaping when used in the shell (such as filenames containing spaces or the echo command shown above) are single list elements.

On Windows, if args is a sequence, it will be converted to a string in a manner described in Converting an argument sequence to a string on Windows. This is because the underlying CreateProcess() operates on strings.

The shell argument (which defaults to False) specifies whether to use the shell as the program to execute. If shell is True, it is recommended to pass args as a string rather than as a sequence.

On Unix with shell=True, the shell defaults to /bin/sh. If args is a string, the string specifies the command to execute through the shell. This means that the string must be formatted exactly as it would be when typed at the shell prompt. This includes, for example, quoting or backslash escaping filenames with spaces in them. If args is a sequence, the first item specifies the command string, and any additional items will be treated as additional arguments to the shell itself. That is to say, Popen does the equivalent of:

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

On Windows with shell=True, the COMSPEC environment variable specifies the default shell. The only time you need to specify shell=True on Windows is when the command you wish to execute is built into the shell (e.g. dir or copy). You do not need shell=True to run a batch file or console-based executable.

```
Warning: Passing shell=True can be a security hazard if combined with untrusted input. See the warning under Frequently Used Arguments for details.
```

bufsize will be supplied as the corresponding argument to the io.open() function when creating the stdin/stdout/stderr pipe file objects: 0 means unbuffered (read and write are one system call and can return short), 1 means line buffered, any other positive value means use a buffer of approximately that size. A negative bufsize (the default) means the system default of io.DEFAULT_BUFFER_SIZE will be used. Changed in version 3.2.4: 3.3.1 bufsize now defaults to -1 to enable buffering by default to match the behavior that most code expects. In 3.2.0 through 3.2.3 and 3.3.0 it incorrectly defaulted to 0 which was unbuffered and allowed short reads. This was unintentional and did not match the behavior of Python 2 as most code expected.

The executable argument specifies a replacement program to execute. It is very seldom needed. When shell=False, executable replaces the program to execute specified by args. However, the original args is still passed to the program. Most programs treat the program specified by args as the command name, which can then be different from the program actually executed. On Unix, the args name becomes the
display name for the executable in utilities such as `ps`. If `shell=True`, on Unix the `executable` argument specifies a replacement shell for the default `/bin/sh`.

`stdin`, `stdout` and `stderr` specify the executed program’s standard input, standard output and standard error file handles, respectively. Valid values are `PIPE`, `DEVNULL`, an existing file descriptor (a positive integer), an existing `file object`, and `None`. `PIPE` indicates that a new pipe to the child should be created. `DEVNULL` indicates that the special file `os.devnull` will be used. With the default settings of `None`, no redirection will occur; the child’s file handles will be inherited from the parent. Additionally, `stderr` can be `STDOUT`, which indicates that the stderr data from the applications should be captured into the same file handle as for stdout.

If `preexec_fn` is set to a callable object, this object will be called in the child process just before the child is executed. (Unix only)

**Warning:** The `preexec_fn` parameter is not safe to use in the presence of threads in your application. The child process could deadlock before exec is called. If you must use it, keep it trivial! Minimize the number of libraries you call into.

**Note:** If you need to modify the environment for the child use the `env` parameter rather than doing it in a `preexec_fn`. The `start_new_session` parameter can take the place of a previously common use of `preexec_fn` to call `os.setsid()` in the child.

If `close_fds` is true, all file descriptors except 0, 1 and 2 will be closed before the child process is executed. (Unix only). The default varies by platform: Always true on Unix. On Windows it is true when `stdin`/`stdout`/`stderr` are `None`, false otherwise. On Windows, if `close_fds` is true then no handles will be inherited by the child process. Note that on Windows, you cannot set `close_fds` to true and also redirect the standard handles by setting `stdin`, `stdout` or `stderr`. Changed in version 3.2: The default for `close_fds` was changed from `False` to what is described above. `pass_fds` is an optional sequence of file descriptors to keep open between the parent and child. Providing any `pass_fds` forces `close_fds` to be `True`. (Unix only) New in version 3.2: The `pass_fds` parameter was added. If `cwd` is not `None`, the function changes the working directory to `cwd` before executing the child. In particular, the function looks for `executable` (or for the first item in `args`) relative to `cwd` if the executable path is a relative path.

If `restore_signals` is True (the default) all signals that Python has set to `SIG_IGN` are restored to `SIG_DFL` in the child process before the exec. Currently this includes the `SIGPIPE`, `SIGXFZ` and `SIGXFSZ` signals. (Unix only) Changed in version 3.2: `restore_signals` was added. If `start_new_session` is True the `setsid()` system call will be made in the child process prior to the execution of the subprocess. (Unix only) Changed in version 3.2: `start_new_session` was added. If `env` is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process’ environment.

**Note:** If specified, `env` must provide any variables required for the program to execute. On Windows, in order to run a side-by-side assembly the specified `env` must include a valid `SystemRoot`.

If `universal_newlines` is `True`, the file objects `stdin`, `stdout` and `stderr` are opened as text streams in universal newlines mode, as described above in `Frequently Used Arguments`, otherwise they are opened as binary streams.

If given, `startupinfo` will be a `STARTUPINFO` object, which is passed to the underlying `CreateProcess` function. `creationflags`, if given, can be `CREATE_NEW_CONSOLE` or `CREATE_NEW_PROCESS_GROUP`. (Windows only)

Popen objects are supported as context managers via the `with` statement: on exit, standard file descriptors are closed, and the process is waited for.

```python
with Popen(["ifconfig"], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

Changed in version 3.2: Added context manager support.
Exceptions

Exceptions raised in the child process, before the new program has started to execute, will be re-raised in the parent. Additionally, the exception object will have one extra attribute called child_traceback, which is a string containing traceback information from the child’s point of view.

The most common exception raised is OSError. This occurs, for example, when trying to execute a non-existent file. Applications should prepare for OSError exceptions.

A ValueError will be raised if Popen is called with invalid arguments.

check_call() and check_output() will raise CalledProcessError if the called process returns a non-zero return code.

All of the functions and methods that accept a timeout parameter, such as call() and Popen.communicate() will raise TimeoutExpired if the timeout expires before the process exits.

Exceptions defined in this module all inherit from SubprocessError.

New in version 3.3: The SubprocessError base class was added.

Security

Unlike some other popen functions, this implementation will never call a system shell implicitly. This means that all characters, including shell metacharacters, can safely be passed to child processes. Obviously, if the shell is invoked explicitly, then it is the application’s responsibility to ensure that all whitespace and metacharacters are quoted appropriately.

17.5.2 Popen Objects

Instances of the Popen class have the following methods:

Popen.poll()
Check if child process has terminated. Set and return returncode attribute.

Popen.wait(timeout=None)
Wait for child process to terminate. Set and return returncode attribute.

If the process does not terminate after timeout seconds, raise a TimeoutExpired exception. It is safe to catch this exception and retry the wait.

Warning: This will deadlock when using stdout=PIPE and/or stderr=PIPE and the child process generates enough output to a pipe such that it blocks waiting for the OS pipe buffer to accept more data. Use communicate() to avoid that.

Changed in version 3.3: timeout was added.

Popen.communicate(input=None, timeout=None)
Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate. The optional input argument should be data to be sent to the child process, or None, if no data should be sent to the child. The type of input must be bytes or, if universal_newlines was True, a string.

communicate() returns a tuple (stdoutdata, stderrdata).

Note that if you want to send data to the process’s stdin, you need to create the Popen object with stdin=PIPE. Similarly, to get anything other than None in the result tuple, you need to give stdout=PIPE and/or stderr=PIPE too.

If the process does not terminate after timeout seconds, a TimeoutExpired exception will be raised. Catching this exception and retrying communication will not lose any output.

The child process is not killed if the timeout expires, so in order to cleanup properly a well-behaved application should kill the child process and finish communication:
proc = subprocess.Popen(...)  
try:  
    outs, errs = proc.communicate(timeout=15)  
except TimeoutExpired:  
    proc.kill()  
    outs, errs = proc.communicate()  

**Note:** The data read is buffered in memory, so do not use this method if the data size is large or unlimited.

Changed in version 3.3: *timeout* was added.

**Popen.** `send_signal(signal)`

Sends the signal `signal` to the child.

**Note:** On Windows, SIGTERM is an alias for `terminate()`. CTRL_C_EVENT and CTRL_BREAK_EVENT can be sent to processes started with a `creationflags` parameter which includes `CREATE_NEW_PROCESS_GROUP`.

**Popen.** `terminate()`

Stop the child. On Posix OSs the method sends SIGTERM to the child. On Windows the Win32 API function TerminateProcess() is called to stop the child.

**Popen.** `kill()`

Kills the child. On Posix OSs the function sends SIGKILL to the child. On Windows `kill()` is an alias for `terminate()`.

The following attributes are also available:

**Warning:** Use `communicate()` rather than `.stdin.write`, `.stdout.read` or `.stderr.read` to avoid deadlocks due to any of the other OS pipe buffers filling up and blocking the child process.

**Popen.** `stdin`

If the `stdin` argument was `PIPE`, this attribute is a *file object* that provides input to the child process. Otherwise, it is `None`.

**Popen.** `stdout`

If the `stdout` argument was `PIPE`, this attribute is a *file object* that provides output from the child process. Otherwise, it is `None`.

**Popen.** `stderr`

If the `stderr` argument was `PIPE`, this attribute is a *file object* that provides error output from the child process. Otherwise, it is `None`.

**Popen.** `pid`

The process ID of the child process.

Note that if you set the `shell` argument to `True`, this is the process ID of the spawned shell.

**Popen.** `returncode`

The child return code, set by `poll()` and `wait()` (and indirectly by `communicate()`). A `None` value indicates that the process hasn’t terminated yet.

A negative value `-N` indicates that the child was terminated by signal `N` (Unix only).

### 17.5.3 Windows Popen Helpers

The `STARTUPINFO` class and following constants are only available on Windows.

**class** `subprocess.STARTUPINFO`

Partial support of the Windows `STARTUPINFO` structure is used for `Popen` creation.
The Python Library Reference, Release 3.3.3

**dwFlags**
A bit field that determines whether certain `STARTUPINFO` attributes are used when the process creates a window.

```python
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_USESHOWWINDOW
```

**hStdInput**
If `dwFlags` specifies `STARTF_USESTDHANDLES`, this attribute is the standard input handle for the process. If `STARTF_USESTDHANDLES` is not specified, the default for standard input is the keyboard buffer.

**hStdOutput**
If `dwFlags` specifies `STARTF_USESTDHANDLES`, this attribute is the standard output handle for the process. Otherwise, this attribute is ignored and the default for standard output is the console window’s buffer.

**hStdError**
If `dwFlags` specifies `STARTF_USESTDHANDLES`, this attribute is the standard error handle for the process. Otherwise, this attribute is ignored and the default for standard error is the console window’s buffer.

**wShowWindow**
If `dwFlags` specifies `STARTF_USESHOWWINDOW`, this attribute can be any of the values that can be specified in the `nCmdShow` parameter for the `ShowWindow` function, except for `SW_SHOWDEFAULT`. Otherwise, this attribute is ignored.

`SW_HIDE` is provided for this attribute. It is used when `Popen` is called with `shell=True`.

### Constants

The `subprocess` module exposes the following constants.

- **subprocess.STD_INPUT_HANDLE**
  The standard input device. Initially, this is the console input buffer, `CONIN$`.

- **subprocess.STD_OUTPUT_HANDLE**
  The standard output device. Initially, this is the active console screen buffer, `CONOUT$`.

- **subprocess.STD_ERROR_HANDLE**
  The standard error device. Initially, this is the active console screen buffer, `CONOUT$`.

- **subprocess.SW_HIDE**
  Hides the window. Another window will be activated.

- **subprocess.STARTF_USESTDHANDLES**
  Specifies that the `STARTUPINFO.hStdInput`, `STARTUPINFO.hStdOutput`, and `STARTUPINFO.hStdError` attributes contain additional information.

- **subprocess.STARTF_USESHOWWINDOW**
  Specifies that the `STARTUPINFO.wShowWindow` attribute contains additional information.

- **subprocess.CREATE_NEW_CONSOLE**
  The new process has a new console, instead of inheriting its parent’s console (the default).

  This flag is always set when `Popen` is created with `shell=True`.

- **subprocess.CREATE_NEW_PROCESS_GROUP**
  A `Popen` creationflags parameter to specify that a new process group will be created. This flag is necessary for using `os.kill()` on the subprocess.

  This flag is ignored if `CREATE_NEW_CONSOLE` is specified.
17.5.4 Replacing Older Functions with the subprocess Module

In this section, “a becomes b” means that b can be used as a replacement for a.

**Note:** All “a” functions in this section fail (more or less) silently if the executed program cannot be found; the “b” replacements raise OSError instead.

In addition, the replacements using check_output() will fail with a CalledProcessError if the requested operation produces a non-zero return code. The output is still available as the output attribute of the raised exception.

In the following examples, we assume that the relevant functions have already been imported from the subprocess module.

### Replacing /bin/sh shell backquote

```python
output='mycmd myarg'
# becomes
output = check_output(['mycmd', 'myarg'])
```

### Replacing shell pipeline

```python
output='dmesg | grep hda'
# becomes
p1 = Popen(['dmesg'], stdout=PIPE)
p2 = Popen(['grep', 'hda'], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close() # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
```

The `p1.stdout.close()` call after starting the `p2` is important in order for `p1` to receive a SIGPIPE if `p2` exits before `p1`.

Alternatively, for trusted input, the shell’s own pipeline support may still be used directly:

```python
output='dmesg | grep hda'
# becomes
output=check_output("dmesg | grep hda", shell=True)
```

### Replacing os.system()

```python
sts = os.system("mycmd + " myarg")
# becomes
sts = call("mycmd + " myarg", shell=True)
```

Notes:

- Calling the program through the shell is usually not required.

A more realistic example would look like this:

```python
try:
    retcode = call("mycmd + " myarg", shell=True)
    if retcode < 0:
        print("Child was terminated by signal", -retcode, file=sys.stderr)
    else:
        print("Child returned", retcode, file=sys.stderr)
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```
Replacing the `os.spawn` family

**P_NOWAIT example:**

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
```

```python
pid = Popen(['/bin/mycmd', 'myarg']).pid
```

**P_WAIT example:**

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
```

```python
retcode = call(['/bin/mycmd', 'myarg'])
```

**Vector example:**

```
os.spawnvp(os.P_NOWAIT, path, args)
```

```python
Popen([path] + args[1:])
```

**Environment example:**

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
```

```python
Popen(['/bin/mycmd', 'myarg'], env={'PATH': '/usr/bin'})
```

Replacing `os.popen()`, `os.popen2()`, `os.popen3()`

```
(child_stdin, child_stdout) = os.popen2(cmd, mode, bufsize)
```

```python
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

```
(child_stdin, child_stdout, child_stderr) = os.popen3(cmd, mode, bufsize)
```

```python
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin, child_stdout, child_stderr) = (p.stdin, p.stdout, p.stderr)
```

```
(child_stdin, child_stdout_and_stderr) = os.popen4(cmd, mode, bufsize)
```

```python
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```

Return code handling translates as follows:

```
pipe = os.popen(cmd, 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print("There were some errors")
```

```python
process = Popen(cmd, 'w', stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print("There were some errors")
```
Replacing functions from the `popen2` module

**Note:** If the cmd argument to popen2 functions is a string, the command is executed through /bin/sh. If it is a list, the command is directly executed.

```
(child_stdout, child.stdin) = popen2.popen2("somestring", bufsize, mode)
==>
p = Popen(["somestring"], shell=True, bufsize=bufsize,
       stdin=PIPE, stdout=PIPE, close_fds=True)
```

```
(child_stdout, child.stdin) = popen2.popen2(["mycmd", "myarg"], bufsize, mode)
==>
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
       stdin=PIPE, stdout=PIPE, close_fds=True)
```

`popen2.Popen3` and `popen2.Popen4` basically work as `subprocess.Popen`, except that:

- `Popen` raises an exception if the execution fails.
- the `capturestderr` argument is replaced with the `stderr` argument.
- `stdin=PIPE` and `stdout=PIPE` must be specified.
- popen2 closes all file descriptors by default, but you have to specify `close_fds=True` with `Popen` to guarantee this behavior on all platforms or past Python versions.

### 17.5.5 Legacy Shell Invocation Functions

This module also provides the following legacy functions from the 2.x `commands` module. These operations implicitly invoke the system shell and none of the guarantees described above regarding security and exception handling consistency are valid for these functions.

**subprocess.getstatusoutput** *(cmd)*

Return `(status, output)` of executing *cmd* in a shell.

Execute the string *cmd* in a shell with `os.popen()` and return a 2-tuple `(status, output). *cmd* is actually run as `{ cmd ; } 2>&1`, so that the returned output will contain output or error messages. A trailing newline is stripped from the output. The exit status for the command can be interpreted according to the rules for the C function `wait()`. Example:

```python
>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(256, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(256, 'sh: /bin/junk: not found')
```

Availability: UNIX.

**subprocess.getoutput** *(cmd)*

Return output (stdout and stderr) of executing *cmd* in a shell.

Like `getstatusoutput()`, except the exit status is ignored and the return value is a string containing the command’s output. Example:

```python
>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'
```

Availability: UNIX.
17.5.6 Notes

Converting an argument sequence to a string on Windows

On Windows, an *args sequence is converted to a string that can be parsed using the following rules (which correspond to the rules used by the MS C runtime):

1. Arguments are delimited by white space, which is either a space or a tab.
2. A string surrounded by double quotation marks is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.
3. A double quotation mark preceded by a backslash is interpreted as a literal double quotation mark.
4. Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
5. If backslashes immediately precede a double quotation mark, every pair of backslashes is interpreted as a literal backslash. If the number of backslashes is odd, the last backslash escapes the next double quotation mark as described in rule 3.

See Also:

*shlex* Module which provides function to parse and escape command lines.

17.6 sched — Event scheduler

Source code: Lib/sched.py

The *sched* module defines a class which implements a general purpose event scheduler:

```python
class sched.schedulers(timefunc=time.time, delayfunc=time.sleep)
```

The *scheduler* class defines a generic interface to scheduling events. It needs two functions to actually deal with the “outside world” — *timefunc* should be callable without arguments, and return a number (the “time”, in any units whatsoever). If time.monotonic is not available, the *timefunc* default is time.time instead. The *delayfunc* function should be callable with one argument, compatible with the output of *timefunc*, and should delay that many time units. *delayfunc* will also be called with the argument 0 after each event is run to allow other threads an opportunity to run in multi-threaded applications. Changed in version 3.3: *timefunc* and *delayfunc* parameters are optional. Changed in version 3.3: *scheduler* class can be safely used in multi-threaded environments.

Example:

```python
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(a='default'):
...     print("From print_time", time.time(), a)
...
>>> def print_some_times():
...     print(time.time())
...     s.enter(10, 1, print_time)
...     s.enter(5, 2, print_time, argument=(‘positional’,))
...     s.enter(5, 1, print_time, kwargs=‘a’, ‘keyword’)  # Python 2
...     s.run()
...     print(time.time())
...
>>> print_some_times()
930343690.257
From print_time 930343695.274 positional
From print_time 930343695.275 keyword
From print_time 930343700.273 default
930343700.276
```
17.6.1 Scheduler Objects

scheduler instances have the following methods and attributes:

scheduler.enterabs(time, priority, action, argument=(), kwargs={})

Schedule a new event. The time argument should be a numeric type compatible with the return value of the timefunc function passed to the constructor. Events scheduled for the same time will be executed in the order of their priority.

Executing the event means executing action(*argument, **kwargs). argument is a sequence holding the positional arguments for action. kwargs is a dictionary holding the keyword arguments for action.

Return value is an event which may be used for later cancellation of the event (see cancel()). Changed in version 3.3: argument parameter is optional. New in version 3.3: kwargs parameter was added.

scheduler.enter(delay, priority, action, argument=(), kwargs={})

Schedule an event for delay more time units. Other than the relative time, the other arguments, the effect and the return value are the same as those for enterabs(). Changed in version 3.3: argument parameter is optional. New in version 3.3: kwargs parameter was added.

scheduler.cancel(event)

Remove the event from the queue. If event is not an event currently in the queue, this method will raise a ValueError.

scheduler.empty()

Return true if the event queue is empty.

scheduler.run(blocking=True)

Run all scheduled events. This method will wait (using the delayfunc() function passed to the constructor) for the next event, then execute it and so on until there are no more scheduled events.

If blocking is False executes the scheduled events due to expire soonest (if any) and then return the deadline of the next scheduled call in the scheduler (if any).

Either action or delayfunc can raise an exception. In either case, the scheduler will maintain a consistent state and propagate the exception. If an exception is raised by action, the event will not be attempted in future calls to run().

If a sequence of events takes longer to run than the time available before the next event, the scheduler will simply fall behind. No events will be dropped; the calling code is responsible for canceling events which are no longer pertinent. New in version 3.3: blocking parameter was added.

scheduler.queue

Read-only attribute returning a list of upcoming events in the order they will be run. Each event is shown as a named tuple with the following fields: time, priority, action, argument, kwags.

17.7 queue — A synchronized queue class

Source code: Lib/queue.py

The queue module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. The Queue class in this module implements all the required locking semantics. It depends on the availability of thread support in Python; see the threading module.

The module implements three types of queue, which differ only in the order in which the entries are retrieved. In a FIFO queue, the first tasks added are the first retrieved. In a LIFO queue, the most recently added entry is the first retrieved (operating like a stack). With a priority queue, the entries are kept sorted (using the heapq module) and the lowest valued entry is retrieved first.

The queue module defines the following classes and exceptions:
class queue.Queue(maxsize=0)
Constructor for a FIFO queue. maxsize is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If maxsize is less than or equal to zero, the queue size is infinite.

class queue.LifoQueue(maxsize=0)
Constructor for a LIFO queue. maxsize is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If maxsize is less than or equal to zero, the queue size is infinite.

class queue.PriorityQueue(maxsize=0)
Constructor for a priority queue. maxsize is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If maxsize is less than or equal to zero, the queue size is infinite.

The lowest valued entries are retrieved first (the lowest valued entry is the one returned by sorted(list(entries))[0]). A typical pattern for entries is a tuple in the form: (priority_number, data).

exception queue.Empty
Exception raised when non-blocking get() (or get_nowait()) is called on a Queue object which is empty.

exception queue.Full
Exception raised when non-blocking put() (or put_nowait()) is called on a Queue object which is full.

17.7.1 Queue Objects

Queue objects (Queue, LifoQueue, or PriorityQueue) provide the public methods described below.

Queue.qsize()
Return the approximate size of the queue. Note, qsize() > 0 doesn’t guarantee that a subsequent get() will not block, nor will qsize() < maxsize guarantee that put() will not block.

Queue.empty()
Return True if the queue is empty, False otherwise. If empty() returns True it doesn’t guarantee that a subsequent call to put() will not block. Similarly, if empty() returns False it doesn’t guarantee that a subsequent call to get() will not block.

Queue.full()
Return True if the queue is full, False otherwise. If full() returns True it doesn’t guarantee that a subsequent call to get() will not block. Similarly, if full() returns False it doesn’t guarantee that a subsequent call to put() will not block.

Queue.put(item, block=True, timeout=None)
Put item into the queue. If optional args block is true and timeout is None (the default), block if necessary until a free slot is available. If timeout is a positive number, it blocks at most timeout seconds and raises the Full exception if no free slot was available within that time. Otherwise (block is false), put an item on the queue if a free slot is immediately available, else raise the Full exception (timeout is ignored in that case).

Queue.put_nowait(item)
Equivalent to put(item, False).

Queue.get(block=True, timeout=None)
Remove and return an item from the queue. If optional args block is true and timeout is None (the default), block if necessary until an item is available. If timeout is a positive number, it blocks at most timeout seconds and raises the Empty exception if no item was available within that time. Otherwise (block is false), return an item if one is immediately available, else raise the Empty exception (timeout is ignored in that case).

Queue.get_nowait()
Equivalent to get(False).
Two methods are offered to support tracking whether enqueued tasks have been fully processed by daemon consumer threads.

**Queue.task_done()**
Indicate that a formerly enqueued task is complete. Used by queue consumer threads. For each `get()` used to fetch a task, a subsequent call to `task_done()` tells the queue that the processing on the task is complete.

If a `join()` is currently blocking, it will resume when all items have been processed (meaning that a `task_done()` call was received for every item that had been `put()` into the queue).

Raises a `ValueError` if called more times than there were items placed in the queue.

**Queue.join()**
Blocks until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls `task_done()` to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, `join()` unblocks.

Example of how to wait for enqueued tasks to be completed:

```python
def worker():
    while True:
        item = q.get()
        do_work(item)
        q.task_done()

q = Queue()
for i in range(num_worker_threads):
    t = Thread(target=worker)
    t.daemon = True
    t.start()

for item in source():
    q.put(item)

q.join()  # block until all tasks are done
```

See Also:

Class **multiprocessing.Queue** A queue class for use in a multi-processing (rather than multi-threading) context.

collections.deque is an alternative implementation of unbounded queues with fast atomic `append()` and `popleft()` operations that do not require locking.

### 17.8 select — Waiting for I/O completion

This module provides access to the `select()` and `poll()` functions available in most operating systems, `devpoll()` available on Solaris and derivatives, `epoll()` available on Linux 2.5+ and `kqueue()` available on most BSD. Note that on Windows, it only works for sockets; on other operating systems, it also works for other file types (in particular, on Unix, it works on pipes). It cannot be used on regular files to determine whether a file has grown since it was last read.

The module defines the following:

**exception select.error**
A deprecated alias of `OSError`. Changed in version 3.3: Following PEP 3151, this class was made an alias of `OSError`. 

---

17.8. select — Waiting for I/O completion 663
select.\texttt{devpoll()}

(Only supported on Solaris and derivatives.) Returns a /dev/poll polling object; see section /dev/poll Polling Objects below for the methods supported by devpoll objects.

\texttt{devpoll()} objects are linked to the number of file descriptors allowed at the time of instantiation. If your program reduces this value, \texttt{devpoll()} will fail. If your program increases this value, \texttt{devpoll()} may return an incomplete list of active file descriptors. New in version 3.3.

select.\texttt{epoll(sizehint=-1, flags=0)}

(Only supported on Linux 2.5.44 and newer.) Return an edge polling object, which can be used as Edge or Level Triggered interface for I/O events. \texttt{sizehint} is deprecated and completely ignored. \texttt{flags} can be set to EPOLL_CLOEXEC, which causes the epoll descriptor to be closed automatically when \texttt{os.execve()} is called. See section Edge and Level Trigger Polling (epoll) Objects below for the methods supported by epoll objects. Changed in version 3.3: Added the \texttt{flags} parameter.

select.\texttt{poll()}

(Not supported by all operating systems.) Returns a polling object, which supports registering and unregistering file descriptors, and then polling them for I/O events; see section Polling Objects below for the methods supported by polling objects.

select.\texttt{kqueue()}

(Only supported on BSD.) Returns a kernel queue object; see section Kqueue Objects below for the methods supported by kqueue objects.

select.\texttt{kevent(ident, filter=KQ\_FILTER\_READ, flags=KQ\_EV\_ADD, fflags=0, data=0, udata=0)}

(Only supported on BSD.) Returns a kernel event object; see section Kevent Objects below for the methods supported by kevent objects.

select.\texttt{select(rlist, wlist, xlist[, timeout])}

This is a straightforward interface to the Unix \texttt{select()} system call. The first three arguments are sequences of ‘waitable objects’: either integers representing file descriptors or objects with a parameterless method named \texttt{fileno()} returning such an integer:

- \texttt{rlist}: wait until ready for reading
- \texttt{wlist}: wait until ready for writing
- \texttt{xlist}: wait for an “exceptional condition” (see the manual page for what your system considers such a condition)

Empty sequences are allowed, but acceptance of three empty sequences is platform-dependent. (It is known to work on Unix but not on Windows.) The optional \texttt{timeout} argument specifies a time-out as a floating point number in seconds. When the \texttt{timeout} argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

The return value is a triple of lists of objects that are ready: subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned.

Among the acceptable object types in the sequences are Python file objects (e.g. \texttt{sys.stdin}, or objects returned by \texttt{open()} or \texttt{os.popen()}), socket objects returned by \texttt{socket.socket()}. You may also define a wrapper class yourself, as long as it has an appropriate \texttt{fileno()} method (that really returns a file descriptor, not just a random integer).

**Note:** File objects on Windows are not acceptable, but sockets are. On Windows, the underlying \texttt{select()} function is provided by the WinSock library, and does not handle file descriptors that don’t originate from WinSock.

select.\texttt{PIPE\_BUF}

The minimum number of bytes which can be written without blocking to a pipe when the pipe has been reported as ready for writing by \texttt{select()}, \texttt{poll()} or another interface in this module. This doesn’t apply to other kind of file-like objects such as sockets.

This value is guaranteed by POSIX to be at least 512. Availability: Unix. New in version 3.2.
17.8.1 /dev/poll Polling Objects

Solaris and derivatives have /dev/poll. While select() is O(highest file descriptor) and poll() is O(number of file descriptors), /dev/poll is O(active file descriptors).

/dev/poll behaviour is very close to the standard poll() object.

devpoll.register(fd[, eventmask])

Register a file descriptor with the polling object. Future calls to the poll() method will then check whether the file descriptor has any pending I/O events. fd can be either an integer, or an object with a fileno() method that returns an integer. File objects implement fileno(), so they can also be used as the argument.

eventmask is an optional bitmask describing the type of events you want to check for. The constants are the same that with poll() object. The default value is a combination of the constants POLLIN, POLLPRI, and POLLOUT.

Warning: Registering a file descriptor that’s already registered is not an error, but the result is undefined. The appropriate action is to unregister or modify it first. This is an important difference compared with poll().

devpoll.modify(fd[, eventmask])

This method does an unregister() followed by a register(). It is (a bit) more efficient that doing the same explicitly.

devpoll.unregister(fd)

Remove a file descriptor being tracked by a polling object. Just like the register() method, fd can be an integer or an object with a fileno() method that returns an integer.

Attempting to remove a file descriptor that was never registered is safely ignored.

devpoll.poll([timeout])

Polls the set of registered file descriptors, and returns a possibly-empty list containing (fd, event) 2-tuples for the descriptors that have events or errors to report. fd is the file descriptor, and event is a bitmask with bits set for the reported events for that descriptor — POLLIN for waiting input, POLLOUT to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If timeout is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If timeout is omitted, -1, or None, the call will block until there is an event for this poll object.

17.8.2 Edge and Level Trigger Polling (epoll) Objects

http://linux.die.net/man/4/epoll

eventmask

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPOLLIN</td>
<td>Available for read</td>
</tr>
<tr>
<td>EPOLLOUT</td>
<td>Available for write</td>
</tr>
<tr>
<td>EPOLLPRI</td>
<td>Urgent data for read</td>
</tr>
<tr>
<td>EPOLLERR</td>
<td>Error condition happened on the assoc. fd</td>
</tr>
<tr>
<td>EPOLLRHUP</td>
<td>Hang up happened on the assoc. fd</td>
</tr>
<tr>
<td>EPOLLSELECT</td>
<td>Set Edge Trigger behavior, the default is Level Trigger behavior</td>
</tr>
<tr>
<td>EPOLLONESHOT</td>
<td>Set one-shot behavior. After one event is pulled out, the fd is internally disabled</td>
</tr>
<tr>
<td>EPOLLRDNDRAW</td>
<td>Equivalent to EPOLLIN</td>
</tr>
<tr>
<td>EPOLLRDNDRAW</td>
<td>Priority data band can be read.</td>
</tr>
<tr>
<td>EPOLLWRNORM</td>
<td>Equivalent to EPOLLOUT</td>
</tr>
<tr>
<td>EPOLLWRBAND</td>
<td>Priority data may be written.</td>
</tr>
<tr>
<td>EPOLLS</td>
<td>Ignored.</td>
</tr>
</tbody>
</table>
The Python Library Reference, Release 3.3.3

epoll.close()
    Close the control file descriptor of the epoll object.

epoll.fileno()
    Return the file descriptor number of the control fd.

epoll.fromfd(fd)
    Create an epoll object from a given file descriptor.

epoll.register(fd[, eventmask])
    Register a fd descriptor with the epoll object.

epoll.modify(fd, eventmask)
    Modify a register file descriptor.

epoll.unregister(fd)
    Remove a registered file descriptor from the epoll object.

epoll.poll(timeout=-1, maxevents=-1)
    Wait for events. timeout in seconds (float)

17.8.3 Polling Objects

The poll() system call, supported on most Unix systems, provides better scalability for network servers that service many, many clients at the same time. poll() scales better because the system call only requires listing the file descriptors of interest, while select() builds a bitmap, turns on bits for the fds of interest, and then afterward the whole bitmap has to be linearly scanned again. select() is O(highest file descriptor), while poll() is O(number of file descriptors).

poll.register(fd[, eventmask])
    Register a file descriptor with the polling object. Future calls to the poll() method will then check whether the file descriptor has any pending I/O events. fd can be either an integer, or an object with a fileno() method that returns an integer. File objects implement fileno(), so they can also be used as the argument.

eventmask is an optional bitmask describing the type of events you want to check for, and can be a combination of the constants POLLIN, POLLPRI, and POLLOUT, described in the table below. If not specified, the default value used will check for all 3 types of events.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLLIN</td>
<td>There is data to read</td>
</tr>
<tr>
<td>POLLPRI</td>
<td>There is urgent data to read</td>
</tr>
<tr>
<td>POLLOUT</td>
<td>Ready for output: writing will not block</td>
</tr>
<tr>
<td>POLLERR</td>
<td>Error condition of some sort</td>
</tr>
<tr>
<td>POLLHUP</td>
<td>Hung up</td>
</tr>
<tr>
<td>POLLNVAL</td>
<td>Invalid request: descriptor not open</td>
</tr>
</tbody>
</table>

Registering a file descriptor that’s already registered is not an error, and has the same effect as registering the descriptor exactly once.

poll.modify(fd, eventmask)
    Modifies an already registered fd. This has the same effect as register(fd, eventmask). Attempting to modify a file descriptor that was never registered causes an IOError exception with errno ENOENT to be raised.

poll.unregister(fd)
    Remove a file descriptor being tracked by a polling object. Just like the register() method, fd can be an integer or an object with a fileno() method that returns an integer.

Attempting to remove a file descriptor that was never registered causes a KeyError exception to be raised.

poll.poll([timeout])
    Polls the set of registered file descriptors, and returns a possibly-empty list containing (fd, event) 2-tuples for the descriptors that have events or errors to report. fd is the file descriptor, and event is a bitmask
with bits set for the reported events for that descriptor — POLLIN for waiting input, POLLOUT to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If timeout is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If timeout is omitted, negative, or None, the call will block until there is an event for this poll object.

17.8.4 Kqueue Objects

kqueue.close()
Close the control file descriptor of the kqueue object.

kqueue.fileno()
Return the file descriptor number of the control fd.

kqueue.fromfd(fd)
Create a kqueue object from a given file descriptor.

kqueue.control(changelist, max_events[, timeout=None]) → eventlist
Low level interface to kevent
- changelist must be an iterable of kevent object or None
- max_events must be 0 or a positive integer
- timeout in seconds (floats possible)

17.8.5 Kevent Objects

http://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2
kevent.ident
Value used to identify the event. The interpretation depends on the filter but it’s usually the file descriptor.
In the constructor ident can either be an int or an object with a fileno() method. kevent stores the integer internally.

kevent.filter
Name of the kernel filter.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KQ_FILTER_READ</td>
<td>Takes a descriptor and returns whenever there is data available to read</td>
</tr>
<tr>
<td>KQ_FILTER_WRITE</td>
<td>Takes a descriptor and returns whenever there is data available to write</td>
</tr>
<tr>
<td>KQ_FILTER_AIO</td>
<td>AIO requests</td>
</tr>
<tr>
<td>KQ_FILTER_VNODE</td>
<td>Returns when one or more of the requested events watched in flag occurs</td>
</tr>
<tr>
<td>KQ_FILTER_PROC</td>
<td>Watch for events on a process id</td>
</tr>
<tr>
<td>KQ_FILTER_NETDEV</td>
<td>Watch for events on a network device [not available on Mac OS X]</td>
</tr>
<tr>
<td>KQ_FILTER_SIGNAL</td>
<td>Returns whenever the watched signal is delivered to the process</td>
</tr>
<tr>
<td>KQ_FILTER_TIMER</td>
<td>Establishes an arbitrary timer</td>
</tr>
</tbody>
</table>

kevent.flags
Filter action.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>KQ_EV_ADD</td>
<td>Adds or modifies an event</td>
</tr>
<tr>
<td>KQ_EV_DELETE</td>
<td>Removes an event from the queue</td>
</tr>
<tr>
<td>KQ_EV_ENABLE</td>
<td>Permits control() to returns the event</td>
</tr>
<tr>
<td>KQ_EV_DISABLE</td>
<td>Disable event</td>
</tr>
<tr>
<td>KQ_EV_ONESHOT</td>
<td>Removes event after first occurrence</td>
</tr>
<tr>
<td>KQ_EV_CLEAR</td>
<td>Reset the state after an event is retrieved</td>
</tr>
<tr>
<td>KQ_EV_SYSFLAGS</td>
<td>Internal event</td>
</tr>
<tr>
<td>KQ_EV_FLAG1</td>
<td>Internal event</td>
</tr>
<tr>
<td>KQ_EV_EOF</td>
<td>Filter specific EOF condition</td>
</tr>
<tr>
<td>KQ_EV_ERROR</td>
<td>See return values</td>
</tr>
</tbody>
</table>
kevent.\texttt{fflags}
Filter specific flags.

\texttt{KQ\_FILTER\_READ} and \texttt{KQ\_FILTER\_WRITE} filter flags:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{KQ_NOTE_LOWAT}</td>
<td>low water mark of a socket buffer</td>
</tr>
</tbody>
</table>

\texttt{KQ\_FILTER\_VNODE} filter flags:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{KQ_NOTE_DELETE}</td>
<td>\texttt{unlink()} was called</td>
</tr>
<tr>
<td>\texttt{KQ_NOTE_WRITE}</td>
<td>a write occurred</td>
</tr>
<tr>
<td>\texttt{KQ_NOTE_EXTEND}</td>
<td>the file was extended</td>
</tr>
<tr>
<td>\texttt{KQ_NOTE_ATTRIB}</td>
<td>an attribute was changed</td>
</tr>
<tr>
<td>\texttt{KQ_NOTE_LINK}</td>
<td>the link count has changed</td>
</tr>
<tr>
<td>\texttt{KQ_NOTE_RENAME}</td>
<td>the file was renamed</td>
</tr>
<tr>
<td>\texttt{KQ_NOTE_REVOKE}</td>
<td>access to the file was revoked</td>
</tr>
</tbody>
</table>

\texttt{KQ\_FILTER\_PROC} filter flags:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{KQ_NOTE_EXIT}</td>
<td>the process has exited</td>
</tr>
<tr>
<td>\texttt{KQ_NOTE_FORK}</td>
<td>the process has called \texttt{fork()}</td>
</tr>
<tr>
<td>\texttt{KQ_NOTE_EXEC}</td>
<td>the process has executed a new process</td>
</tr>
<tr>
<td>\texttt{KQ_NOTE_PCTRLMASK}</td>
<td>internal filter flag</td>
</tr>
<tr>
<td>\texttt{KQ_NOTE_PDATAMASK}</td>
<td>internal filter flag</td>
</tr>
<tr>
<td>\texttt{KQ_NOTE_TRACK}</td>
<td>follow a process across \texttt{fork()}</td>
</tr>
<tr>
<td>\texttt{KQ_NOTE_CHILD}</td>
<td>returned on the child process for \texttt{NOTE_TRACK}</td>
</tr>
<tr>
<td>\texttt{KQ_NOTE_TRACKERR}</td>
<td>unable to attach to a child</td>
</tr>
</tbody>
</table>

\texttt{KQ\_FILTER\_NETDEV} filter flags (not available on Mac OS X):

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{KQ_NOTE_LINKUP}</td>
<td>link is up</td>
</tr>
<tr>
<td>\texttt{KQ_NOTE_LINKDOWN}</td>
<td>link is down</td>
</tr>
<tr>
<td>\texttt{KQ_NOTE_LINKINV}</td>
<td>link state is invalid</td>
</tr>
</tbody>
</table>

kevent.\texttt{data}
Filter specific data.

kevent.\texttt{udata}
User defined value.

The following are support modules for some of the above services:

17.9 \texttt{dummy\_threading} — Drop-in replacement for the \texttt{threading} module

Source code: Lib/dummy_threading.py

This module provides a duplicate interface to the \texttt{threading} module. It is meant to be imported when the \texttt{_thread} module is not provided on a platform.

Suggested usage is:

\texttt{try:}
\begin{verbatim}
    import threading
except ImportError:
    import dummy_threading as threading
\end{verbatim}
Be careful to not use this module where deadlock might occur from a thread being created that blocks waiting for another thread to be created. This often occurs with blocking I/O.

17.10 _thread — Low-level threading API

This module provides low-level primitives for working with multiple threads (also called light-weight processes or tasks) — multiple threads of control sharing their global data space. For synchronization, simple locks (also called mutexes or binary semaphores) are provided. The threading module provides an easier to use and higher-level threading API built on top of this module.

The module is optional. It is supported on Windows, Linux, SGI IRIX, Solaris 2.x, as well as on systems that have a POSIX thread (a.k.a. “pthread”) implementation. For systems lacking the _thread module, the _dummy_thread module is available. It duplicates this module’s interface and can be used as a drop-in replacement.

It defines the following constants and functions:

exception _thread.error
    Raised on thread-specific errors. Changed in version 3.3: This is now a synonym of the built-in RuntimeError.

_thread.LockType
    This is the type of lock objects.

_thread.start_new_thread(function, args[, kwargs])
    Start a new thread and return its identifier. The thread executes the function function with the argument list args (which must be a tuple). The optional kwargs argument specifies a dictionary of keyword arguments. When the function returns, the thread silently exits. When the function terminates with an unhandled exception, a stack trace is printed and then the thread exits (but other threads continue to run).

_thread.interrupt_main()
    Raise a KeyboardInterrupt exception in the main thread. A subthread can use this function to interrupt the main thread.

_thread.exit()
    Raise the SystemExit exception. When not caught, this will cause the thread to exit silently.

_thread.allocate_lock()
    Return a new lock object. Methods of locks are described below. The lock is initially unlocked.

_thread.get_ident()
    Return the ‘thread identifier’ of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

_thread.stack_size([size])
    Return the thread stack size used when creating new threads. The optional size argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32.768 (32 KiB). If changing the thread stack size is unsupported, a RuntimeError is raised. If the specified stack size is invalid, a ValueError is raised and the stack size is unmodified. 32 KiB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32 KiB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4 KiB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information). Availability: Windows, systems with POSIX threads.

_thread.TIMEOUT_MAX
    The maximum value allowed for the timeout parameter of Lock.acquire(). Specifying a timeout greater than this value will raise an OverflowError. New in version 3.2.

Lock objects have the following methods:
lock.acquire\((\text{waitflag}=1, \text{timeout}=-1)\)

Without any optional argument, this method acquires the lock unconditionally, if necessary waiting until it is released by another thread (only one thread at a time can acquire a lock — that’s their reason for existence).

If the integer \text{waitflag} argument is present, the action depends on its value: if it is zero, the lock is only acquired if it can be acquired immediately without waiting, while if it is nonzero, the lock is acquired unconditionally as above.

If the floating-point \text{timeout} argument is present and positive, it specifies the maximum wait time in seconds before returning. A negative \text{timeout} argument specifies an unbounded wait. You cannot specify a \text{timeout} if \text{waitflag} is zero.

The return value is \text{True} if the lock is acquired successfully, \text{False} if not. Changed in version 3.2: The \text{timeout} parameter is new. Changed in version 3.2: Lock acquires can now be interrupted by signals on POSIX.

lock.release()

Releases the lock. The lock must have been acquired earlier, but not necessarily by the same thread.

lock.locked()

Return the status of the lock: \text{True} if it has been acquired by some thread, \text{False} if not.

In addition to these methods, lock objects can also be used via the \text{with} statement, e.g.:

\begin{verbatim}
import _thread

a_lock = _thread.allocate_lock()

with a_lock:
    print("a_lock is locked while this executes")
\end{verbatim}

Caveats:

- Threads interact strangely with interrupts: the \texttt{KeyboardInterrupt} exception will be received by an arbitrary thread. (When the \texttt{signal} module is available, interrupts always go to the main thread.)
- Calling \texttt{sys.exit()} or raising the \texttt{SystemExit} exception is equivalent to calling _\texttt{thread}.exit().
- Not all built-in functions that may block waiting for I/O allow other threads to run. (The most popular ones (\texttt{time.sleep()}, \texttt{io.FileIO.read()}, \texttt{select.select()}) work as expected.)
- It is not possible to interrupt the \texttt{acquire()} method on a lock — the \texttt{KeyboardInterrupt} exception will happen after the lock has been acquired.
- When the main thread exits, it is system defined whether the other threads survive. On most systems, they are killed without executing \texttt{try ... finally} clauses or executing object destructors.
- When the main thread exits, it does not do any of its usual cleanup (except that \texttt{try ... finally} clauses are honored), and the standard I/O files are not flushed.

17.11 _dummy_thread — Drop-in replacement for the _thread module

Source code: Lib/_dummy_thread.py

This module provides a duplicate interface to the _\texttt{thread} module. It is meant to be imported when the _\texttt{thread} module is not provided on a platform.

Suggested usage is:
try:
    import _thread
except ImportError:
    import _dummy_thread as _thread

Be careful to not use this module where deadlock might occur from a thread being created that blocks waiting for another thread to be created. This often occurs with blocking I/O.
The modules described in this chapter provide mechanisms for different processes to communicate. Some modules only work for two processes that are on the same machine, e.g. `signal` and `mmap`. Other modules support networking protocols that two or more processes can use to communicate across machines.

The list of modules described in this chapter is:

18.1 `socket` — Low-level networking interface

This module provides access to the BSD `socket` interface. It is available on all modern Unix systems, Windows, MacOS, OS/2, and probably additional platforms.

**Note:** Some behavior may be platform dependent, since calls are made to the operating system socket APIs.

The Python interface is a straightforward transliteration of the Unix system call and library interface for sockets to Python’s object-oriented style: the `socket()` function returns a `socket object` whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with `read()` and `write()` operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

**See Also:**

Module `socketserver` Classes that simplify writing network servers.

Module `ssl` A TLS/SSL wrapper for socket objects.

18.1.1 Socket families

Depending on the system and the build options, various socket families are supported by this module.

The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created. Socket addresses are represented as follows:

- The address of an `AF_UNIX` socket bound to a file system node is represented as a string, using the file system encoding and the `surrogateescape` error handler (see PEP 383). An address in Linux’s abstract namespace is returned as a `bytes` object with an initial null byte; note that sockets in this namespace can communicate with normal file system sockets, so programs intended to run on Linux may need to deal with both types of address. A string or `bytes` object can be used for either type of address when passing it as an argument.

  Changed in version 3.3: Previously, `AF_UNIX` socket paths were assumed to use UTF-8 encoding.
• A pair (host, port) is used for the AF_INET address family, where host is a string representing either a hostname in Internet domain notation like ‘daring.cwi.nl’ or an IPv4 address like ‘100.50.200.5’, and port is an integer.

• For AF_INET6 address family, a four-tuple (host, port, flowinfo, scopeid) is used, where flowinfo and scopeid represent the sin6_flowinfo and sin6_scope_id members in struct sockaddr_in6 in C. For socket module methods, flowinfo and scopeid can be omitted just for backward compatibility. Note, however, omission of scopeid can cause problems in manipulating scoped IPv6 addresses.

• AF_NETLINK sockets are represented as pairs (pid, groups).

• Linux-only support for TIPC is available using the AF_TIPC address family. TIPC is an open, non-IP based networked protocol designed for use in clustered computer environments. Addresses are represented by a tuple, and the fields depend on the address type. The general tuple form is (addr_type, v1, v2, v3 [, scope]), where:
  – addr_type is one of TIPC_ADDR_NAMESEQ, TIPC_ADDR_NAME, or TIPC_ADDR_ID.
  – scope is one of TIPC_ZONE_SCOPE, TIPC_CLUSTER_SCOPE, and TIPC_NODE_SCOPE.
  – If addr_type is TIPC_ADDR_NAME, then v1 is the server type, v2 is the port identifier, and v3 should be 0.
  – If addr_type is TIPC_ADDR_NAMESEQ, then v1 is the server type, v2 is the lower port number, and v3 is the upper port number.
  – If addr_type is TIPC_ADDR_ID, then v1 is the node, v2 is the reference, and v3 should be set to 0.

• A tuple (interface, ) is used for the AF_CAN address family, where interface is a string representing a network interface name like ‘can0’. The network interface name can be used to receive packets from all network interfaces of this family.

• A string or a tuple (id, unit) is used for the SYSPROTO_CONTROL protocol of the PF_SYSTEM family. The string is the name of a kernel control using a dynamically-assigned ID. The tuple can be used if ID and unit number of the kernel control are known or if a registered ID is used. New in version 3.3.

• Certain other address families (AF_BLUETOOTH, AF_PACKET) support specific representations.

For IPv4 addresses, two special forms are accepted instead of a host address: the empty string represents INADDR_ANY, and the string ‘broadcast’ represents INADDR_BROADCAST. This behavior is not compatible with IPv6, therefore, you may want to avoid these if you intend to support IPv6 with your Python programs.

If you use a hostname in the host portion of IPv4/v6 socket address, the program may show a nondeterministic behavior, as Python uses the first address returned from the DNS resolution. The socket address will be resolved differently into an actual IPv4/v6 address, depending on the results from DNS resolution and/or the host configuration. For deterministic behavior use a numeric address in host portion.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised; starting from Python 3.3, errors related to socket or address semantics raise OSError or one of its subclasses (they used to raise socket.error).

Non-blocking mode is supported through setblocking(). A generalization of this based on timeouts is supported through settimeout().

18.1.2 Module contents

The module socket exports the following constants and functions:

exception socket.error

A deprecated alias of OSError. Changed in version 3.3: Following PEP 3151, this class was made an alias of OSError.
exception `socket.herror`

A subclass of `OSError`, this exception is raised for address-related errors, i.e. for functions that use `h_errno` in the POSIX C API, including `gethostbyname_ex()` and `gethostbyaddr()`. The accompanying value is a pair `(h_errno, string)` representing an error returned by a library call. `h_errno` is a numeric value, while `string` represents the description of `h_errno`, as returned by the `hstrerror()` C function. Changed in version 3.3: This class was made a subclass of `OSError`.

exception `socket.gaierror`

A subclass of `OSError`, this exception is raised for address-related errors by `getaddrinfo()` and `getnameinfo()`. The accompanying value is a pair `(error, string)` representing an error returned by a library call. `string` represents the description of `error`, as returned by the `gai_strerror()` C function. The numeric `error` value will match one of the EAI_* constants defined in this module. Changed in version 3.3: This class was made a subclass of `OSError`.

exception `socket.timeout`

A subclass of `OSError`, this exception is raised when a timeout occurs on a socket which has had timeouts enabled via a prior call to `settimeout()` (or implicitly through `setdefaulttimeout()`). The accompanying value is a string whose value is currently always “timed out”. Changed in version 3.3: This class was made a subclass of `OSError`.

`socket.AF_UNIX`

`socket.AF_INET`

`socket.AF_INET6`

These constants represent the address (and protocol) families, used for the first argument to `socket()`. If the `AF_UNIX` constant is not defined then this protocol is unsupported. More constants may be available depending on the system.

`socket.SOCK_STREAM`

`socket.SOCK_DGRAM`

`socket.SOCK_RAW`

`socket.SOCK_RDM`

`socket.SOCK_SEQPACKET`

These constants represent the socket types, used for the second argument to `socket()`. More constants may be available depending on the system. (Only `SOCK_STREAM` and `SOCK_DGRAM` appear to be generally useful.)

`socket.SOCK_CLOEXEC`

`socket.SOCK_NONBLOCK`

These two constants, if defined, can be combined with the socket types and allow you to set some flags atomically (thus avoiding possible race conditions and the need for separate calls).

See Also:

Secure File Descriptor Handling for a more thorough explanation.

Availability: Linux >= 2.6.27. New in version 3.2.

`SO_*`

`socket.SOMAXCONN`

`MSG_*`

`SOL_*`

`SCM_*`

`IPPROTO_*`

`IPPORT_*`

`INADDR_*`

`IP_*`

`IPV6_*`

`EAI_*`

`AI_*`

`NI_*`

`TCP_*`

Many constants of these forms, documented in the Unix documentation on sockets and/or the IP protocol, are also defined in the socket module. They are generally used in arguments to the `setsockopt()` and
getsockopt() methods of socket objects. In most cases, only those symbols that are defined in the Unix
header files are defined; for a few symbols, default values are provided.

socket.AF_CAN
socket.PF_CAN
SOL_CAN_*
CAN_*
Many constants of these forms, documented in the Linux documentation, are also defined in the socket
module.
Availability: Linux >= 2.6.25. New in version 3.3.

socket.AF_RDS
socket.PF_RDS
socket.SOL_RDS
RDS_*
Many constants of these forms, documented in the Linux documentation, are also defined in the socket
module.
Availability: Linux >= 2.6.30. New in version 3.3.

SIO_*
RCVALL_*
Constants for Windows' WSAIoctl(). The constants are used as arguments to the ioctl() method of
socket objects.

TIPC_*
TIPC related constants, matching the ones exported by the C socket API. See the TIPC documentation for
more information.

socket.has_ipv6
This constant contains a boolean value which indicates if IPv6 is supported on this platform.

socket.create_connection(address[, timeout[, source_address]])
Connect to a TCP service listening on the Internet address (a 2-tuple (host, port)), and return the
socket object. This is a higher-level function than socket.connect(): if host is a non-numeric host-
name, it will try to resolve it for both AF_INET and AF_INET6, and then try to connect to all possible
addresses in turn until a connection succeeds. This makes it easy to write clients that are compatible to both
IPv4 and IPv6.

Passing the optional timeout parameter will set the timeout on the socket instance before attempting to con-
nect. If no timeout is supplied, the global default timeout setting returned by getdefaulttimeout() is
used.

If supplied, source_address must be a 2-tuple (host, port) for the socket to bind to as its source address
before connecting. If host or port are "" or 0 respectively the OS default behavior will be used. Changed in
version 3.2: source_address was added.Changed in version 3.2: support for the with statement was added.

socket.getaddrinfo(host, port, family=0, type=0, proto=0, flags=0)
Translate the host/port argument into a sequence of 5-tuples that contain all the necessary arguments for
creating a socket connected to that service. host is a domain name, a string representation of an IPv4/v6
address or None. port is a string service name such as 'http', a numeric port number or None. By
passing None as the value of host and port, you can pass NULL to the underlying C API.

The family, type and proto arguments can be optionally specified in order to narrow the list of addresses
returned. Passing zero as a value for each of these arguments selects the full range of results. The flags
argument can be one or several of the AI_* constants, and will influence how results are computed and
returned. For example, AI_NUMERICHOST will disable domain name resolution and will raise an error if
host is a domain name.

The function returns a list of 5-tuples with the following structure:

(family, type, proto, canonname, sockaddr)

In these tuples, family, type, proto are all integers and are meant to be passed to the socket() func-
tion. canonname will be a string representing the canonical name of the host if AI_CANONNAME is
part of the flags argument; else canonname will be empty. sockaddr is a tuple describing a socket address, whose format depends on the returned family (a (address, port) 2-tuple for AF_INET, a (address, port, flow info, scope id) 4-tuple for AF_INET6), and is meant to be passed to the socket.connect() method.

The following example fetches address information for a hypothetical TCP connection to www.python.org on port 80 (results may differ on your system if IPv6 isn’t enabled):

```python
>>> socket.getaddrinfo("www.python.org", 80, proto=socket.SOL_TCP)
[(2, 1, 6, '', ('82.94.164.162', 80)),
 (10, 1, 6, '', ('2001:888:2000:d::a2', 80, 0, 0))]
```

Changed in version 3.2: parameters can now be passed as single keyword arguments.

socket.getfqdn([name])

Return a fully qualified domain name for name. If name is omitted or empty, it is interpreted as the local host. To find the fully qualified name, the hostname returned by gethostbyaddr() is checked, followed by aliases for the host, if available. The first name which includes a period is selected. In case no fully qualified domain name is available, the hostname as returned by gethostname() is returned.

socket.gethostname()  

Return a string containing the hostname of the machine where the Python interpreter is currently executing. If you want to know the current machine’s IP address, you may want to use gethostbyaddr(gethostname()). This operation assumes that there is a valid address-to-host mapping for the host, and the assumption does not always hold. Note: gethostname() doesn’t always return the fully qualified domain name; use getfqdn() (see above).

socket.gethostbyaddr(ip_address)  

Return a triple (hostname, aliaslist, ipaddrlist) where hostname is the primary host name responding to the given ip_address, aliaslist is a (possibly empty) list of alternative host names for the same address, and ipaddrlist is a list of IPv4 addresses for the same interface on the same host (often but not always a single address). To find the fully qualified domain name, use the function getfqdn(). gethostbyaddr() supports both IPv4 and IPv6.

socket.getnameinfo(sockaddr, flags)  

Translate a socket address sockaddr into a 2-tuple (host, port). Depending on the settings of flags, the result can contain a fully-qualified domain name or numeric address representation in host. Similarly, port can contain a string port name or a numeric port number.

socket.getprotobyname(protocolname)  

Translate an Internet protocol name (for example, ‘icmp’) to a constant suitable for passing as the (optional) third argument to the socket() function. This is usually only needed for sockets opened in “raw” mode (SOCK_RAW); for the normal socket modes, the correct protocol is chosen automatically if the protocol is omitted or zero.
socket.getservbyname(servicename[, protocolname])

Translate an Internet service name and protocol name to a port number for that service. The optional protocol name, if given, should be ‘tcp’ or ‘udp’, otherwise any protocol will match.

socket.getservbyport(port[, protocolname])

Translate an Internet port number and protocol name to a service name for that service. The optional protocol name, if given, should be ‘tcp’ or ‘udp’, otherwise any protocol will match.

socket.socket([family[, type[, proto]]])

Create a new socket using the given address family, socket type and protocol number. The address family should be AF_INET (the default), AF_INET6, AF_UNIX, AF_CAN or AF_RDS. The socket type should be SOCK_STREAM (the default), SOCK_DGRAM, SOCK_RAW or perhaps one of the other SOCK_* constants. The protocol number is usually zero and may be omitted in that case or CAN_RAW in case the address family is AF_CAN. Changed in version 3.3: The AF_CAN family was added. The AF_RDS family was added.

socket.socketpair([family[, type[, proto]]])

Build a pair of connected socket objects using the given address family, socket type, and protocol number. Address family, socket type, and protocol number are as for the socket() function above. The default family is AF_UNIX if defined on the platform; otherwise, the default is AF_INET. Availability: Unix. Changed in version 3.2: The returned socket objects now support the whole socket API, rather than a subset.

socket.fromfd(fd, family, type[, proto])

Duplicate the file descriptor fd (an integer as returned by a file object’s fileno() method) and build a socket object from the result. Address family, socket type and protocol number are as for the socket() function above. The file descriptor should refer to a socket, but this is not checked — subsequent operations on the object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (such as a server started by the Unix inet daemon). The socket is assumed to be in blocking mode.

socket.ntohl(x)

Convert 32-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.ntohs(x)

Convert 16-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

socket.htonl(x)

Convert 32-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.htons(x)

Convert 16-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

socket.inet_aton(ip_string)

Convert an IPv4 address from dotted-quad string format (for example, ‘123.45.67.89’) to 32-bit packed binary format, as a bytes object four characters in length. This is useful when conversing with a program that uses the standard C library and needs objects of type struct in_addr, which is the C type for the 32-bit packed binary this function returns.

inet_aton() also accepts strings with less than three dots; see the Unix manual page inet(3) for details.

If the IPv4 address string passed to this function is invalid, OSError will be raised. Note that exactly what is valid depends on the underlying C implementation of inet_aton().

inet_aton() does not support IPv6, and inet_pton() should be used instead for IPv4/IPv6 dual stack support.

socket.inet_ntoa(packed_ip)

Convert a 32-bit packed IPv4 address (a bytes object four characters in length) to its standard dotted-quad string representation (for example, ‘123.45.67.89’). This is useful when conversing with a program that
uses the standard C library and needs objects of type `struct in_addr`, which is the C type for the 32-bit packed binary data this function takes as an argument.

If the byte sequence passed to this function is not exactly 4 bytes in length, `OSError` will be raised. `inet_ntoa()` does not support IPv6, and `inet_ntop()` should be used instead for IPv4/IPv6 dual stack support.

```python
socket.inet_ntop(address_family, ip_string)
```
Convert an IP address from its family-specific string format to a packed, binary format. `inet_ntop()` is useful when a library or network protocol calls for an object of type `struct in_addr` (similar to `inet_aton()`) or `struct in6_addr`.

Supported values for `address_family` are currently `AF_INET` and `AF_INET6`. If the IP address string `ip_string` is invalid, `OSError` will be raised. Note that exactly what is valid depends on both the value of `address_family` and the underlying implementation of `inet_ntop()`.

Availability: Unix (maybe not all platforms).

```python
socket.inet_ntop(address_family, packed_ip)
```
Convert a packed IP address (a bytes object of some number of characters) to its standard, family-specific string representation (for example, ‘7.10.0.5’ or ‘5aef:2b::8’). `inet_ntop()` is useful when a library or network protocol returns an object of type `struct in_addr` (similar to `inet_ntoa()`) or `struct in6_addr`.

Supported values for `address_family` are currently `AF_INET` and `AF_INET6`. If the string `packed_ip` is not the correct length for the specified address family, `ValueError` will be raised. A `OSError` is raised for errors from the call to `inet_ntop()`.

Availability: Unix (maybe not all platforms).

```python
socket.CMSG_LEN(length)
```
Return the total length, without trailing padding, of an ancillary data item with associated data of the given `length`. This value can often be used as the buffer size for `recvmsg()` to receive a single item of ancillary data, but RFC 3542 requires portable applications to use `CMSG_SPACE()` and thus include space for padding, even when the item will be the last in the buffer. Raises `OverflowError` if `length` is outside the permissible range of values.

Availability: most Unix platforms, possibly others. New in version 3.3.

```python
socket.CMSG_SPACE(length)
```
Return the buffer size needed for `recvmsg()` to receive an ancillary data item with associated data of the given `length`, along with any trailing padding. The buffer space needed to receive multiple items is the sum of the `CMSG_SPACE()` values for their associated data lengths. Raises `OverflowError` if `length` is outside the permissible range of values.

Note that some systems might support ancillary data without providing this function. Also note that setting the buffer size using the results of this function may not precisely limit the amount of ancillary data that can be received, since additional data may be able to fit into the padding area.

Availability: most Unix platforms, possibly others. New in version 3.3.

```python
socket.getdefaulttimeout()
```
Return the default timeout in seconds (float) for new socket objects. A value of `None` indicates that new socket objects have no timeout. When the socket module is first imported, the default is `None`.

```python
socket.setdefaulttimeout(timeout)
```
Set the default timeout in seconds (float) for new socket objects. When the socket module is first imported, the default is `None`. See `settimeout()` for possible values and their respective meanings.

```python
socket.sethostname(name)
```
Set the machine’s hostname to `name`. This will raise an `OSError` if you don’t have enough rights.

Availability: Unix. New in version 3.3.

```python
socket.if_nameindex()
```
Return a list of network interface information (index int, name string) tuples. `OSError` if the system call fails.
Availability: Unix. New in version 3.3.

```python
socket.if_nametoindex(if_name)
```
Return a network interface index number corresponding to an interface name. `OSError` if no interface with the given name exists.

Availability: Unix. New in version 3.3.

```python
socket.if_indextoname(if_index)
```
Return a network interface name corresponding to a interface index number. `OSError` if no interface with the given index exists.

Availability: Unix. New in version 3.3.

```python
socket.fromshare(data)
```
Instantiate a socket from data obtained from `share()`. The socket is assumed to be in blocking mode.


```python
socket.SocketType
```
This is a Python type object that represents the socket object type. It is the same as `type(socket(...))`.

### 18.1.3 Socket Objects

Socket objects have the following methods. Except for `makefile()` these correspond to Unix system calls applicable to sockets.

```python
socket.accept()
```
Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair (`conn`, `address`) where `conn` is a new socket object usable to send and receive data on the connection, and `address` is the address bound to the socket on the other end of the connection.

```python
socket.bind(address)
```
Bind the socket to `address`. The socket must not already be bound. (The format of `address` depends on the address family — see above.)

```python
socket.close()
```
Close the socket. All future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

**Note:** `close()` releases the resource associated with a connection but does not necessarily close the connection immediately. If you want to close the connection in a timely fashion, call `shutdown()` before `close()`.

```python
socket.connect(address)
```
Connect to a remote socket at `address`. (The format of `address` depends on the address family — see above.)

```python
socket.connect_ex(address)
```
Like `connect(address)`, but return an error indicator instead of raising an exception for errors returned by the C-level `connect()` call (other problems, such as “host not found,” can still raise exceptions). The error indicator is 0 if the operation succeeded, otherwise the value of the `errno` variable. This is useful to support, for example, asynchronous connects.

```python
socket.detach()
```
Put the socket object into closed state without actually closing the underlying file descriptor. The file descriptor is returned, and can be reused for other purposes. New in version 3.2.

```python
socket.fileno()
```
Return the socket’s file descriptor (a small integer). This is useful with `select.select()`.

Under Windows the small integer returned by this method cannot be used where a file descriptor can be used (such as `os.fdopen()`). Unix does not have this limitation.
socket.getpeername()

Return the remote address to which the socket is connected. This is useful to find out the port number of a remote IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.) On some systems this function is not supported.

socket.getsockname()

Return the socket’s own address. This is useful to find out the port number of an IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.)

socket.getsockopt(level, optname[, buflen])

Return the value of the given socket option (see the Unix man page getsockopt(2)). The needed symbolic constants (SO_* etc.) are defined in this module. If buflen is absent, an integer option is assumed and its integer value is returned by the function. If buflen is present, it specifies the maximum length of the buffer used to receive the option in, and this buffer is returned as a bytes object. It is up to the caller to decode the contents of the buffer (see the optional built-in module struct for a way to decode C structures encoded as byte strings).

socket.gettimeout()

Return the timeout in seconds (float) associated with socket operations, or None if no timeout is set. This reflects the last call to setblocking() or settimeout().

socket.ioctl(control, option)

Platform Windows

The ioctl() method is a limited interface to the WSAIoctl system interface. Please refer to the Win32 documentation for more information.

On other platforms, the generic fcntl.fcntl() and fcntl.ioctl() functions may be used; they accept a socket object as their first argument.

socket.listen(backlog)

Listen for connections made to the socket. The backlog argument specifies the maximum number of queued connections and should be at least 0; the maximum value is system-dependent (usually 5), the minimum value is forced to 0.

socket.makefile(mode='r', buffering=None, *, encoding=None, errors=None, newline=None)

Return a file object associated with the socket. The exact returned type depends on the arguments given to makefile(). These arguments are interpreted the same way as by the built-in open() function.

Closing the file object won’t close the socket unless there are no remaining references to the socket. The socket must be in blocking mode; it can have a timeout, but the file object’s internal buffer may end up in an inconsistent state if a timeout occurs.

Note: On Windows, the file-like object created by makefile() cannot be used where a file object with a file descriptor is expected, such as the stream arguments of subprocess.Popen().

socket.recv(bufsize[, flags])

Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by bufsize. See the Unix manual page recv(2) for the meaning of the optional argument flags; it defaults to zero.

Note: For best match with hardware and network realities, the value of bufsize should be a relatively small power of 2, for example, 4096.

socket.recvfrom(bufsize[, flags])

Receive data from the socket. The return value is a pair (bytes, address) where bytes is a bytes object representing the data received and address is the address of the socket sending the data. See the Unix manual page recv(2) for the meaning of the optional argument flags; it defaults to zero. (The format of address depends on the address family — see above.)
socket.recvmsg([bufsize[, ancbufsize[, flags]]])

Receive normal data (up to bufsize bytes) and ancillary data from the socket. The ancbufsize argument sets the size in bytes of the internal buffer used to receive the ancillary data; it defaults to 0, meaning that no ancillary data will be received. Appropriate buffer sizes for ancillary data can be calculated using CMSG_SPACE() or CMSG_LEN(), and items which do not fit into the buffer might be truncated or discarded. The flags argument defaults to 0 and has the same meaning as for recv().

The return value is a 4-tuple: (data, ancdata, msg_flags, address). The data item is a bytes object holding the non-ancillary data received. The ancdata item is a list of zero or more tuples (cmsg_level, cmsg_type, cmsg_data) representing the ancillary data (control messages) received: cmsg_level and cmsg_type are integers specifying the protocol level and protocol-specific type respectively, and cmsg_data is a bytes object holding the associated data. The msg_flags item is the bitwise OR of various flags indicating conditions on the received message; see your system documentation for details. If the receiving socket is unconnected, address is the address of the sending socket, if available; otherwise, its value is unspecified.

On some systems, sendmsg() and recvmsg() can be used to pass file descriptors between processes over an AF_UNIX socket. When this facility is used (it is often restricted to SOCK_STREAM sockets), recvmsg() will return, in its ancillary data, items of the form (socket.SOL_SOCKET, socket.SCM_RIGHTS, fds), where fds is a bytes object representing the new file descriptors as a binary array of the native C int type. If recvmsg() raises an exception after the system call returns, it will first attempt to close any file descriptors received via this mechanism.

Some systems do not indicate the truncated length of ancillary data items which have been only partially received. If an item appears to extend beyond the end of the buffer, recvmsg() will issue a RuntimeWarning, and will return the part of it which is inside the buffer provided it has not been truncated before the start of its associated data.

On systems which support the SCM_RIGHTS mechanism, the following function will receive up to maxfds file descriptors, returning the message data and a list containing the descriptors (while ignoring unexpected conditions such as unrelated control messages being received). See also sendmsg().

```python
import socket, array

def recv_fds(sock, msglen, maxfds):
    fds = array.array("i")  # Array of ints
    msg, ancdata, flags, addr = sock.recvmsg(msglen, socket.CMSG_LEN(maxfds * fds.itemsize))
    for cmsg_level, cmsg_type, cmsg_data in ancdata:
        if (cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_RIGHTS):
            # Append data, ignoring any truncated integers at the end.
            fds.fromstring(cmsg_data[:len(cmsg_data) - (len(cmsg_data) % fds.itemsize)])
    return msg, list(fds)
```

Availability: most Unix platforms, possibly others. New in version 3.3.

socket.recvmsg_into([buffers[, ancbufsize[, flags]]])

Receive normal data and ancillary data from the socket, behaving as recvmsg() would, but scatter the non-ancillary data into a series of buffers instead of returning a new bytes object. The buffers argument must be an iterable of objects that export writable buffers (e.g. bytearray objects); these will be filled with successive chunks of the non-ancillary data until it has all been written or there are no more buffers. The operating system may set a limit (sysconf() value SC_IOV_MAX) on the number of buffers that can be used. The ancbufsize and flags arguments have the same meaning as for recvmsg().

The return value is a 4-tuple: (nbytes, ancdata, msg_flags, address), where nbytes is the total number of bytes of non-ancillary data written into the buffers, and ancdata, msg_flags and address are the same as for recvmsg().

Example:

```python
>>> import socket
>>> s1, s2 = socket.socketpair()
>>> b1 = bytearray(b'----')
```
>>> b2 = bytearray(b'0123456789')
>>> b3 = bytearray(b'--------------')
>>> s1.send(b'Mary had a little lamb')
22
>>> s2.recvmsg_into([b1, memoryview(b2)[2:9], b3])
(22, [], 0, None)
>>> [b1, b2, b3]
[bytearray(b'Mary'), bytearray(b'01 had a 9'), bytearray(b'little lamb---')]
Availability: most Unix platforms, possibly others. New in version 3.3.

socket.setblocking(flag)
Set blocking or non-blocking mode of the socket: if flag is false, the socket is set to non-blocking, else to blocking mode.

This method is a shorthand for certain settimeout() calls:

• sock.setblocking(True) is equivalent to sock.settimeout(None)
• sock.setblocking(False) is equivalent to sock.settimeout(0.0)

socket.settimeout(value)
Set a timeout on blocking socket operations. The value argument can be a nonnegative floating point number expressing seconds, or None. If a non-zero value is given, subsequent socket operations will raise a timeout exception if the timeout period value has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If None is given, the socket is put in blocking mode.

For further information, please consult the notes on socket timeouts.

socket.setsockopt(level, optname, value)
Set the value of the given socket option (see the Unix manual page setsockopt(2)). The needed symbolic constants are defined in the socket module (SO_* etc.). The value can be an integer or a bytes object representing a buffer. In the latter case it is up to the caller to ensure that the bytestring contains the proper bits (see the optional built-in module struct for a way to encode C structures as bytestrings).

socket.shutdown(how)
Shut down one or both halves of the connection. If how is SHUT_RD, further receives are disallowed. If how is SHUT_WR, further sends are disallowed. If how is SHUT_RDWR, further sends and receives are disallowed.

socket.share(process_id)

platform Windows
Duplacet a socket and prepare it for sharing with a target process. The target process must be provided with process_id. The resulting bytes object can then be passed to the target process using some form of interprocess communication and the socket can be recreated there using fromshare(). Once this method has been called, it is safe to close the socket since the operating system has already duplicated it for the target process.

New in version 3.3.

Note that there are no methods read() or write(); use recv() and send() without flags argument instead.

Socket objects also have these (read-only) attributes that correspond to the values given to the socket constructor.

socket.family
The socket family.

socket.type
The socket type.

socket.proto
The socket protocol.

18.1.4 Notes on socket timeouts

A socket object can be in one of three modes: blocking, non-blocking, or timeout. Sockets are by default always created in blocking mode, but this can be changed by calling setdefaulttimeout().

• In blocking mode, operations block until complete or the system returns an error (such as connection timed out).

• In non-blocking mode, operations fail (with an error that is unfortunately system-dependent) if they cannot be completed immediately: functions from the select can be used to know when and whether a socket is available for reading or writing.
In **timeout mode**, operations fail if they cannot be completed within the timeout specified for the socket (they raise a `timeout` exception) or if the system returns an error.

**Note:** At the operating system level, sockets in **timeout mode** are internally set in non-blocking mode. Also, the blocking and timeout modes are shared between file descriptors and socket objects that refer to the same network endpoint. This implementation detail can have visible consequences if e.g. you decide to use the `fileno()` of a socket.

### Timeouts and the `connect` method

The `connect()` operation is also subject to the timeout setting, and in general it is recommended to call `settimeout()` before calling `connect()` or pass a timeout parameter to `create_connection()`. However, the system network stack may also return a connection timeout error of its own regardless of any Python socket timeout setting.

### Timeouts and the `accept` method

If `getdefaulttimeout()` is not `None`, sockets returned by the `accept()` method inherit that timeout. Otherwise, the behaviour depends on settings of the listening socket:

- if the listening socket is in **blocking mode** or in **timeout mode**, the socket returned by `accept()` is in **blocking mode**;
- if the listening socket is in **non-blocking mode**, whether the socket returned by `accept()` is in blocking or non-blocking mode is operating system-dependent. If you want to ensure cross-platform behaviour, it is recommended you manually override this setting.

#### 18.1.5 Example

Here are four minimal example programs using the TCP/IP protocol: a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence `socket(), bind(), listen(), accept()` (possibly repeating the `accept()` to service more than one client), while a client only needs the sequence `socket(), connect()`. Also note that the server does not `sendall()`/`recv()` on the socket it is listening on but on the new socket returned by `accept()`.

The first two examples support IPv4 only.

```python
# Echo server program
import socket

HOST = ''  # Symbolic name meaning all available interfaces
PORT = 50007  # Arbitrary non-privileged port

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print('Connected by', addr)
while True:
    data = conn.recv(1024)
    if not data: break
    conn.sendall(data)
conn.close()

# Echo client program
import socket

HOST = 'daring.cwi.nl'  # The remote host
```
PORT = 50007  # The same port as used by the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.sendall(b'Hello, world')
data = s.recv(1024)
s.close()
print('Received', repr(data))

The next two examples are identical to the above two, but support both IPv4 and IPv6. The server side will listen to the first address family available (it should listen to both instead). On most of IPv6-ready systems, IPv6 will take precedence and the server may not accept IPv4 traffic. The client side will try to connect to the all addresses returned as a result of the name resolution, and sends traffic to the first one connected successfully.

# Echo server program
import socket
import sys
HOST = None  # Symbolic name meaning all available interfaces
PORT = 50007  # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
try:
    s = socket.socket(af, socktype, proto)
except OSError as msg:
    s = None
    continue
try:
    s.bind(sa)
    s.listen(1)
except OSError as msg:
    s.close()
    s = None
    continue
break
if s is None:
    print('could not open socket')
sys.exit(1)
conn, addr = s.accept()
print('Connected by', addr)
while True:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()

# Echo client program
import socket
import sys
HOST = 'daring.cwi.nl'  # The remote host
PORT = 50007  # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
try:
    s = socket.socket(af, socktype, proto)
except OSError as msg:
    s = None
continue
try:
    s.connect(sa)
except OSError as msg:
    s.close()
    s = None
continue
break
if s is None:
    print('could not open socket')
    sys.exit(1)
s.sendall(b'Hello, world')
data = s.recv(1024)
s.close()
print('Received', repr(data))

The next example shows how to write a very simple network sniffer with raw sockets on Windows. The example requires administrator privileges to modify the interface:

```python
import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a package
print(s.recvfrom(65565))

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

The last example shows how to use the socket interface to communicate to a CAN network. This example might require special priviledge:

```python
import socket
import struct

# CAN frame packing/unpacking (see ‘struct can_frame’ in <linux/can.h>)
can_frame_fmt = '=IB3x8s'
can_frame_size = struct.calcsize(can_frame_fmt)

def build_can_frame(can_id, data):
    can_dlc = len(data)
    data = data.ljust(8, b'\x00')
    return struct.pack(can_frame_fmt, can_id, can_dlc, data)

def dissect_can_frame(frame):
    can_id, can_dlc, data = struct.unpack(can_frame_fmt, frame)
    return (can_id, can_dlc, data[:can_dlc])
```
# create a raw socket and bind it to the ‘vcan0’ interface
s = socket.socket(socket.AF_CAN, socket.SOCK_RAW, socket.CAN_RAW)
s.bind(('vcan0',))

while True:
    cf, addr = s.recvfrom(can_frame_size)

    print('Received: can_id=%x, can_dlc=%x, data=%s' % dissect_can_frame(cf))

    try:
        s.send(cf)
    except OSError:
        print('Error sending CAN frame')

    try:
        s.send(build_can_frame(0x01, b'\x01\x02\x03'))
    except OSError:
        print('Error sending CAN frame')

Running an example several times with too small delay between executions, could lead to this error:

OSError: [Errno 98] Address already in use

This is because the previous execution has left the socket in a TIME_WAIT state, and can’t be immediately reused.

There is a socket flag to set, in order to prevent this, socket.SO_REUSEADDR:

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))

the SO_REUSEADDR flag tells the kernel to reuse a local socket in TIME_WAIT state, without waiting for its natural timeout to expire.

See Also:

For an introduction to socket programming (in C), see the following papers:

- An Introductory 4.3BSD Interprocess Communication Tutorial, by Stuart Sechrest
- An Advanced 4.3BSD Interprocess Communication Tutorial, by Samuel J. Leffler et al,

both in the UNIX Programmer’s Manual, Supplementary Documents 1 (sections PS1:7 and PS1:8). The platform-specific reference material for the various socket-related system calls are also a valuable source of information on the details of socket semantics. For Unix, refer to the manual pages; for Windows, see the WinSock (or Winsock 2) specification. For IPv6-ready APIs, readers may want to refer to RFC 3493 titled Basic Socket Interface Extensions for IPv6.

## 18.2 ssl — TLS/SSL wrapper for socket objects

**Source code:** Lib/ssl.py

This module provides access to Transport Layer Security (often known as “Secure Sockets Layer”) encryption and peer authentication facilities for network sockets, both client-side and server-side. This module uses the OpenSSL library. It is available on all modern Unix systems, Windows, Mac OS X, and probably additional platforms, as long as OpenSSL is installed on that platform.

**Note:** Some behavior may be platform dependent, since calls are made to the operating system socket APIs. The installed version of OpenSSL may also cause variations in behavior.
This section documents the objects and functions in the `ssl` module; for more general information about TLS, SSL, and certificates, the reader is referred to the documents in the “See Also” section at the bottom.

This module provides a class, `ssl.SSLSocket`, which is derived from the `socket.socket` type, and provides a socket-like wrapper that also encrypts and decrypts the data going over the socket with SSL. It supports additional methods such as `getpeercert()`, which retrieves the certificate of the other side of the connection, and `cipher()`, which retrieves the cipher being used for the secure connection.

For more sophisticated applications, the `ssl.SSLContext` class helps manage settings and certificates, which can then be inherited by SSL sockets created through the `SSLContext.wrap_socket()` method.

### 18.2.1 Functions, Constants, and Exceptions

**exception ssl.SSLError**

Raised to signal an error from the underlying SSL implementation (currently provided by the OpenSSL library). This signifies some problem in the higher-level encryption and authentication layer that’s superimposed on the underlying network connection. This error is a subtype of `OSError`. The error code and message of `SSLError` instances are provided by the OpenSSL library. Changed in version 3.3: `SSLError` used to be a subtype of `socket.error`.

- **library**: A string mnemonic designating the OpenSSL submodule in which the error occurred, such as `SSL`, `PEM` or `X509`. The range of possible values depends on the OpenSSL version. New in version 3.3.

- **reason**: A string mnemonic designating the reason this error occurred, for example `CERTIFICATE_VERIFY_FAILED`. The range of possible values depends on the OpenSSL version. New in version 3.3.

**exception ssl.SSLZeroReturnError**

A subclass of `SSLError` raised when trying to read or write and the SSL connection has been closed cleanly. Note that this doesn’t mean that the underlying transport (read TCP) has been closed. New in version 3.3.

**exception ssl.SSLWantReadError**

A subclass of `SSLError` raised by a non-blocking SSL socket when trying to read or write data, but more data needs to be received on the underlying TCP transport before the request can be fulfilled. New in version 3.3.

**exception ssl.SSLWantWriteError**

A subclass of `SSLError` raised by a non-blocking SSL socket when trying to read or write data, but more data needs to be sent on the underlying TCP transport before the request can be fulfilled. New in version 3.3.

**exception ssl.SSLSyscallError**

A subclass of `SSLError` raised when a system error was encountered while trying to fulfill an operation on a SSL socket. Unfortunately, there is no easy way to inspect the original errno number. New in version 3.3.

**exception ssl.SSLEOFError**

A subclass of `SSLError` raised when the SSL connection has been terminated abruptly. Generally, you shouldn’t try to reuse the underlying transport when this error is encountered. New in version 3.3.

**exception ssl.CertificateError**

Raised to signal an error with a certificate (such as mismatching hostname). Certificate errors detected by OpenSSL, though, raise an `SSLError`.

18.2. `ssl` — TLS/SSL wrapper for socket objects
Socket creation

The following function allows for standalone socket creation. Starting from Python 3.2, it can be more flexible to use `sslContext.wrap_socket()` instead.

```python
def wrap_socket(sock, keyfile=None, certfile=None, server_side=False, cert_reqs=CERT_NONE, ssl_version={see docs}, ca_certs=None, do_handshake_on_connect=True, suppress_ragged_eofs=True, ciphers=None)
```

Takes an instance `sock` of `socket.socket`, and returns an instance of `ssl.SSLSocket`, a subtype of `socket.socket`, which wraps the underlying socket in an SSL context. For client-side sockets, the context construction is lazy; if the underlying socket isn’t connected yet, the context construction will be performed after `connect()` is called on the socket. For server-side sockets, if the socket has no remote peer, it is assumed to be a listening socket, and the server-side SSL wrapping is automatically performed on client connections accepted via the `accept()` method. `wrap_socket()` may raise `SSLError`.

The `keyfile` and `certfile` parameters specify optional files which contain a certificate to be used to identify the local side of the connection. See the discussion of Certificates for more information on how the certificate is stored in the `certfile`.

The parameter `server_side` is a boolean which identifies whether server-side or client-side behavior is desired from this socket.

The parameter `cert_reqs` specifies whether a certificate is required from the other side of the connection, and whether it will be validated if provided. It must be one of the three values `CERT_NONE` (certificates ignored), `CERT OPTIONAL` (not required, but validated if provided), or `CERT_REQUIRED` (required and validated). If the value of this parameter is not `CERT_NONE`, then the `ca_certs` parameter must point to a file of CA certificates.

The `ca_certs` file contains a set of concatenated “certification authority” certificates, which are used to validate certificates passed from the other end of the connection. See the discussion of Certificates for more information about how to arrange the certificates in this file.

The parameter `ssl_version` specifies which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server’s choice. Most of the versions are not interoperable with the other versions. If not specified, the default is `PROTOCOL_SSLv23`; it provides the most compatibility with other versions.

Here’s a table showing which versions in a client (down the side) can connect to which versions in a server (along the top):

<table>
<thead>
<tr>
<th>client / server</th>
<th>SSLv2</th>
<th>SSLv3</th>
<th>SSLv23</th>
<th>TLSv1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSLv2</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>SSLv3</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>SSLv23</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>TLSv1</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Note:** Which connections succeed will vary depending on the version of OpenSSL. For instance, in some older versions of OpenSSL (such as 0.9.7 on OS X 10.4), an SSLv2 client could not connect to an SSLv23 server. Another example: beginning with OpenSSL 1.0.0, an SSLv23 client will not actually attempt SSLv2 connections unless you explicitly enable SSLv2 ciphers; for example, you might specify "ALL" or "SSLv2" as the `ciphers` parameter to enable them.

The `ciphers` parameter sets the available ciphers for this SSL object. It should be a string in the OpenSSL cipher list format.

The parameter `do_handshake_on_connect` specifies whether to do the SSL handshake automatically after doing a `socket.connect()`, or whether the application program will call it explicitly, by invoking the `SSLSocket.do_handshake()` method. Calling `SSLSocket.do_handshake()` explicitly gives the program control over the blocking behavior of the socket I/O involved in the handshake.

The parameter `suppress_ragged_eofs` specifies how the `SSLSocket.recv()` method should signal unexpected EOF from the other end of the connection. If specified as `True` (the default), it returns a normal EOF (an empty bytes object) in response to unexpected EOF errors raised from the underlying
socket; if False, it will raise the exceptions back to the caller. Changed in version 3.2: New optional argument ciphers.

Random generation

**ssl.RAND_bytes**(num)
Returns num cryptographically strong pseudo-random bytes. Raises an SSLError if the PRNG has not been seeded with enough data or if the operation is not supported by the current RAND method. RAND_status() can be used to check the status of the PRNG and RAND_add() can be used to seed the PRNG.

Read the Wikipedia article, Cryptographically secure pseudorandom number generator (CSPRNG), to get the requirements of a cryptographically generator. New in version 3.3.

**ssl.RAND_pseudo_bytes**(num)
Returns (bytes, is_cryptographic): bytes are num pseudo-random bytes, is_cryptographic is True if the bytes generated are cryptographically strong. Raises an SSLError if the operation is not supported by the current RAND method.

Generated pseudo-random byte sequences will be unique if they are of sufficient length, but are not necessarily unpredictable. They can be used for non-cryptographic purposes and for certain purposes in cryptographic protocols, but usually not for key generation etc. New in version 3.3.

**ssl.RAND_status()**
Returns True if the SSL pseudo-random number generator has been seeded with ‘enough’ randomness, and False otherwise. You can use ssl.RAND_egd() and ssl.RAND_add() to increase the randomness of the pseudo-random number generator.

**ssl.RAND_egd**(path)
If you are running an entropy-gathering daemon (EGD) somewhere, and path is the pathname of a socket connection open to it, this will read 256 bytes of randomness from the socket, and add it to the SSL pseudo-random number generator to increase the security of generated secret keys. This is typically only necessary on systems without better sources of randomness.


**ssl.RAND_add**(bytes, entropy)
Mixes the given bytes into the SSL pseudo-random number generator. The parameter entropy (a float) is a lower bound on the entropy contained in string (so you can always use 0.0). See RFC 1750 for more information on sources of entropy.

Certificate handling

**ssl.match_hostname**(cert, hostname)
Verify that cert (in decoded format as returned by SSLSocket.getpeercert()) matches the given hostname. The rules applied are those for checking the identity of HTTPS servers as outlined in RFC 2818 and RFC 6125, except that IP addresses are not currently supported. In addition to HTTPS, this function should be suitable for checking the identity of servers in various SSL-based protocols such as FTPS, IMAPS, POPS and others.

CertificateError is raised on failure. On success, the function returns nothing:

```python
>>> cert = {'subject': ((('commonName', 'example.com'),),)}
>>> ssl.match_hostname(cert, "example.com")
>>> ssl.match_hostname(cert, "example.org")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/py3k/Lib/ssl.py", line 130, in match_hostname
ssl.CertificateError: hostname 'example.org' doesn't match 'example.com'
```
New in version 3.2. Changed in version 3.3.3: The function now follows RFC 6125, section 6.4.3 and does not match multiple wildcards (e.g. *.*.com or *a*.example.org) nor a wildcard inside an internationalized domain names (IDN) fragment. IDN A-labels such as www*.xn--pthon-kva.org are still supported, but x*.python.org no longer matches xn--tda.python.org.

```
ssl.cert_time_to_seconds(timestring)
```

Returns a floating-point value containing a normal seconds-after-the-epoch time value, given the time-string representing the “notBefore” or “notAfter” date from a certificate.

Here’s an example:

```python
>>> import ssl
>>> ssl.cert_time_to_seconds("May 9 00:00:00 2007 GMT")
1178694000.0
>>> import time
>>> time.ctime(ssl.cert_time_to_seconds("May 9 00:00:00 2007 GMT"))
'Wed May 9 00:00:00 2007'
```

```
ssl.get_server_certificate(addr, ssl_version=PROTOCOL_SSLv3, ca_certs=None)
```

Given the address `addr` of an SSL-protected server, as a `(hostname, port-number)` pair, fetches the server’s certificate, and returns it as a PEM-encoded string. If `ssl_version` is specified, uses that version of the SSL protocol to attempt to connect to the server. If `ca_certs` is specified, it should be a file containing a list of root certificates, the same format as used for the same parameter in `wrap_socket()`. The call will attempt to validate the server certificate against that set of root certificates, and will fail if the validation attempt fails. Changed in version 3.3: This function is now IPv6-compatible.

```
ssl.DER_cert_to_PEM_cert(DER_cert_bytes)
```

Given a certificate as a DER-encoded blob of bytes, returns a PEM-encoded string version of the same certificate.

```
ssl.PEM_cert_to_DER_cert(PEM_cert_string)
```

Given a certificate as an ASCII PEM string, returns a DER-encoded sequence of bytes for that same certificate.

**Constants**

```
ssl.CERT_NONE
```

Possible value for `SSLContext.verify_mode`, or the `cert_reqs` parameter to `wrap_socket()`. In this mode (the default), no certificates will be required from the other side of the socket connection. If a certificate is received from the other end, no attempt to validate it is made.

See the discussion of *Security considerations* below.

```
ssl.CERT_OPTIONAL
```

Possible value for `SSLContext.verify_mode`, or the `cert_reqs` parameter to `wrap_socket()`. In this mode, no certificates will be required from the other side of the socket connection; but if they are provided, validation will be attempted and an `SSLError` will be raised on failure.

Use of this setting requires a valid set of CA certificates to be passed, either to `SSLContext.load_verify_locations()` or as a value of the `ca_certs` parameter to `wrap_socket()`.

```
ssl.CERT_REQUIRED
```

Possible value for `SSLContext.verify_mode`, or the `cert_reqs` parameter to `wrap_socket()`. In this mode, certificates are required from the other side of the socket connection; an `SSLError` will be raised if no certificate is provided, or if its validation fails.

Use of this setting requires a valid set of CA certificates to be passed, either to `SSLContext.load_verify_locations()` or as a value of the `ca_certs` parameter to `wrap_socket()`.
ssl.PROTOCOL_SSLv2
Selects SSL version 2 as the channel encryption protocol.
This protocol is not available if OpenSSL is compiled with OPENSSL_NO_SSL2 flag.

**Warning:** SSL version 2 is insecure. Its use is highly discouraged.

ssl.PROTOCOL_SSLv23
Selects SSL version 2 or 3 as the channel encryption protocol. This is a setting to use with servers for maximum compatibility with the other end of an SSL connection, but it may cause the specific ciphers chosen for the encryption to be of fairly low quality.

ssl.PROTOCOL_SSLv3
Selects SSL version 3 as the channel encryption protocol. For clients, this is the maximally compatible SSL variant.

ssl.PROTOCOL_TLSv1
Selects TLS version 1 as the channel encryption protocol. This is the most modern version, and probably the best choice for maximum protection, if both sides can speak it.

ssl.OP_ALL
Enables workarounds for various bugs present in other SSL implementations. This option is set by default. It does not necessarily set the same flags as OpenSSL’s SSL_OP_ALL constant. New in version 3.2.

ssl.OP_NO_SSLv2
Prevents an SSLv2 connection. This option is only applicable in conjunction with PROTOCOL_SSLv23. It prevents the peers from choosing SSLv2 as the protocol version. New in version 3.2.

ssl.OP_NO_SSLv3
Prevents an SSLv3 connection. This option is only applicable in conjunction with PROTOCOL_SSLv23. It prevents the peers from choosing SSLv3 as the protocol version. New in version 3.2.

ssl.OP_NO_TLSv1
Prevents a TLSv1 connection. This option is only applicable in conjunction with PROTOCOL_SSLv23. It prevents the peers from choosing TLSv1 as the protocol version. New in version 3.2.

ssl.OP_CIPHER_SERVER_PREFERENCE
Use the server’s cipher ordering preference, rather than the client’s. This option has no effect on client sockets and SSLv2 server sockets. New in version 3.3.

ssl.OP_SINGLE_DH_USE
Prevents re-use of the same DH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets. New in version 3.3.

ssl.OP_SINGLE_ECDH_USE
Prevents re-use of the same ECDH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets. New in version 3.3.

ssl.OP_NO_COMPRESSION
Disable compression on the SSL channel. This is useful if the application protocol supports its own compression scheme.
This option is only available with OpenSSL 1.0.0 and later. New in version 3.3.

ssl.HAS_ECDH
Whether the OpenSSL library has built-in support for Elliptic Curve-based Diffie-Hellman key exchange. This should be true unless the feature was explicitly disabled by the distributor. New in version 3.3.

ssl.HAS_SNI
Whether the OpenSSL library has built-in support for the *Server Name Indication* extension to the SSLv3 and TLSv1 protocols (as defined in RFC 4366). When true, you can use the server_hostname argument to SSLContext.wrap_socket(). New in version 3.2.

18.2. ssl — TLS/SSL wrapper for socket objects
ssl.HAS_NPN
Whether the OpenSSL library has built-in support for Next Protocol Negotiation as described in the NPN
draft specification. When true, you can use the SSLContext.set_npn_protocols() method to
advertise which protocols you want to support. New in version 3.3.

ssl.CHANNEL_BINDING_TYPES
List of supported TLS channel binding types. Strings in this list can be used as arguments to
SSLSocket.get_channel_binding(). New in version 3.3.

ssl.OPENSSL_VERSION
The version string of the OpenSSL library loaded by the interpreter:

```python
>>> ssl.OPENSSL_VERSION
'OpenSSL 0.9.8k 25 Mar 2009'
```

New in version 3.2.

ssl.OPENSSL_VERSION_INFO
A tuple of five integers representing version information about the OpenSSL library:

```python
>>> ssl.OPENSSL_VERSION_INFO
(0, 9, 8, 11, 15)
```

New in version 3.2.

ssl.OPENSSL_VERSION_NUMBER
The raw version number of the OpenSSL library, as a single integer:

```python
>>> ssl.OPENSSL_VERSION_NUMBER
9470143
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x9080bf'
```

New in version 3.2.

## 18.2.2 SSL Sockets

SSL sockets provide the following methods of Socket Objects:

- accept()
- bind()
- close()
- connect()
- detach()
- fileno()
- getpeernamem, getsockname()
- getsockopt(), setsockopt()
- gettimeout(), settimeout(), setblocking()
- listen()
- makefile()
- recv(), recv_into() (but passing a non-zero flags argument is not allowed)
- send(), sendall() (with the same limitation)
- shutdown()
However, since the SSL (and TLS) protocol has its own framing atop of TCP, the SSL sockets abstraction can, in certain respects, diverge from the specification of normal, OS-level sockets. See especially the notes on non-blocking sockets.

SSL sockets also have the following additional methods and attributes:

**SSLSocket.**

- **do_handshake()**
  Perform the SSL setup handshake.

- **getpeercert (binary_form=False)**
  If there is no certificate for the peer on the other end of the connection, returns `None`.
  If the `binary_form` parameter is `False`, and a certificate was received from the peer, this method returns a `dict` instance. If the certificate was not validated, the dict is empty. If the certificate was validated, it returns a dict with several keys, amongst them `subject` (the principal for which the certificate was issued) and `issuer` (the principal issuing the certificate). If a certificate contains an instance of the Subject Alternative Name extension (see [RFC 3280](https://tools.ietf.org/html/rfc3280)), there will also be a `subjectAltName` key in the dictionary.

  The `subject` and `issuer` fields are tuples containing the sequence of relative distinguished names (RDNs) given in the certificate’s data structure for the respective fields, and each RDN is a sequence of name-value pairs. Here is a real-world example:

  ```python
  {'issuer': ((('countryName', 'IL'),),
              (('organizationName', 'StartCom Ltd.'),),
              (('organizationalUnitName',
                'Secure Digital Certificate Signing'),),
              (('commonName',
                'StartCom Class 2 Primary Intermediate Server CA'),)),
  'notAfter': 'Nov 22 08:15:19 2013 GMT',
  'notBefore': 'Nov 21 03:09:52 2011 GMT',
  'serialNumber': '95F0',
  'subject': (((('description', '571208-SLe2570eYfQ07Z'),),
               (('countryName', 'US'),),
               (('stateOrProvinceName', 'California'),),
               (('localityName', 'San Francisco'),),
               (('organizationName', 'Electronic Frontier Foundation, Inc.'),),
               (('commonName', '*.eff.org'),),
               (('emailAddress', 'hostmaster@eff.org'),)),
  'subjectAltName': (('DNS', '*.eff.org'), ('DNS', 'eff.org')),
  'version': 3}
  ```

  **Note:** To validate a certificate for a particular service, you can use the `match_hostname()` function.

  If the `binary_form` parameter is `True`, and a certificate was provided, this method returns the DER-encoded form of the entire certificate as a sequence of bytes, or `None` if the peer did not provide a certificate.

  Whether the peer provides a certificate depends on the SSL socket’s role:

  - for a client SSL socket, the server will always provide a certificate, regardless of whether validation was required;
  - for a server SSL socket, the client will only provide a certificate when requested by the server; therefore `getpeercert()` will return `None` if you used `CERT_NONE` (rather than `CERT_OPTIONAL` or `CERT_REQUIRED`).

  Changed in version 3.2: The returned dictionary includes additional items such as `issuer` and `notBefore`.

- **SSLSocket.**
  **cipher()**

  Returns a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used. If no connection has been established, returns `None`. 
SSL socket methods:

- **SSLSocket.compression()**
  Return the compression algorithm being used as a string, or None if the connection isn’t compressed.
  If the higher-level protocol supports its own compression mechanism, you can use OP_NO_COMPRESSION to disable SSL-level compression. New in version 3.3.

- **SSLSocket.get_channel_binding(cb_type=“tls-unique”)**
  Get channel binding data for current connection, as a bytes object. Returns None if not connected or the handshake has not been completed.
  The cb_type parameter allows selection of the desired channel binding type. Valid channel binding types are listed in the CHANNEL_BINDING_TYPES list. Currently only the ‘tls-unique’ channel binding, defined by RFC 5929, is supported. ValueError will be raised if an unsupported channel binding type is requested. New in version 3.3.

- **SSLSocket.selected_npn_protocol()**
  Returns the protocol that was selected during the TLS/SSL handshake. If SSLContext.set_npn_protocols() was not called, or if the other party does not support NPN, or if the handshake has not yet happened, this will return None. New in version 3.3.

- **SSLSocket.unwrap()**
  Performs the SSL shutdown handshake, which removes the TLS layer from the underlying socket, and returns the underlying socket object. This can be used to go from encrypted operation over a connection to unencrypted. The returned socket should always be used for further communication with the other side of the connection, rather than the original socket.

SSL socket context:

- **SSLContext.context**
  The SSLContext object this SSL socket is tied to. If the SSL socket was created using the top-level wrap_socket() function (rather than SSLContext.wrap_socket()), this is a custom context object created for this SSL socket. New in version 3.2.

### 18.2.3 SSL Contexts

New in version 3.2. An SSL context holds various data longer-lived than single SSL connections, such as SSL configuration options, certificate(s) and private key(s). It also manages a cache of SSL sessions for server-side sockets, in order to speed up repeated connections from the same clients.

```python
class ssl.SSLContext (protocol)
    Create a new SSL context. You must pass protocol which must be one of the PROTOCOL_* constants defined in this module. PROTOCOL_SSLv23 is recommended for maximum interoperability.
```

SSLContext objects have the following methods and attributes:

- **SSLContext.load_cert_chain(certfile, keyfile=None, password=None)**
  Load a private key and the corresponding certificate. The certfile string must be the path to a single file in PEM format containing the certificate as well as any number of CA certificates needed to establish the certificate’s authenticity. The keyfile string, if present, must point to a file containing the private key in. Otherwise the private key will be taken from certfile as well. See the discussion of Certificates for more information on how the certificate is stored in the certfile.
  
  The password argument may be a function to call to get the password for decrypting the private key. It will only be called if the private key is encrypted and a password is necessary. It will be called with no arguments, and it should return a string, bytes, or bytearray. If the return value is a string it will be encoded as UTF-8 before using it to decrypt the key. Alternatively a string, bytes, or bytearray value may be supplied directly as the password argument. It will be ignored if the private key is not encrypted and no password is needed.
  
  If the password argument is not specified and a password is required, OpenSSL’s built-in password prompting mechanism will be used to interactively prompt the user for a password.

  An SSLError is raised if the private key doesn’t match with the certificate. Changed in version 3.3: New optional argument password.
SSLContext.load_verify_locations( cafile=None, capath=None)
Load a set of "certification authority" (CA) certificates used to validate other peers’ certificates when verify_mode is other than CERT_NONE. At least one of cafile or capath must be specified.

The cafile string, if present, is the path to a file of concatenated CA certificates in PEM format. See the discussion of Certificates for more information about how to arrange the certificates in this file.

The capath string, if present, is the path to a directory containing several CA certificates in PEM format, following an OpenSSL specific layout.

SSLContext.set_default_verify_paths()
Load a set of default "certification authority" (CA) certificates from a filesystem path defined when building the OpenSSL library. Unfortunately, there’s no easy way to know whether this method succeeds: no error is returned if no certificates are to be found. When the OpenSSL library is provided as part of the operating system, though, it is likely to be configured properly.

SSLContext.set_ciphers(ciphers)
Set the available ciphers for sockets created with this context. It should be a string in the OpenSSL cipher list format. If no cipher can be selected (because compile-time options or other configuration forbids use of all the specified ciphers), an SSLError will be raised.

Note: when connected, the SSLSocket.cipher() method of SSL sockets will give the currently selected cipher.

SSLContext.set_npn_protocols(protocols)
Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of strings, like [‘http/1.1’, ‘spdy/2’], ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to the NPN draft specification. After a successful handshake, the SSLSocket.selected_npn_protocol() method will return the agreed-upon protocol.

This method will raise NotImplementedErorr if HAS_NPN is False. New in version 3.3.

SSLContext.load_dh_params(dhfile)
Load the key generation parameters for Diffie-Helman (DH) key exchange. Using DH key exchange improves forward secrecy at the expense of computational resources (both on the server and on the client). The dhfile parameter should be the path to a file containing DH parameters in PEM format.

This setting doesn’t apply to client sockets. You can also use the OP_SINGLE_DH_USE option to further improve security. New in version 3.3.

SSLContext.set_ecdh_curve(curve_name)
Set the curve name for Elliptic Curve-based Diffie-Hellman (ECDH) key exchange. ECDH is significantly faster than regular DH while arguably as secure. The curve_name parameter should be a string describing a well-known elliptic curve, for example prime256v1 for a widely supported curve.

This setting doesn’t apply to client sockets. You can also use the OP_SINGLE_ECDH_USE option to further improve security.

This method is not available if HAS_ECDH is False. New in version 3.3.

See Also:
SSL/TLS & Perfect Forward Secrecy Vincent Bernat.

SSLContext.wrap_socket(sock, server_side=False, do_handshake_on_connect=True, suppress_ragged_eofs=True, server_hostname=None)
Wrap an existing Python socket sock and return an SSLSocket object. The SSL socket is tied to the context, its settings and certificates. The parameters server_side, do_handshake_on_connect and suppress_ragged_eofs have the same meaning as in the top-level wrap_socket() function.

On client connections, the optional parameter server_hostname specifies the hostname of the service which we are connecting to. This allows a single server to host multiple SSL-based services with distinct certificates, quite similarly to HTTP virtual hosts. Specifying server_hostname will raise a ValueError if the
OpenSSL library doesn’t have support for it (that is, if HAS_SNI is False). Specifying server_hostname will also raise a ValueError if server_side is true.

SSLContext.session_stats()
Get statistics about the SSL sessions created or managed by this context. A dictionary is returned which maps the names of each piece of information to their numeric values. For example, here is the total number of hits and misses in the session cache since the context was created:

```python
>>> stats = context.session_stats()
0, 0
```

SSLContext.options
An integer representing the set of SSL options enabled on this context. The default value is OP_ALL, but you can specify other options such as OP_NO_SSLv2 by ORing them together.

Note: With versions of OpenSSL older than 0.9.8m, it is only possible to set options, not to clear them. Attempting to clear an option (by resetting the corresponding bits) will raise a ValueError.

SSLContext.protocol
The protocol version chosen when constructing the context. This attribute is read-only.

SSLContext.verify_mode
Whether to try to verify other peers’ certificates and how to behave if verification fails. This attribute must be one of CERT_NONE, CERT_OPTIONAL or CERT_REQUIRED.

## 18.2.4 Certificates

Certificates in general are part of a public-key / private-key system. In this system, each principal, (which may be a machine, or a person, or an organization) is assigned a unique two-part encryption key. One part of the key is public, and is called the public key; the other part is kept secret, and is called the private key. The two parts are related, in that if you encrypt a message with one of the parts, you can decrypt it with the other part, and only with the other part.

A certificate contains information about two principals. It contains the name of a subject, and the subject’s public key. It also contains a statement by a second principal, the issuer, that the subject is who he claims to be, and that this is indeed the subject’s public key. The issuer’s statement is signed with the issuer’s private key, which only the issuer knows. However, anyone can verify the issuer’s statement by finding the issuer’s public key, decrypting the statement with it, and comparing it to the other information in the certificate. The certificate also contains information about the time period over which it is valid. This is expressed as two fields, called “notBefore” and “notAfter”.

In the Python use of certificates, a client or server can use a certificate to prove who they are. The other side of a network connection can also be required to produce a certificate, and that certificate can be validated to the satisfaction of the client or server that requires such validation. The connection attempt can be set to raise an exception if the validation fails. Validation is done automatically, by the underlying OpenSSL framework; the application need not concern itself with its mechanics. But the application does usually need to provide sets of certificates to allow this process to take place.

Python uses files to contain certificates. They should be formatted as “PEM” (see RFC 1422), which is a base-64 encoded form wrapped with a header line and a footer line:

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```
Certificate chains

The Python files which contain certificates can contain a sequence of certificates, sometimes called a certificate chain. This chain should start with the specific certificate for the principal who “is” the client or server, and then the certificate for the issuer of that certificate, and then the certificate for the issuer of that certificate, and so on up the chain till you get to a certificate which is self-signed, that is, a certificate which has the same subject and issuer, sometimes called a root certificate. The certificates should just be concatenated together in the certificate file. For example, suppose we had a three certificate chain, from our server certificate to the certificate of the certification authority that signed our server certificate, to the root certificate of the agency which issued the certification authority’s certificate:

```
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the root certificate for the CA’s issuer)...
-----END CERTIFICATE-----
```

CA certificates

If you are going to require validation of the other side of the connection’s certificate, you need to provide a “CA certs” file, filled with the certificate chains for each issuer you are willing to trust. Again, this file just contains these chains concatenated together. For validation, Python will use the first chain it finds in the file which matches. Some “standard” root certificates are available from various certification authorities: CACert.org, Thawte, Verisign, Positive SSL (used by python.org), Equifax and GeoTrust.

In general, if you are using SSL3 or TLS1, you don’t need to put the full chain in your “CA certs” file; you only need the root certificates, and the remote peer is supposed to furnish the other certificates necessary to chain from its certificate to a root certificate. See RFC 4158 for more discussion of the way in which certification chains can be built.

Combined key and certificate

Often the private key is stored in the same file as the certificate; in this case, only the certfile parameter to SSLContext.load_cert_chain() and wrap_socket() needs to be passed. If the private key is stored with the certificate, it should come before the first certificate in the certificate chain:

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Self-signed certificates

If you are going to create a server that provides SSL-encrypted connection services, you will need to acquire a certificate for that service. There are many ways of acquiring appropriate certificates, such as buying one from a certification authority. Another common practice is to generate a self-signed certificate. The simplest way to do this is with the OpenSSL package, using something like the following:

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
Generating a 1024 bit RSA private key
........+++++
.............................+++++
```

18.2. ssl — TLS/SSL wrapper for socket objects
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be incorporated into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN. There are quite a few fields but you can leave some blank For some fields there will be a default value, If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization.com
Email Address []:ops@myserver.mygroup.myorganization.com
%
The disadvantage of a self-signed certificate is that it is its own root certificate, and no one else will have it in their cache of known (and trusted) root certificates.

18.2.5 Examples

Testing for SSL support

To test for the presence of SSL support in a Python installation, user code should use the following idiom:

try:
    import ssl
except ImportError:
    pass
else:
    ...
    # do something that requires SSL support

Client-side operation

This example connects to an SSL server and prints the server’s certificate:

```python
import socket, ssl, pprint

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# require a certificate from the server
ssl_sock = ssl.wrap_socket(s,
    ca_certs="/etc/ca_certs_file",
    cert_reqs=ssl.CERT_REQUIRED)
ssl_sock.connect(('www.verisign.com', 443))

pprint.pprint(ssl_sock.getpeercert())
# note that closing the SSLSocket will also close the underlying socket
ssl_sock.close()
```

As of January 6, 2012, the certificate printed by this program looks like this:

```python
{'issuer': [({'countryName': 'US'},),
    ({'organizationName': 'VeriSign, Inc.'},),
    ({'organizationalUnitName': 'VeriSign Trust Network'},),
    ({'organizationalUnitName': 'Terms of use at https://www.verisign.com/rpa (c)06'},),
    ({'commonName',
```
{'VeriSign Class 3 Extended Validation SSL SGC CA'),

'notAfter': 'May 25 23:59:59 2012 GMT',
'notBefore': 'May 26 00:00:00 2010 GMT',
'serialNumber': '53D2BEBF924A7245E83CA0E46CAA2477',
'subject': (((('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
((('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
(('businessCategory', 'V1.0, Clause 5.(b)'),),
(('serialNumber', '2497886'),),
(('countryName', 'US'),),
(('postalCode', '94043'),),
(('stateOrProvinceName', 'California'),),
(('localityName', 'Mountain View'),),
(('streetAddress', '487 East Middlefield Road'),),
(('organizationName', 'VeriSign, Inc.'),),
(('organizationalUnitName', 'Production Security Services'),),
(('commonName', 'www.verisign.com'),),

'subjectAltName': ((('DNS', 'www.verisign.com'),
    ('DNS', 'verisign.com'),
    ('DNS', 'www.verisign.net'),
    ('DNS', 'verisign.net'),
    ('DNS', 'www.verisign.mobi'),
    ('DNS', 'verisign.mobi'),
    ('DNS', 'www.verisign.eu'),
    ('DNS', 'verisign.eu')),

'version': 3}

This other example first creates an SSL context, instructs it to verify certificates sent by peers, and feeds it a set of recognized certificate authorities (CA):

```python
>>> context = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> context.verify_mode = ssl.CERT_REQUIRED
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(it is assumed your operating system places a bundle of all CA certificates in /etc/ssl/certs/ca-bundle.crt; if not, you’ll get an error and have to adjust the location)

When you use the context to connect to a server, CERT_REQUIRED validates the server certificate: it ensures that the server certificate was signed with one of the CA certificates, and checks the signature for correctness:

```python
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET))
>>> conn.connect(('linuxfr.org', 443))
```

You should then fetch the certificate and check its fields for conformity:

```python
>>> cert = conn.getpeer-cert()  
>>> ssl.match_hostname(cert, "linuxfr.org")
```

Visual inspection shows that the certificate does identify the desired service (that is, the HTTPS host linuxfr.org):

```python
>>> pprint.pprint(cert)
{'issuer': (((('organizationName', 'CAcert Inc.'),),
    (('organizationalUnitName', 'http://www.CAcert.org'),),
    (('commonName', 'CAcert Class 3 Root'),)),

'notAfter': 'Jun 7 21:02:24 2013 GMT',
'notBefore': 'Jun 8 21:02:24 2011 GMT',
'serialNumber': 'D3E9',
'subject': (((('commonName', 'linuxfr.org'),)),

'subjectAltName': ((('DNS', 'linuxfr.org'),
    ('othername', '<unsupported>'),
    ('DNS', 'linuxfr.org'),
    ('othername', '<unsupported>'),
    ('DNS', 'dev.linuxfr.org'),
    ('DNS', 'linuxfr.org'),
```
Now that you are assured of its authenticity, you can proceed to talk with the server:

```python
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r
"
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 302 Found',
 b'Date: Sun, 16 May 2010 13:43:28 GMT',
 b'Server: Apache/2.2',
 b'Location: https://linuxfr.org/pub/',
 b'Vary: Accept-Encoding',
 b'Connection: close',
 b'Content-Type: text/html; charset=iso-8859-1',
 b'',
 b'']
```

See the discussion of Security considerations below.

### Server-side operation

For server operation, typically you’ll need to have a server certificate, and private key, each in a file. You’ll first create a context holding the key and the certificate, so that clients can check your authenticity. Then you’ll open a socket, bind it to a port, call `listen()` on it, and start waiting for clients to connect:

```python
import socket, ssl

certfile = "mycertfile"
certfile = "mycertfile",
context = ssl.SSLContext(ssl.PROTOCOL_TLSv1)
context.load_cert_chain(certfile=certfile, keyfile=certfile)
bindsocket = socket.socket()
bindsocket.bind(('myaddr.mydomain.com', 10023))
bindsocket.listen(5)

while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

Then you’ll read data from the `connstream` and do something with it till you are finished with the client (or the client is finished with you):

```python
def deal_with_client(connstream):
data = connstream.recv(1024)
    # empty data means the client is finished with us
while data:
    if not do_something(connstream, data):
        # we’ll assume do_something returns False
```
18.2.6 Notes on non-blocking sockets

When working with non-blocking sockets, there are several things you need to be aware of:

- Calling `select()` tells you that the OS-level socket can be read from (or written to), but it does not imply that there is sufficient data at the upper SSL layer. For example, only part of an SSL frame might have arrived. Therefore, you must be ready to handle `SSLSocket.recv()` and `SSLSocket.send()` failures, and retry after another call to `select()`.

  (of course, similar provisions apply when using other primitives such as `poll()`)

- The SSL handshake itself will be non-blocking: the `SSLSocket.do_handshake()` method has to be retried until it returns successfully. Here is a synopsis using `select()` to wait for the socket’s readiness:

  ```python
  while True:
      try:
          sock.do_handshake()
          break
      except ssl.SSLWantReadError:
          select.select([sock], [], [])
      except ssl.SSLWantWriteError:
          select.select([], [sock], [])
  ```

18.2.7 Security considerations

Verifying certificates

`CERT_NONE` is the default. Since it does not authenticate the other peer, it can be insecure, especially in client mode where most of time you would like to ensure the authenticity of the server you’re talking to. Therefore, when in client mode, it is highly recommended to use `CERT_REQUIRED`. However, it is in itself not sufficient; you also have to check that the server certificate, which can be obtained by calling `SSLSocket.getpeercert()`, matches the desired service. For many protocols and applications, the service can be identified by the hostname; in this case, the `match_hostname()` function can be used.

In server mode, if you want to authenticate your clients using the SSL layer (rather than using a higher-level authentication mechanism), you’ll also have to specify `CERT_REQUIRED` and similarly check the client certificate.

```
Note: In client mode, `CERT_OPTIONAL` and `CERT_REQUIRED` are equivalent unless anonymous ciphers are enabled (they are disabled by default).
```

Protocol versions

SSL version 2 is considered insecure and is therefore dangerous to use. If you want maximum compatibility between clients and servers, it is recommended to use `PROTOCOL_SSLv23` as the protocol version and then disable SSLv2 explicitly using the `SSLContext.options` attribute:

```python
context = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
context.options |= ssl.OP_NO_SSLv2
```

The SSL context created above will allow SSLv3 and TLSv1 connections, but not SSLv2.

18.2. `ssl` — TLS/SSL wrapper for socket objects

703
Cipher selection

If you have advanced security requirements, fine-tuning of the ciphers enabled when negotiating an SSL session is possible through the `SSLContext.set_ciphers()` method. Starting from Python 3.2.3, the ssl module disables certain weak ciphers by default, but you may want to further restrict the cipher choice. For example:

```python
context = ssl.SSLContext(ssl.PROTOCOL_TLSv1)
context.set_ciphers('HIGH:!aNULL:!eNULL')
```

The `!aNULL:!eNULL` part of the cipher spec is necessary to disable ciphers which don’t provide both encryption and authentication. Be sure to read OpenSSL’s documentation about the cipher list format. If you want to check which ciphers are enabled by a given cipher list, use the `openssl ciphers` command on your system.

See Also:

- [Class `socket.socket` Documentation of underlying `socket` class](#)
- [SSL/TLS Strong Encryption: An Introduction](#)
- [Intro from the Apache webserver documentation](#)
- [RFC 1422: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management](#)
- [Steve Kent](#)
- [RFC 1750: Randomness Recommendations for Security](#)
- [D. Eastlake et. al.](#)
- [RFC 3280: Internet X.509 Public Key Infrastructure Certificate and CRL Profile](#)
- [Housley et. al.](#)
- [RFC 4366: Transport Layer Security (TLS) Extensions](#)
- [Blake-Wilson et. al.](#)

18.3 asyncore — Asynchronous socket handler

Source code: Lib/asyncore.py

This module provides the basic infrastructure for writing asynchronous socket service clients and servers.

There are only two ways to have a program on a single processor do “more than one thing at a time.” Multi-threaded programming is the simplest and most popular way to do it, but there is another very different technique, that lets you have nearly all the advantages of multi-threading, without actually using multiple threads. It’s really only practical if your program is largely I/O bound. If your program is processor bound, then pre-emptive scheduled threads are probably what you really need. Network servers are rarely processor bound, however.

If your operating system supports the `select()` system call in its I/O library (and nearly all do), then you can use it to juggle multiple communication channels at once; doing other work while your I/O is taking place in the “background.” Although this strategy can seem strange and complex, especially at first, it is in many ways easier to understand and control than multi-threaded programming. The `asyncore` module solves many of the difficult problems for you, making the task of building sophisticated high-performance network servers and clients a snap. For “conversational” applications and protocols the companion `asynchat` module is invaluable.

The basic idea behind both modules is to create one or more network channels, instances of class `asyncore.dispatcher` and `asynchat.async_chat`. Creating the channels adds them to a global map, used by the `loop()` function if you do not provide it with your own `map`.

Once the initial channel(s) is(are) created, calling the `loop()` function activates channel service, which continues until the last channel (including any that have been added to the map during asynchronous service) is closed.

```python
asyncore.loop([timeout[, use_poll[, map[, count]]]])
```

Enter a polling loop that terminates after count passes or all open channels have been closed. All arguments are optional. The `count` parameter defaults to None, resulting in the loop terminating only when all channels have been closed. The `timeout` argument sets the timeout parameter for the appropriate `select()` or `poll()` call, measured in seconds; the default is 30 seconds. The `use_poll` parameter, if true, indicates that `poll()` should be used in preference to `select()` (the default is False).

The `map` parameter is a dictionary whose items are the channels to watch. As channels are closed they are deleted from their map. If `map` is omitted, a global map is used. Channels (instances of
asyncore.dispatcher, asynchat.async_chat and subclasses thereof) can freely be mixed in the map.

class asyncore.dispatcher

The dispatcher class is a thin wrapper around a low-level socket object. To make it more useful, it has a few methods for event-handling which are called from the asynchronous loop. Otherwise, it can be treated as a normal non-blocking socket object.

The firing of low-level events at certain times or in certain connection states tells the asynchronous loop that certain higher-level events have taken place. For example, if we have asked for a socket to connect to another host, we know that the connection has been made when the socket becomes writable for the first time (at this point you know that you may write to it with the expectation of success). The implied higher-level events are:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle_connect()</td>
<td>Implied by the first read or write event</td>
</tr>
<tr>
<td>handle_close()</td>
<td>Implied by a read event with no data available</td>
</tr>
<tr>
<td>handle_accepted()</td>
<td>Implied by a read event on a listening socket</td>
</tr>
</tbody>
</table>

During asynchronous processing, each mapped channel’s readable() and writable() methods are used to determine whether the channel’s socket should be added to the list of channels select()ed or poll()ed for read and write events.

Thus, the set of channel events is larger than the basic socket events. The full set of methods that can be overridden in your subclass follows:

handle_read()

Called when the asynchronous loop detects that a read() call on the channel’s socket will succeed.

handle_write()

Called when the asynchronous loop detects that a writable socket can be written. Often this method will implement the necessary buffering for performance. For example:

```python
def handle_write(self):
    sent = self.send(self.buffer)
    self.buffer = self.buffer[sent:]
```

handle_expt()

Called when there is out of band (OOB) data for a socket connection. This will almost never happen, as OOB is tenuously supported and rarely used.

handle_connect()

Called when the active opener’s socket actually makes a connection. Might send a “welcome” banner, or initiate a protocol negotiation with the remote endpoint, for example.

handle_close()

Called when the socket is closed.

handle_error()

Called when an exception is raised and not otherwise handled. The default version prints a condensed traceback.

handle_accept()

Called on listening channels (passive openers) when a connection can be established with a new remote endpoint that has issued a connect() call for the local endpoint. Deprecated in version 3.2; use handle_accepted() instead. Deprecated since version 3.2.

handle_accepted(sock, addr)

Called on listening channels (passive openers) when a connection has been established with a new remote endpoint that has issued a connect() call for the local endpoint. sock is a new socket object usable to send and receive data on the connection, and addr is the address bound to the socket on the other end of the connection. New in version 3.2.

readable()

Called each time around the asynchronous loop to determine whether a channel’s socket should be
added to the list on which read events can occur. The default method simply returns True, indicating that by default, all channels will be interested in read events.

**writable()**

Called each time around the asynchronous loop to determine whether a channel’s socket should be added to the list on which write events can occur. The default method simply returns True, indicating that by default, all channels will be interested in write events.

In addition, each channel delegates or extends many of the socket methods. Most of these are nearly identical to their socket partners.

**create_socket (family=socket.AF_INET, type=socket.SOCK_STREAM)**

This is identical to the creation of a normal socket, and will use the same options for creation. Refer to the socket documentation for information on creating sockets. Changed in version 3.3: family and type arguments can be omitted.

**connect (address)**

As with the normal socket object, address is a tuple with the first element the host to connect to, and the second the port number.

**send (data)**

Send data to the remote end-point of the socket.

**recv (buffer_size)**

Read at most buffer_size bytes from the socket’s remote end-point. An empty string implies that the channel has been closed from the other end.

**listen (backlog)**

Listen for connections made to the socket. The backlog argument specifies the maximum number of queued connections and should be at least 1; the maximum value is system-dependent (usually 5).

**bind (address)**

Bind the socket to address. The socket must not already be bound. (The format of address depends on the address family — refer to the socket documentation for more information.) To mark the socket as re-usable (setting the SO_REUSEADDR option), call the dispatcher object’s set_reuse_addr() method.

**accept ()**

Accept a connection. The socket must be bound to an address and listening for connections. The return value can be either None or a pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection. When None is returned it means the connection didn’t take place, in which case the server should just ignore this event and keep listening for further incoming connections.

**close ()**

Close the socket. All future operations on the socket object will fail. The remote end-point will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

**class asyncore.dispatcher_with_send**

A dispatcher subclass which adds simple buffered output capability, useful for simple clients. For more sophisticated usage use asynchat.async_chat.

**class asyncore.file_dispatcher**

A file_dispatcher takes a file descriptor or file object along with an optional map argument and wraps it for use with the poll() or loop() functions. If provided a file object or anything with a fileno() method, that method will be called and passed to the file_wrapper constructor. Availability: UNIX.

**class asyncore.file_wrapper**

A file_wrapper takes an integer file descriptor and calls os.dup() to duplicate the handle so that the original handle may be closed independently of the file_wrapper. This class implements sufficient methods to emulate a socket for use by the file_dispatcher class. Availability: UNIX.
18.3.1 asyncore Example basic HTTP client

Here is a very basic HTTP client that uses the `dispatcher` class to implement its socket handling:

```python
import asyncore

class HTTPClient(asyncore.dispatcher):
    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.connect( (host, 80) )
        self.buffer = bytes('GET %s HTTP/1.0\r\nHost: %s\r\n\n' % (path, host), 'ascii')

    def handle_connect(self):
        pass

    def handle_close(self):
        self.close()

    def handle_read(self):
        print(self.recv(8192))

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]

client = HTTPClient('www.python.org', '/')
asyncore.loop()
```

18.3.2 asyncore Example basic echo server

Here is a basic echo server that uses the `dispatcher` class to accept connections and dispatches the incoming connections to a handler:

```python
import asyncore

class EchoHandler(asyncore.dispatcher_with_send):
    def handle_read(self):
        data = self.recv(8192)
        if data:
            self.send(data)

class EchoServer(asyncore.dispatcher):
    def __init__(self, host, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.set_reuse_addr()
        self.bind((host, port))
        self.listen(5)
```
def handle_accepted(self, sock, addr):
    print('Incoming connection from %s' % repr(addr))
handler = EchoHandler(sock)

server = EchoServer('localhost', 8080)
asyncore.loop()

18.4 asynchat — Asynchronous socket command/response handler

Source code: Lib/asynchat.py

This module builds on the asyncore infrastructure, simplifying asynchronous clients and servers and making it easier to handle protocols whose elements are terminated by arbitrary strings, or are of variable length. asynchat defines the abstract class async_chat that you subclass, providing implementations of the collect_incoming_data() and found_terminator() methods. It uses the same asynchronous loop as asyncore, and the two types of channel, asyncore.dispatcher and asynchat.async_chat, can freely be mixed in the channel map. Typically an asyncore.dispatcher server channel generates new asynchat.async_chat channel objects as it receives incoming connection requests.

class asynchat.async_chat
This class is an abstract subclass of asyncore.dispatcher. To make practical use of the code you must subclass async_chat, providing meaningful collect_incoming_data() and found_terminator() methods. The asyncore.dispatcher methods can be used, although not all make sense in a message/response context.

Like asyncore.dispatcher, async_chat defines a set of events that are generated by an analysis of socket conditions after a select() call. Once the polling loop has been started the async_chat object’s methods are called by the event-processing framework with no action on the part of the programmer.

Two class attributes can be modified, to improve performance, or possibly even to conserve memory.

ac_in_buffer_size
The asynchronous input buffer size (default 4096).

ac_out_buffer_size
The asynchronous output buffer size (default 4096).

Unlike asyncore.dispatcher, async_chat allows you to define a first-in-first-out queue (fifo) of producers. A producer need have only one method, more(), which should return data to be transmitted on the channel. The producer indicates exhaustion (i.e. that it contains no more data) by having its more() method return the empty string. At this point the async_chat object removes the producer from the fifo and starts using the next producer, if any. When the producer fifo is empty the handle_write() method does nothing. You use the channel object’s set_terminator() method to describe how to recognize the end of, or an important breakpoint in, an incoming transmission from the remote endpoint.

To build a functioning async_chat subclass your input methods collect_incoming_data() and found_terminator() must handle the data that the channel receives asynchronously. The methods are described below.

async_chat.close_when_done()
Pushes a None on to the producer fifo. When this producer is popped off the fifo it causes the channel to be closed.

async_chat.collect_incoming_data(data)
Called with data holding an arbitrary amount of received data. The default method, which must be overridden, raises a NotImplemented exception.
async_chat.delete_buffers()
In emergencies this method will discard any data held in the input and/or output buffers and the producer fifo.

async_chat.found_terminator()
Called when the incoming data stream matches the termination condition set by set_terminator(). The default method, which must be overridden, raises a NotImplementedError exception. The buffered input data should be available via an instance attribute.

async_chat.get_terminator()
Returns the current terminator for the channel.

async_chat.push(data)
Pushes data on to the channel’s fifo to ensure its transmission. This is all you need to do to have the channel write the data out to the network, although it is possible to use your own producers in more complex schemes to implement encryption and chunking, for example.

async_chat.push_with_producer(producer)
Takes a producer object and adds it to the producer fifo associated with the channel. When all currently-pushed producers have been exhausted the channel will consume this producer’s data by calling its more() method and send the data to the remote endpoint.

async_chat.set_terminator(term)
Sets the terminating condition to be recognized on the channel. term may be any of three types of value, corresponding to three different ways to handle incoming protocol data.

<table>
<thead>
<tr>
<th>term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Will call found_terminator() when the string is found in the input stream</td>
</tr>
<tr>
<td>integer</td>
<td>Will call found_terminator() when the indicated number of characters have been received</td>
</tr>
<tr>
<td>None</td>
<td>The channel continues to collect data forever</td>
</tr>
</tbody>
</table>

Note that any data following the terminator will be available for reading by the channel after found_terminator() is called.

### 18.4.1 asynchat - Auxiliary Classes

class asynchat.fifo(list=None)
A fifo holding data which has been pushed by the application but not yet popped for writing to the channel. A fifo is a list used to hold data and/or producers until they are required. If the list argument is provided then it should contain producers or data items to be written to the channel.

is_empty()
Returns True if and only if the fifo is empty.

first()
Returns the least-recently push()ed item from the fifo.

push(data)
Adds the given data (which may be a string or a producer object) to the producer fifo.

pop()
If the fifo is not empty, returns True, first(), deleting the popped item. Returns False, None for an empty fifo.

### 18.4.2 asynchat Example

The following partial example shows how HTTP requests can be read with asyn_chat. A web server might create an http_request_handler object for each incoming client connection. Notice that initially the channel terminator is set to match the blank line at the end of the HTTP headers, and a flag indicates that the headers are being read.
Once the headers have been read, if the request is of type POST (indicating that further data are present in the input stream) then the \texttt{Content-Length:} header is used to set a numeric terminator to read the right amount of data from the channel.

The \texttt{handle\_request()} method is called once all relevant input has been marshalled, after setting the channel terminator to \texttt{None} to ensure that any extraneous data sent by the web client are ignored.

```python
import asynchat

class http\_request\_handler(asynchat.async\_chat):
    def \_init\_(self, sock, addr, sessions, log):
        asynchat.async\_chat.\_init\_(self, sock=sock)
        self.addr = addr
        self.sessions = sessions
        self.ibuffer = []
        self.obuffer = b"
        self.set\_terminator(b"\r\n\r\n")
        self.reading\_headers = True
        self.handling = False
        self.cgi\_data = None
        self.log = log

    def collect\_incoming\_data(self, data):
        """Buffer the data""
        self.ibuffer.append(data)

    def found\_terminator(self):
        if self.reading\_headers:
            self.reading\_headers = False
            self.parse\_headers("".join(self.ibuffer))
            self.ibuffer = []
        if self.op.upper() == b"POST":
            clen = self.headers\_get\_header("content\_length")
            self.set\_terminator(int(clen))
        else:
            self.handling = True
            self.set\_terminator(None)
            self.handle\_request()
        elif not self.handling:
            self.set\_terminator(None) # browsers sometimes over-send
            self.cgi\_data = parse(self.headers, b"".join(self.ibuffer))
            self.handling = True
            self.ibuffer = []
            self.handle\_request()
```

### 18.5 \texttt{signal} — Set handlers for asynchronous events

This module provides mechanisms to use signal handlers in Python.

#### 18.5.1 General rules

The \texttt{signal.signal()} function allows to define custom handlers to be executed when a signal is received. A small number of default handlers are installed: SIGPIPE is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions) and SIGINT is translated into a \texttt{KeyboardInterrupt} exception.
A handler for a particular signal, once set, remains installed until it is explicitly reset (Python emulates the BSD style interface regardless of the underlying implementation), with the exception of the handler for SIGCHLD, which follows the underlying implementation.

There is no way to “block” signals temporarily from critical sections (since this is not supported by all Unix flavors).

**Execution of Python signal handlers**

A Python signal handler does not get executed inside the low-level (C) signal handler. Instead, the low-level signal handler sets a flag which tells the virtual machine to execute the corresponding Python signal handler at a later point (for example at the next bytecode instruction). This has consequences:

- It makes little sense to catch synchronous errors like SIGFPE or SIGSEGV that are caused by an invalid operation in C code. Python will return from the signal handler to the C code, which is likely to raise the same signal again, causing Python to apparently hang. From Python 3.3 onwards, you can use the faulthandler module to report on synchronous errors.
- A long-running calculation implemented purely in C (such as regular expression matching on a large body of text) may run uninterrupted for an arbitrary amount of time, regardless of any signals received. The Python signal handlers will be called when the calculation finishes.

**Signals and threads**

Python signal handlers are always executed in the main Python thread, even if the signal was received in another thread. This means that signals can’t be used as a means of inter-thread communication. You can use the synchronization primitives from the threading module instead.

Besides, only the main thread is allowed to set a new signal handler.

18.5.2 Module contents

The variables defined in the signal module are:

**signal.SIG_DFL**
This is one of two standard signal handling options; it will simply perform the default function for the signal. For example, on most systems the default action for SIGQUIT is to dump core and exit, while the default action for SIGCHLD is to simply ignore it.

**signal.SIG_IGN**
This is another standard signal handler, which will simply ignore the given signal.

**SIG***
All the signal numbers are defined symbolically. For example, the hangup signal is defined as signal.SIGHUP; the variable names are identical to the names used in C programs, as found in <signal.h>. The Unix man page for ‘signal()’ lists the existing signals (on some systems this is signal(2), on others the list is in signal(7)). Note that not all systems define the same set of signal names; only those names defined by the system are defined by this module.

**signal.CTRL_C_EVENT**
The signal corresponding to the CTRL+C keystroke event. This signal can only be used with os.kill().


**signal.CTRL_BREAK_EVENT**
The signal corresponding to the CTRL+BREAK keystroke event. This signal can only be used with os.kill().


**signal.NSIG**
One more than the number of the highest signal number.
signal.ITUER_REAL
    Decrements interval timer in real time, and delivers SIGALRM upon expiration.

signal.ITUIM_VIRTUAL
    Decrements interval timer only when the process is executing, and delivers SIGVTALRM upon expiration.

signal.ITUIM_PROF
    Decrements interval timer both when the process executes and when the system is executing on behalf of the process. Coupled with ITIMER_VIRTUAL, this timer is usually used to profile the time spent by the application in user and kernel space. SIGPROF is delivered upon expiration.

signal.SIG_BLOCK
    A possible value for the how parameter to pthread_sigmask() indicating that signals are to be blocked. New in version 3.3.

signal.SIG_UNBLOCK
    A possible value for the how parameter to pthread_sigmask() indicating that signals are to be unblocked. New in version 3.3.

signal.SIG_SETMASK
    A possible value for the how parameter to pthread_sigmask() indicating that the signal mask is to be replaced. New in version 3.3.

The signal module defines one exception:

exception signal.ITimerError
    Raised to signal an error from the underlying setitimer() or getitimer() implementation. Expect this error if an invalid interval timer or a negative time is passed to setitimer(). This error is a subtype of OSError. New in version 3.3: This error used to be a subtype of IOError, which is now an alias of OSError.

The signal module defines the following functions:

signal.alarm(time)
    If time is non-zero, this function requests that a SIGALRM signal be sent to the process in time seconds.
    Any previously scheduled alarm is canceled (only one alarm can be scheduled at any time). The returned value is then the number of seconds before any previously set alarm was to have been delivered. If time is zero, no alarm is scheduled, and any scheduled alarm is canceled. If the return value is zero, no alarm is currently scheduled. (See the Unix man page alarm(2).) Availability: Unix.

signal.getsignal(signalnum)
    Return the current signal handler for the signal signalnum. The returned value may be a callable Python object, or one of the special values signal.SIG_IGN, signal.SIG_DFL or None. Here, signal.SIG_IGN means that the signal was previously ignored, signal.SIG_DFL means that the default way of handling the signal was previously in use, and None means that the previous signal handler was not installed from Python.

signal.pause()
    Cause the process to sleep until a signal is received; the appropriate handler will then be called. Returns nothing. Not on Windows. (See the Unix man page signal(2).)

    See also sigwait(), sigwaitinfo(), sigtimedwait() and sigpending().

signal.pthread_kill(thread_id, signum)
    Send the signal signum to the thread thread_id, another thread in the same process as the caller. The target thread can be executing any code (Python or not). However, if the target thread is executing the Python interpreter, the Python signal handlers will be executed by the main thread. Therefore, the only point of sending a signal to a particular Python thread would be to force a running system call to fail with InterruptedError.

    Use threading.get_ident() or the ident attribute of threading.Thread objects to get a suitable value for thread_id.

    If signum is 0, then no signal is sent, but error checking is still performed; this can be used to check if the target thread is still running.
The Python Library Reference, Release 3.3.3

Availability: Unix (see the man page `pthread_kill(3)` for further information).

See also `os.kill()`. New in version 3.3.

**`signal.pthread_sigmask (how, mask)`**

Fetch and/or change the signal mask of the calling thread. The signal mask is the set of signals whose delivery is currently blocked for the caller. Return the old signal mask as a set of signals.

The behavior of the call is dependent on the value of `how`, as follows.

- **SIG_BLOCK**: The set of blocked signals is the union of the current set and the `mask` argument.
- **SIG_UNBLOCK**: The signals in `mask` are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.
- **SIG_SETMASK**: The set of blocked signals is set to the `mask` argument.

`mask` is a set of signal numbers (e.g. `[signal.SIGINT, signal.SIGTERM]`). Use `range(1, signal.NSIG)` for a full mask including all signals.

For example, `signal.pthread_sigmask(signal.SIG_BLOCK, [])` reads the signal mask of the calling thread.

Availability: Unix. See the man page `sigprocmask(3)` and `pthread_sigmask(3)` for further information.

See also `pause()`, `sigpending()` and `sigwait()`. New in version 3.3.

**`signal.setitimer (which, seconds[, interval])`**

Sets given interval timer (one of `signal.ITIMER_REAL`, `signal.ITIMER_VIRTUAL` or `signal.ITIMER_PROF`) specified by `which` to fire after `seconds` (float is accepted, different from `alarm()`) and after that every `interval` seconds. The interval timer specified by `which` can be cleared by setting seconds to zero.

When an interval timer fires, a signal is sent to the process. The signal sent is dependent on the timer being used; `signal.ITIMER_REAL` will deliver `SIGALRM`, `signal.ITIMER_VIRTUAL` sends `SIGVTALRM`, and `signal.ITIMER_PROF` will deliver `SIGPROF`.

The old values are returned as a tuple: (delay, interval).

Attempting to pass an invalid interval timer will cause an `ItimerError`. Availability: Unix.

**`signal.getitimer (which)`**

Returns current value of a given interval timer specified by `which`. Availability: Unix.

**`signal.set_wakeup_fd (fd)`**

Set the wakeup file descriptor to `fd`. When a signal is received, the signal number is written as a single byte into the fd. This can be used by a library to wakeup a poll or select call, allowing the signal to be fully processed.

The old wakeup fd is returned. `fd` must be non-blocking. It is up to the library to remove any bytes before calling poll or select again.

Use for example `struct.unpack('%uB' % len(data), data)` to decode the signal numbers list.

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a `ValueError` exception to be raised.

**`signal.siginterrupt (signalnum, flag)`**

Change system call restart behaviour: if `flag` is `False`, system calls will be restarted when interrupted by signal `signalnum`, otherwise system calls will be interrupted. Returns nothing. Availability: Unix (see the man page `siginterrupt(3)` for further information).

Note that installing a signal handler with `signal()` will reset the restart behaviour to interruptible by implicitly calling `siginterrupt()` with a true `flag` value for the given signal.

**`signal.signal (signalnum, handler)`**

Set the handler for signal `signalnum` to the function `handler`. `handler` can be a callable Python object taking two arguments (see below), or one of the special values `signal.SIG_IGN` or `signal.SIG_DFL`. The
previous signal handler will be returned (see the description of \texttt{getsignal()} above). (See the Unix man
page \texttt{signal(2)}.)

When threads are enabled, this function can only be called from the main thread; attempting to call it from
other threads will cause a \texttt{ValueError} exception to be raised.

The \texttt{handler} is called with two arguments: the signal number and the current stack frame (\texttt{None} or a frame
object; for a description of frame objects, see the description in the \texttt{inspect} module).

On Windows, \texttt{signal()} can only be called with \texttt{SIGABRT}, \texttt{SIGFPE}, \texttt{SIGILL}, \texttt{SIGINT}, \texttt{SIGSEGV}, or
\texttt{SIGTERM}. A \texttt{ValueError} will be raised in any other case.

\begin{verbatim}

signal.\texttt{sigpending()}

Examine the set of signals that are pending for delivery to the calling thread (i.e., the signals which have
been raised while blocked). Return the set of the pending signals.

Availability: Unix (see the man page \texttt{sigpending(2)} for further information).

See also \texttt{pause()}, \texttt{pthread_sigmask()} and \texttt{sigwait()}. New in version 3.3.

signal.\texttt{sigwait(sigset)}

Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set
\texttt{sigset}. The function accepts the signal (removes it from the pending list of signals), and returns the signal
number.

Availability: Unix (see the man page \texttt{sigwait(3)} for further information).

See also \texttt{pause()}, \texttt{pthread_sigmask()}, \texttt{sigpending()}, \texttt{sigwaitinfo()} and
\texttt{sigtimedwait()}. New in version 3.3.

signal.\texttt{sigwaitinfo(sigset)}

Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set
\texttt{sigset}. The function accepts the signal and removes it from the pending list of signals. If one of the
signals in \texttt{sigset} is already pending for the calling thread, the function will return immediately with infor-
mation about that signal. The signal handler is not called for the delivered signal. The function raises an
\texttt{InterruptedError} if it is interrupted by a signal that is not in \texttt{sigset}.

The return value is an object representing the data contained in the \texttt{siginfo_t} structure, namely:
\texttt{si_signo, si_code, si_errno, si_pid, si_uid, si_status, si_band}.

Availability: Unix (see the man page \texttt{sigwaitinfo(2)} for further information).

See also \texttt{pause()}, \texttt{sigwait()} and \texttt{sigtimedwait()}. New in version 3.3.

signal.\texttt{sigtimedwait(sigset, timeout)}

Like \texttt{sigwaitinfo()}, but takes an additional \texttt{timeout} argument specifying a timeout. If \texttt{timeout} is spec-
ified as 0, a poll is performed. Returns \texttt{None} if a timeout occurs.

Availability: Unix (see the man page \texttt{sigtimedwait(2)} for further information).

See also \texttt{pause()}, \texttt{sigwait()} and \texttt{sigwaitinfo()}. New in version 3.3.

\subsection{Example}

Here is a minimal example program. It uses the \texttt{alarm()} function to limit the time spent waiting to open a
file; this is useful if the file is for a serial device that may not be turned on, which would normally cause the
\texttt{os.open()} to hang indefinitely. The solution is to set a 5-second alarm before opening the file; if the operation
takes too long, the alarm signal will be sent, and the handler raises an exception.

\begin{verbatim}

import signal, os

def handler(signum, frame):
    print(\'Signal handler called with signal\', signum)
    raise OSError(\"Couldn’t open device!\")

import signal
signal.signal(signal.SIGINT, handler)

# open device...

\end{verbatim}
# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)
signal.alarm(0)  # Disable the alarm

18.6 mmap — Memory-mapped file support

Memory-mapped file objects behave like both bytearray and like file objects. You can use mmap objects in most places where bytearray are expected; for example, you can use the re module to search through a memory-mapped file. You can also change a single byte by doing obj[index] = 97, or change a subsequence by assigning to a slice: obj[i1:i2] = b'...'. You can also read and write data starting at the current file position, and seek() through the file to different positions.

A memory-mapped file is created by the mmap constructor, which is different on Unix and on Windows. In either case you must provide a file descriptor for a file opened for update. If you wish to map an existing Python file object, use its fileno() method to obtain the correct value for the fileno parameter. Otherwise, you can open the file using the os.open() function, which returns a file descriptor directly (the file still needs to be closed when done).

**Note:** If you want to create a memory-mapping for a writable, buffered file, you should flush() the file first. This is necessary to ensure that local modifications to the buffers are actually available to the mapping.

For both the Unix and Windows versions of the constructor, access may be specified as an optional keyword parameter. access accepts one of three values: ACCESS_READ, ACCESS_WRITE, or ACCESS_COPY to specify read-only, write-through or copy-on-write memory respectively. access can be used on both Unix and Windows. If access is not specified, Windows mmap returns a write-through mapping. The initial memory values for all three access types are taken from the specified file. Assignment to an ACCESS_READ memory map raises a TypeError exception. Assignment to an ACCESS_WRITE memory map affects both memory and the underlying file. Assignment to an ACCESS_COPY memory map affects memory but does not update the underlying file.

To map anonymous memory, -1 should be passed as the fileno along with the length.

```
18.6. mmap — Memory-mapped file support
```

```python
class mmap (fileno, length, tagname=None, access=ACCESS_DEFAULT[ , offset ])
    (Windows version) Maps length bytes from the file specified by the file handle fileno, and creates a mmap object.
    If length is larger than the current size of the file, the file is extended to contain length bytes. If length is 0, the maximum length of the map is the current size of the file, except that if the file is empty Windows raises an exception (you cannot create an empty mapping on Windows).

    tagname, if specified and not None, is a string giving a tag name for the mapping. Windows allows you to have many different mappings against the same file. If you specify the name of an existing tag, that tag is opened, otherwise a new tag of this name is created. If this parameter is omitted or None, the mapping is created without a name. Avoiding the use of the tag parameter will assist in keeping your code portable between Unix and Windows.

    offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. offset defaults to 0. offset must be a multiple of the ALLOCATIONGRANULARITY.

class mmap (fileno, length, flags=MAP_SHARED, prot=PROT_WRITE|PROT_READ, access=ACCESS_DEFAULT[ , offset ])
    (Unix version) Maps length bytes from the file specified by the file descriptor fileno, and returns a mmap object. If length is 0, the maximum length of the map will be the current size of the file when mmap is called.
```
flags specifies the nature of the mapping. MAP_PRIVATE creates a private copy-on-write mapping, so changes to the contents of the mmap object will be private to this process, and MAP_SHARED creates a mapping that’s shared with all other processes mapping the same areas of the file. The default value is MAP_SHARED.

prot, if specified, gives the desired memory protection; the two most useful values are PROT_READ and PROT_WRITE, to specify that the pages may be read or written. prot defaults to PROT_READ | PROT_WRITE.

access may be specified in lieu of flags and prot as an optional keyword parameter. It is an error to specify both flags, prot and access. See the description of access above for information on how to use this parameter.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. offset defaults to 0. offset must be a multiple of the PAGESIZE or ALLOCATION-GRANULARITY.

To ensure validity of the created memory mapping the file specified by the descriptor fileno is internally automatically synchronized with physical backing store on Mac OS X and OpenVMS.

This example shows a simple way of using mmap:

```python
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write(b"Hello Python!
"
)

with open("hello.txt", "r+b") as f:
    mm = mmap.mmap(f.fileno(), 0)
    print(mm.readline())  # prints b"Hello Python!
"
    print(mm[:5])  # prints b"Hello"
    mm[6:] = b" world!
"
    mm.seek(0)
    print(mm.readline())  # prints b"Hello world!
"

mmap can also be used as a context manager in a with statement:

```python
import mmap

with mmap.mmap(-1, 13) as mm:
    mm.write("Hello world!")
```

New in version 3.2: Context manager support. The next example demonstrates how to create an anonymous map and exchange data between the parent and child processes:

```python
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write(b"Hello world!")
pid = os.fork()
```
if pid == 0:  # In a child process
    mm.seek(0)
    print(mm.readline())

    mm.close()

Memory-mapped file objects support the following methods:

mmap.close()
Closes the mmap. Subsequent calls to other methods of the object will result in a ValueError exception being raised. This will not close the open file.

mmap.closed
True if the file is closed. New in version 3.2.

mmap.find([sub, start[, end]])
Returns the lowest index in the object where the subsequence sub is found, such that sub is contained in the range [start, end]. Optional arguments start and end are interpreted as in slice notation. Returns -1 on failure.

mmap.flush([offset[, size]])
Flushes changes made to the in-memory copy of a file back to disk. Without use of this call there is no guarantee that changes are written back before the object is destroyed. If offset and size are specified, only changes to the given range of bytes will be flushed to disk; otherwise, the whole extent of the mapping is flushed.

(Windows version) A nonzero value returned indicates success; zero indicates failure.

(Unix version) A zero value is returned to indicate success. An exception is raised when the call failed.

mmap.move(dest, src, count)
Copy the count bytes starting at offset src to the destination index dest. If the mmap was created with ACCESS_READ, then calls to move will raise a TypeError exception.

mmap.read([n])
Return a bytes containing up to n bytes starting from the current file position. If the argument is omitted, None or negative, return all bytes from the current file position to the end of the mapping. The file position is updated to point after the bytes that were returned. Changed in version 3.3: Argument can be omitted or None.

mmap.read_byte()
Returns a byte at the current file position as an integer, and advances the file position by 1.

mmap.readline()
Returns a single line, starting at the current file position and up to the next newline.

mmap.resize(newsize)
Resizes the map and the underlying file, if any. If the mmap was created with ACCESS_READ or ACCESS_COPY, resizing the map will raise a TypeError exception.

mmap.rfind([sub, start[, end]])
Returns the highest index in the object where the subsequence sub is found, such that sub is contained in the range [start, end]. Optional arguments start and end are interpreted as in slice notation. Returns -1 on failure.

mmap.seek(pos[, whence])
Set the file’s current position. whence argument is optional and defaults to os.SEEK_SET or 0 (absolute file positioning); other values are os.SEEK_CUR or 1 (seek relative to the current position) and os.SEEK_END or 2 (seek relative to the file’s end).

mmap.size()
Return the length of the file, which can be larger than the size of the memory-mapped area.

mmap.tell()
Returns the current position of the file pointer.
mmap.write(bytes)
    Write the bytes in bytes into memory at the current position of the file pointer; the file position is updated to point after the bytes that were written. If the mmap was created with ACCESS_READ, then writing to it will raise a TypeError exception.

mmap.write_byte(byte)
    Write the integer byte into memory at the current position of the file pointer; the file position is advanced by 1. If the mmap was created with ACCESS_READ, then writing to it will raise a TypeError exception.
CHAPTER
NINETEEN

INTERNET DATA HANDLING

This chapter describes modules which support handling data formats commonly used on the Internet.

19.1 email — An email and MIME handling package

The email package is a library for managing email messages, including MIME and other RFC 2822-based message documents. It is specifically not designed to do any sending of email messages to SMTP (RFC 2821), NNTP, or other servers; those are functions of modules such as smtplib and nntplib. The email package attempts to be as RFC-compliant as possible, supporting in addition to RFC 2822, such MIME-related RFCs as RFC 2045, RFC 2046, RFC 2047, and RFC 2231.

The primary distinguishing feature of the email package is that it splits the parsing and generating of email messages from the internal object model representation of email. Applications using the email package deal primarily with objects; you can add sub-objects to messages, remove sub-objects from messages, completely rearrange the contents, etc. There is a separate parser and a separate generator which handles the transformation from flat text to the object model, and then back to flat text again. There are also handy subclasses for some common MIME object types, and a few miscellaneous utilities that help with such common tasks as extracting and parsing message field values, creating RFC-compliant dates, etc.

The following sections describe the functionality of the email package. The ordering follows a progression that should be common in applications: an email message is read as flat text from a file or other source, the text is parsed to produce the object structure of the email message, this structure is manipulated, and finally, the object tree is rendered back into flat text.

It is perfectly feasible to create the object structure out of whole cloth — i.e. completely from scratch. From there, a similar progression can be taken as above.

Also included are detailed specifications of all the classes and modules that the email package provides, the exception classes you might encounter while using the email package, some auxiliary utilities, and a few examples. For users of the older mimelib package, or previous versions of the email package, a section on differences and porting is provided.

Contents of the email package documentation:

19.1.1 email.message: Representing an email message

The central class in the email package is the Message class, imported from the email.message module. It is the base class for the email object model. Message provides the core functionality for setting and querying header fields, and for accessing message bodies.

Conceptually, a Message object consists of headers and payloads. Headers are RFC 2822 style field names and values where the field name and value are separated by a colon. The colon is not part of either the field name or the field value.

Headers are stored and returned in case-preserving form but are matched case-insensitively. There may also be a single envelope header, also known as the Unix-From header or the From_ header. The payload is either a
string in the case of simple message objects or a list of \texttt{Message} objects for MIME container documents (e.g. \texttt{multipart/*} and \texttt{message/rfc822}).

\texttt{Message} objects provide a mapping style interface for accessing the message headers, and an explicit interface for accessing both the headers and the payload. It provides convenience methods for generating a flat text representation of the message object tree, for accessing commonly used header parameters, and for recursively walking over the object tree.

Here are the methods of the \texttt{Message} class:

\begin{verbatim}
class email.message.Message (policy=compat32)
  The \texttt{policy} argument determines the \texttt{policy} that will be used to update the message model. The default value, \texttt{compat32} maintains backward compatibility with the Python 3.2 version of the email package. For more information see the \texttt{policy} documentation. Changed in version 3.3: The \texttt{policy} keyword argument was added.

  as_string (unixfrom=False, maxheaderlen=0)
  Return the entire message flattened as a string. When optional \texttt{unixfrom} is \texttt{True}, the envelope header is included in the returned string. \texttt{unixfrom} defaults to \texttt{False}. Flattening the message may trigger changes to the \texttt{Message} if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

  Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with \texttt{From} that is required by the unix mbox format. For more flexibility, instantiate a \texttt{Generator} instance and use its \texttt{flatten()} method directly. For example:

  from io import StringIO
  from email.generator import Generator
  fp = StringIO()
  g = Generator(fp, mangle_from_\_\_True, maxheaderlen=60)
  g.flatten(msg)
  text = fp.getvalue()

  \_\_str\__()
  Equivalent to \texttt{as_string(unixfrom=True)}.

  is_multipart()
  Return \texttt{True} if the message’s payload is a list of sub-\texttt{Message} objects, otherwise return \texttt{False}. When \texttt{is_multipart()} returns \texttt{False}, the payload should be a string object.

  set_unixfrom (unixfrom)
  Set the message’s envelope header to \texttt{unixfrom}, which should be a string.

  get_unixfrom()
  Return the message’s envelope header. Defaults to \texttt{None} if the envelope header was never set.

  attach (payload)
  Add the given \texttt{payload} to the current payload, which must be \texttt{None} or a list of \texttt{Message} objects before the call. After the call, the payload will always be a list of \texttt{Message} objects. If you want to set the payload to a scalar object (e.g. a string), use \texttt{set_payload()} instead.

  get_payload (i=\_\_None, decode=\_\_False)
  Return the current payload, which will be a list of \texttt{Message} objects when \texttt{is_multipart()} is \texttt{True}, or a string when \texttt{is_multipart()} is \texttt{False}. If the payload is a list and you mutate the list object, you modify the message’s payload in place.

  With optional argument \texttt{i}, \texttt{get_payload()} will return the \texttt{i}-th element of the payload, counting from zero, if \texttt{is_multipart()} is \texttt{True}. An \texttt{IndexError} will be raised if \texttt{i} is less than 0 or greater than or equal to the number of items in the payload. If the payload is a string (i.e. \texttt{is_multipart()} is \texttt{False}) and \texttt{i} is given, a \texttt{TypeError} is raised.

  Optional \texttt{decode} is a flag indicating whether the payload should be decoded or not, according to the \texttt{Content-Transfer-Encoding} header. When \texttt{True} and the message is not a multipart,
the payload will be decoded if this header’s value is quoted-printable or base64. If some other encoding is used, or Content-Transfer-Encoding header is missing, the payload is returned as-is (undecoded). In all cases the returned value is binary data. If the message is a multipart and the decode flag is True, then None is returned. If the payload is base64 and it was not perfectly formed (missing padding, characters outside the base64 alphabet), then an appropriate defect will be added to the message’s defect property (InvalidBase64PaddingDefect or InvalidBase64CharactersDefect, respectively).

When decode is False (the default) the body is returned as a string without decoding the Content-Transfer-Encoding. However, for a Content-Transfer-Encoding of 8bit, an attempt is made to decode the original bytes using the charset specified by the Content-Type header, using the replace error handler. If no charset is specified, or if the charset given is not recognized by the email package, the body is decoded using the default ASCII charset.

set_payload (payload, charset=None)

Set the entire message object’s payload to payload. It is the client’s responsibility to ensure the payload invariants. Optional charset sets the message’s default character set; see set_charset() for details.

set_charset (charset)

Set the character set of the payload to charset, which can either be a Charset instance (see email.charset), a string naming a character set, or None. If it is a string, it will be converted to a Charset instance. If charset is None, the charset parameter will be removed from the Content-Type header (the message will not be otherwise modified). Anything else will generate a TypeError.

If there is no existing MIME-Version header one will be added. If there is no existing Content-Type header, one will be added with a value of text/plain. Whether the Content-Type header already exists or not, its charset parameter will be set to charset.output_charset. If charset.input_charset and charset.output_charset differ, the payload will be re-encoded to the output_charset. If there is no existing Content-Transfer-Encoding header, then the payload will be transfer-encoded, if needed, using the specified Charset, and a header with the appropriate value will be added. If a Content-Transfer-Encoding header already exists, the payload is assumed to already be correctly encoded using that Content-Transfer-Encoding and is not modified.

get_charset ()

Return the Charset instance associated with the message’s payload.

The following methods implement a mapping-like interface for accessing the message’s RFC 2822 headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by keys(), but in a Message object, headers are always returned in the order they appeared in the original message, or were added to the message later. Any header deleted and then re-added are always appended to the end of the header list.

These semantic differences are intentional and are biased toward maximal convenience.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

In a model generated from bytes, any header values that (in contravention of the RFCs) contain non-ASCII bytes will, when retrieved through this interface, be represented as Header objects with a charset of unknown-8bit.

__len__ ()

Return the total number of headers, including duplicates.

__contains__ (name)

Return true if the message object has a field named name. Matching is done case-insensitively and name should not include the trailing colon. Used for the in operator, e.g.:

if ‘message-id’ in myMessage:
    print(‘Message-ID:’, myMessage[‘message-id’])
__getitem__(name)
Return the value of the named header field. name should not include the colon field separator. If the header is missing, None is returned; a KeyError is never raised.

Note that if the named field appears more than once in the message’s headers, exactly which of those field values will be returned is undefined. Use the get_all() method to get the values of all the extant named headers.

__setitem__(name, val)
Add a header to the message with field name name and value val. The field is appended to the end of the message’s existing fields.

Note that this does not overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name name, delete the field first, e.g.:

def msg[‘subject’]
msg[‘subject’] = ‘Python roolz!’

__delitem__(name)
Delete all occurrences of the field with name name from the message’s headers. No exception is raised if the named field isn’t present in the headers.

keys()
Return a list of all the message’s header field names.

values()
Return a list of all the message’s field values.

items()
Return a list of 2-tuples containing all the message’s field headers and values.

get (name, failobj=None)
Return the value of the named header field. This is identical to __getitem__() except that optional failobj is returned if the named header is missing (defaults to None).

Here are some additional useful methods:

get_all (name, failobj=None)
Return a list of all the values for the field named name. If there are no such named headers in the message, failobj is returned (defaults to None).

add_header(_name, _value, **_params)
Extended header setting. This method is similar to __setitem__() except that additional header parameters can be provided as keyword arguments. _name is the header field to add and _value is the primary value for the header.

For each item in the keyword argument dictionary _params, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as key="value" unless the value is None, in which case only the key will be added. If the value contains non-ASCII characters, it can be specified as a three tuple in the format (CHARSET, LANGUAGE, VALUE), where CHARSET is a string naming the charset to be used to encode the value, LANGUAGE can usually be set to None or the empty string (see RFC 2231 for other possibilities), and VALUE is the string value containing non-ASCII code points. If a three tuple is not passed and the value contains non-ASCII characters, it is automatically encoded in RFC 2231 format using a CHARSET of utf-8 and a LANGUAGE of None.

Here’s an example:

msg.add_header(‘Content-Disposition’, ‘attachment’, filename=’bud.gif’)  

This will add a header that looks like

Content-Disposition: attachment; filename=bud.gif
An example with non-ASCII characters:

```python
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

Which produces

```
Content-Disposition: attachment; filename="iso-8859-1\'\'Fu%DFballer.ppt"
```

---

**replace_header**(_name, _value)

Replace a header. Replace the first header found in the message that matches _name, retaining header order and field name case. If no matching header was found, a **KeyError** is raised.

**get_content_type**()

Return the message’s content type. The returned string is coerced to lower case of the form `maintype/subtype`. If there was no **Content-Type** header in the message the default type as given by **get_default_type()** will be returned. Since according to

**RFC 2045**, messages always have a default type, **get_content_type()** will always return a value.

**RFC 2045** defines a message’s default type to be `text/plain` unless it appears inside a `multipart/digest` container, in which case it would be `message/rfc822`. If the **Content-Type** header has an invalid type specification,

**RFC 2045** mandates that the default type be `text/plain`.

**get_content_maintype**()

Return the message’s main content type. This is the `maintype` part of the string returned by **get_content_type()**.

**get_content_subtype**()

Return the message’s sub-content type. This is the `subtype` part of the string returned by **get_content_type()**.

**get_default_type**()

Return the default content type. Most messages have a default content type of `text/plain`, except for messages that are subparts of `multipart/digest` containers. Such subparts have a default content type of `message/rfc822`.

**set_default_type**(ctype)

Set the default content type. `ctype` should either be `text/plain` or `message/rfc822`, although this is not enforced. The default content type is not stored in the **Content-Type** header.

**get_params**(failobj=None, header='content-type', unquote=True)

Return the message’s **Content-Type** parameters, as a list. The elements of the returned list are 2-tuples of key/value pairs, as split on the `=` sign. The left hand side of the `=` is the key, while the right hand side is the value. If there is no `=` sign in the parameter the value is the empty string, otherwise the value is as described in **get_param()** and is unquoted if optional **unquote** is True (the default).

Optional **failobj** is the object to return if there is no **Content-Type** header. Optional **header** is the header to search instead of **Content-Type**.

**get_param**(param, failobj=None, header='content-type', unquote=True)

Return the value of the **Content-Type** header’s parameter **param** as a string. If the message has no **Content-Type** header or if there is no such parameter, then **failobj** is returned (defaults to None). Optional **header** if given, specifies the message header to use instead of **Content-Type**.

Parameter keys are always compared case insensitively. The return value can either be a string, or a 3-tuple if the parameter was **RFC 2231** encoded. When it’s a 3-tuple, the elements of the value are of the form `(CHARSET, LANGUAGE, VALUE)`. Note that both CHARSET and LANGUAGE can be None, in which case you should consider VALUE to be encoded in the `us-ascii` charset. You can usually ignore LANGUAGE.
If your application doesn’t care whether the parameter was encoded as in

**RFC 2231**, you can collapse the parameter value by calling

```
email.utils.collapse_rfc2231_value()
```

passing in the return value from `get_param()`. This will return a suitably decoded Unicode string when the value is a tuple, or the original string unquoted if it isn’t. For example:

```python
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

In any case, the parameter value (either the returned string, or the VALUE item in the 3-tuple) is always unquoted, unless `unquote` is set to `False`.

### `set_param`

```python
set_param(param, value, header='Content-Type', requote=True, charset=None, language='')
```

Set a parameter in the `Content-Type` header. If the parameter already exists in the header, its value will be replaced with `value`. If the `Content-Type` header as not yet been defined for this message, it will be set to `text/plain` and the new parameter value will be appended as per **RFC 2045**.

Optional `header` specifies an alternative header to `Content-Type`, and all parameters will be quoted as necessary unless optional `requote` is `False` (the default is `True`).

If optional `charset` is specified, the parameter will be encoded according to **RFC 2231**. Optional `language` specifies the RFC 2231 language, defaulting to the empty string. Both `charset` and `language` should be strings.

### `del_param`

```python
del_param(param, header='content-type', requote=True)
```

Remove the given parameter completely from the `Content-Type` header. The header will be re-written in place without the parameter or its value. All values will be quoted as necessary unless `requote` is `False` (the default is `True`). Optional `header` specifies an alternative to `Content-Type`.

### `set_type`

```python
set_type(type, header='Content-Type', requote=True)
```

Set the main type and subtype for the `Content-Type` header. `type` must be a string in the form `maintype/subtype`, otherwise a `ValueError` is raised.

This method replaces the `Content-Type` header, keeping all the parameters in place. If `requote` is `False`, this leaves the existing header’s quoting as is, otherwise the parameters will be quoted (the default).

An alternative header can be specified in the `header` argument. When the `Content-Type` header is set a `MIME-Version` header is also added.

### `get_filename`

```python
get_filename(failobj=None)
```

Return the value of the `filename` parameter of the `Content-Disposition` header of the message. If the header does not have a `filename` parameter, this method falls back to looking for the `name` parameter on the `Content-Type` header. If neither is found, or the header is missing, then `failobj` is returned. The returned string will always be unquoted as per `email.utils.unquote()`.

### `get_boundary`

```python
get_boundary(failobj=None)
```

Return the value of the `boundary` parameter of the `Content-Type` header of the message, or `failobj` if either the header is missing, or has no `boundary` parameter. The returned string will always be unquoted as per `email.utils.unquote()`.

### `set_boundary`

```python
set_boundary(boundary)
```

Set the `boundary` parameter of the `Content-Type` header to `boundary`. `set_boundary()` will always quote `boundary` if necessary. A `HeaderParseError` is raised if the message object has no `Content-Type` header.

Note that using this method is subtly different than deleting the old `Content-Type` header and adding a new one with the new boundary via `add_header()`, because `set_boundary()` preserves the order of the `Content-Type` header in the list of headers. However, it does not preserve any continuation lines which may have been present in the original `Content-Type` header.

### `get_content_charset`

```python
get_content_charset(failobj=None)
```

Return the `charset` parameter of the `Content-Type` header, coerced to lower case. If there is no `Content-Type` header, or if that header has no `charset` parameter, `failobj` is returned.
Note that this method differs from `get_charset()` which returns the `Charset` instance for the default encoding of the message body.

**get_charsets**(failobj=None)

Return a list containing the character set names in the message. If the message is a `multipart`, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

Each item in the list will be a string which is the value of the `charset` parameter in the `Content-Type` header for the represented subpart. However, if the subpart has no `Content-Type` header, no `charset` parameter, or is not of the `text` main MIME type, then that item in the returned list will be `failobj`.

**walk()**

The `walk()` method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use `walk()` as the iterator in a `for` loop; each iteration returns the next subpart.

Here’s an example that prints the MIME type of every part of a multipart message structure:

```python
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

Message objects can also optionally contain two instance attributes, which can be used when generating the plain text of a MIME message.

**preamble**

The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

The `preamble` attribute contains this leading extra-armor text for MIME documents. When the `Parser` discovers some text after the headers but before the first boundary string, it assigns this text to the message’s `preamble` attribute. When the `Generator` is writing out the plain text representation of a MIME message, and it finds the message has a `preamble` attribute, it will write this text in the area between the headers and the first boundary. See `email.parser` and `email.generator` for details.

Note that if the message object has no preamble, the `preamble` attribute will be `None`.

**epilogue**

The `epilogue` attribute acts the same way as the `preamble` attribute, except that it contains text that appears between the last boundary and the end of the message.

You do not need to set the epilogue to the empty string in order for the `Generator` to print a newline at the end of the file.

**defects**

The `defects` attribute contains a list of all the problems found when parsing this message. See `email.errors` for a detailed description of the possible parsing defects.

### 19.1.2 email.parser: Parsing email messages

Message object structures can be created in one of two ways: they can be created from whole cloth by instantiating `Message` objects and stringing them together via `attach()` and `set_payload()` calls, or they can be created by parsing a flat text representation of the email message.
The `email` package provides a standard parser that understands most email document structures, including MIME documents. You can pass the parser a string or a file object, and the parser will return to you the root `Message` instance of the object structure. For simple, non-MIME messages the payload of this root object will likely be a string containing the text of the message. For MIME messages, the root object will return `True` from its `is_multipart()` method, and the subparts can be accessed via the `get_payload()` and `walk()` methods.

There are actually two parser interfaces available for use, the classic `Parser` API and the incremental `FeedParser` API. The classic `Parser` API is fine if you have the entire text of the message in memory as a string, or if the entire message lives in a file on the file system. `FeedParser` is more appropriate for when you’re reading the message from a stream which might block waiting for more input (e.g. reading an email message from a socket). The `FeedParser` can consume and parse the message incrementally, and only returns the root object when you close the parser.

Note that the parser can be extended in limited ways, and of course you can implement your own parser completely from scratch. There is no magical connection between the `email` package’s bundled parser and the `Message` class, so your custom parser can create message object trees any way it finds necessary.

**FeedParser API**

The `FeedParser`, imported from the `email.feedparser` module, provides an API that is conducive to incremental parsing of email messages, such as would be necessary when reading the text of an email message from a source that can block (e.g. a socket). The `FeedParser` can of course be used to parse an email message fully contained in a string or a file, but the classic `Parser` API may be more convenient for such use cases. The semantics and results of the two parser APIs are identical.

The `FeedParser`’s API is simple; you create an instance, feed it a bunch of text until there’s no more to feed it, then close the parser to retrieve the root message object. The `FeedParser` is extremely accurate when parsing standards-compliant messages, and it does a very good job of parsing non-compliant messages, providing information about how a message was deemed broken. It will populate a message object’s `defects` attribute with a list of any problems it found in a message. See the `email.errors` module for the list of defects that it can find.

Here is the API for the `FeedParser`:

```python
class email.parser.FeedParser(_factory=email.message.Message, *, policy=policy.default)
    Create a `FeedParser` instance. Optional `_factory` is a no-argument callable that will be called whenever a new message object is needed. It defaults to the `email.message.Message` class.

    The `policy` keyword specifies a `policy` object that controls a number of aspects of the parser’s operation. The default policy maintains backward compatibility. Changed in version 3.3: Added the `policy` keyword.

    `feed(data)`
    Feed the `FeedParser` some more data. `data` should be a string containing one or more lines. The lines can be partial and the `FeedParser` will stitch such partial lines together properly. The lines in the string can have any of the common three line endings, carriage return, newline, or carriage return and newline (they can even be mixed).

    `close()`
    Closing a `FeedParser` completes the parsing of all previously fed data, and returns the root message object. It is undefined what happens if you feed more data to a closed `FeedParser`.
```

```python
class email.parser.BytesFeedParser(_factory=email.message.Message)
    Works exactly like `FeedParser` except that the input to the `feed()` method must be bytes and not string. New in version 3.2.
```

**Parser class API**

The `Parser` class, imported from the `email.parser` module, provides an API that can be used to parse a message when the complete contents of the message are available in a string or file. The `email.parser` module also provides header-only parsers, called `HeaderParser` and `BytesHeaderParser`, which can be

---

1 As of email package version 3.0, introduced in Python 2.4, the classic `Parser` was re-implemented in terms of the `FeedParser`, so the semantics and results are identical between the two parsers.
used if you’re only interested in the headers of the message. `HeaderParser` and `BytesHeaderParser` can be much faster in these situations, since they do not attempt to parse the message body, instead setting the payload to the raw body as a string. They have the same API as the `Parser` and `BytesParser` classes. New in version 3.3: The `BytesHeaderParser` class.

```python
class email.parser.Parser(_class=email.message.Message, *, policy=policy.default)
```

The constructor for the `Parser` class takes an optional argument `_class`. This must be a callable factory (such as a function or a class), and it is used whenever a sub-message object needs to be created. It defaults to `Message` (see `email.message`). The factory will be called without arguments.

The `policy` keyword specifies a `policy` object that controls a number of aspects of the parser’s operation. The default policy maintains backward compatibility. Changed in version 3.3: Removed the `strict` argument that was deprecated in 2.4. Added the `policy` keyword. The other public `Parser` methods are:

```python
parse(fp, headersonly=False)
```

Read all the data from the file-like object `fp`, parse the resulting text, and return the root message object. `fp` must support both the `readline()` and the `read()` methods on file-like objects.

The text contained in `fp` must be formatted as a block of RFC 2822 style headers and header continuation lines, optionally preceded by a envelope header. The header block is terminated either by the end of the data or by a blank line. Following the header block is the body of the message (which may contain MIME-encoded subparts).

Optional `headersonly` is a flag specifying whether to stop parsing after reading the headers or not. The default is `False`, meaning it parses the entire contents of the file.

```python
parsestr(text, headersonly=False)
```

Similar to the `parse()` method, except it takes a string object instead of a file-like object. Calling this method on a string is exactly equivalent to wrapping `text` in a `StringIO` instance first and calling `parse()`.

Optional `headersonly` is as with the `parse()` method.

```python
class email.parser.BytesParser(_class=email.message.Message, *, policy=policy.default)
```

This class is exactly parallel to `Parser`, but handles bytes input. The `_class` and `strict` arguments are interpreted in the same way as for the `Parser` constructor.

The `policy` keyword specifies a `policy` object that controls a number of aspects of the parser’s operation. The default policy maintains backward compatibility. Changed in version 3.3: Removed the `strict` argument. Added the `policy` keyword.

```python
parse(fp, headersonly=False)
```

Read all the data from the binary file-like object `fp`, parse the resulting bytes, and return the message object. `fp` must support both the `readline()` and the `read()` methods on file-like objects.

The bytes contained in `fp` must be formatted as a block of RFC 2822 style headers and header continuation lines, optionally preceded by a envelope header. The header block is terminated either by the end of the data or by a blank line. Following the header block is the body of the message (which may contain MIME-encoded subparts, including subparts with a `Content-Transfer-Encoding` of 8bit).

Optional `headersonly` is a flag specifying whether to stop parsing after reading the headers or not. The default is `False`, meaning it parses the entire contents of the file.

```python
parsebytes(bytes, headersonly=False)
```

Similar to the `parse()` method, except it takes a byte string object instead of a file-like object. Calling this method on a byte string is exactly equivalent to wrapping `text` in a `BytesIO` instance first and calling `parse()`.

Optional `headersonly` is as with the `parse()` method.

New in version 3.2.

Since creating a message object structure from a string or a file object is such a common task, four functions are provided as a convenience. They are available in the top-level `email` package namespace.
email.message_from_string(s, _class=email.message.Message, *, policy=policy.default)
Return a message object structure from a string. This is exactly equivalent to Parser().parsestr(s).
_class and policy are interpreted as with the Parser class constructor. Changed in version 3.3: Removed the strict argument. Added the policy keyword.

eemail.message_from_bytes(s, _class=email.message.Message, *, policy=policy.default)
Return a message object structure from a byte string. This is exactly equivalent to BytesParser().parsebytes(s). Optional _class and strict are interpreted as with the Parser class constructor. New in version 3.2.Changed in version 3.3: Removed the strict argument. Added the policy keyword.

eemail.message_from_file(fp, _class=email.message.Message, *, policy=policy.default)
Return a message object structure tree from an open file object. This is exactly equivalent to Parser().parse(fp). _class and policy are interpreted as with the Parser class constructor. Changed in version Removed: the strict argument. Added the policy keyword.

eemail.message_from_binary_file(fp, _class=email.message.Message, *, policy=policy.default)
Return a message object structure tree from an open binary file object. This is exactly equivalent to BytesParser().parse(fp). _class and policy are interpreted as with the Parser class constructor. New in version 3.2.Changed in version 3.3: Removed the strict argument. Added the policy keyword.

Here’s an example of how you might use this at an interactive Python prompt:

```python
>>> import email
>>> msg = email.message_from_string(myString)
```

### Additional notes

Here are some notes on the parsing semantics:

- Most non-multipart type messages are parsed as a single message object with a string payload. These objects will return False for is_multipart(). Their get_payload() method will return a string object.

- All multipart type messages will be parsed as a container message object with a list of sub-message objects for their payload. The outer container message will return True for is_multipart() and their get_payload() method will return the list of Message subparts.

- Most messages with a content type of message/* (e.g. message/delivery-status and message/rfc822) will also be parsed as container object containing a list payload of length 1. Their is_multipart() method will return True. The single element in the list payload will be a sub-message object.

- Some non-standards compliant messages may not be internally consistent about their multipart-edness. Such messages may have a Content-Type header of type multipart, but their is_multipart() method may return False. If such messages were parsed with the FeedParser, they will have an instance of the MultipartInvariantViolationDefect class in their defects attribute list. See email.errors for details.

### 19.1.3 email.generator: Generating MIME documents

One of the most common tasks is to generate the flat text of the email message represented by a message object structure. You will need to do this if you want to send your message via the smtplib module or the nntplib module, or print the message on the console. Taking a message object structure and producing a flat text document is the job of the Generator class.

Again, as with the email.parser module, you aren’t limited to the functionality of the bundled generator; you could write one from scratch yourself. However the bundled generator knows how to generate most email in a standards-compliant way, should handle MIME and non-MIME email messages just fine, and is designed so that the transformation from flat text, to a message structure via the Parser class, and back to flat text, is idempotent.
(the input is identical to the output)\(^2\). On the other hand, using the Generator on a Message constructed by program may result in changes to the Message object as defaults are filled in.

**bytes** output can be generated using the BytesGenerator class. If the message object structure contains non-ASCII bytes, this generator’s flatten() method will emit the original bytes. Parsing a binary message and then flattening it with BytesGenerator should be idempotent for standards compliant messages.

Here are the public methods of the Generator class, imported from the email.generator module:

```python
class email.generator.Generator(outfp, mangle_from_=True, maxheaderlen=78, *, policy=None)
```

The constructor for the Generator class takes a file-like object called outfp for an argument. outfp must support the write() method and be usable as the output file for the print() function.

Optional mangle_from_ is a flag that, when True, puts a > character in front of any line in the body that starts exactly as From, i.e. From followed by a space at the beginning of the line. This is the only guaranteed portable way to avoid having such lines be mistaken for a Unix mailbox format envelope header separator (see WHY THE CONTENT-LENGTH FORMAT IS BAD for details). mangle_from_ defaults to True, but you might want to set this to False if you are not writing Unix mailbox format files.

Optional maxheaderlen specifies the longest length for a non-continued header. When a header line is longer than maxheaderlen (in characters, with tabs expanded to 8 spaces), the header will be split as defined in the Header class. Set to zero to disable header wrapping. The default is 78, as recommended (but not required) by RFC 2822.

The policy keyword specifies a policy object that controls a number of aspects of the generator’s operation. If no policy is specified, then the policy attached to the message object passed to flatten is used. Changed in version 3.3: Added the policy keyword. The other public Generator methods are:

```python
flatten(msg, unixfrom=False, linesep=None)
```

Print the textual representation of the message object structure rooted at msg to the output file specified when the Generator instance was created. Subparts are visited depth-first and the resulting text will be properly MIME encoded.

Optional unixfrom is a flag that forces the printing of the envelope header delimiter before the first RFC 2822 header of the root message object. If the root object has no envelope header, a standard one is crafted. By default, this is set to False to inhibit the printing of the envelope delimiter.

Note that for subparts, no envelope header is ever printed.

Optional linesep specifies the line separator character used to terminate lines in the output. If specified it overrides the value specified by the msg’s or Generator’s policy.

Because strings cannot represent non-ASCII bytes, if the policy that applies when flatten is run has cte_type set to 8bit, Generator will operate as if it were set to 7bit. This means that messages parsed with a Bytes parser that have a Content-Transfer-Encoding of 8bit will be converted to a use a 7bit Content-Transfer-Encoding. Non-ASCII bytes in the headers will be RFC 2047 encoded with a charset of unknown-8bit. Changed in version 3.2: Added support for re-encoding 8bit message bodies, and the linesep argument.

```python
clone(fp)
```

Return an independent clone of this Generator instance with the exact same options.

```python
write(s)
```

Write the string s to the underlying file object, i.e. outfp passed to Generator’s constructor. This provides just enough file-like API for Generator instances to be used in the print() function.

As a convenience, see the Message methods as_string() and str(aMessage), a.k.a. __str__(), which simplify the generation of a formatted string representation of a message object. For more detail, see email.message.

\(^2\) This statement assumes that you use the appropriate setting for the unixfrom argument, and that you set maxheaderlen=0 (which will preserve whatever the input line lengths were). It is also not strictly true, since in many cases runs of whitespace in headers are collapsed into single blanks. The latter is a bug that will eventually be fixed.
class email.generator.BytesGenerator (outfp, mangle_from_=True, maxheaderlen=78, *, policy=policy.default)

The constructor for the BytesGenerator class takes a binary file-like object called outfp for an argument. outfp must support a write() method that accepts binary data.

Optional mangle_from_ is a flag that, when True, puts a > character in front of any line in the body that starts exactly as From, i.e. From followed by a space at the beginning of the line. This is the only guaranteed portable way to avoid having such lines be mistaken for a Unix mailbox format envelope header separator (see WHY THE CONTENT-LENGTH FORMAT IS BAD for details). mangle_from_ defaults to True, but you might want to set this to False if you are not writing Unix mailbox format files.

Optional maxheaderlen specifies the longest length for a non-continued header. When a header line is longer than maxheaderlen (in characters, with tabs expanded to 8 spaces), the header will be split as defined in the Header class. Set to zero to disable header wrapping. The default is 78, as recommended (but not required) by RFC 2822.

The policy keyword specifies a policy object that controls a number of aspects of the generator’s operation. The default policy maintains backward compatibility. Changed in version 3.3: Added the policy keyword. The other public BytesGenerator methods are:

flatten (msg, unixfrom=False, linesep=None)

Print the textual representation of the message object structure rooted at msg to the output file specified when the BytesGenerator instance was created. Subparts are visited depth-first and the resulting text will be properly MIME encoded. If the policy option cte_type is 8bit (the default), then any bytes with the high bit set in the original parsed message that have not been modified will be copied faithfully to the output. If cte_type is 7bit, the bytes will be converted as needed using an ASCII-compatible Content-Transfer-Encoding. In particular, RFC-invalid non-ASCII bytes in headers will be encoded using the MIME unknown-8bit character set, thus rendering them RFC-compliant.

Messages parsed with a Bytes parser that have a Content-Transfer-Encoding of 8bit will be reconstructed as 8bit if they have not been modified.

Optional unixfrom is a flag that forces the printing of the envelope header delimiter before the first RFC 2822 header of the root message object. If the root object has no envelope header, a standard one is crafted. By default, this is set to False to inhibit the printing of the envelope delimiter.

Note that for subparts, no envelope header is ever printed.

Optional linesep specifies the line separator character used to terminate lines in the output. If specified it overrides the value specified by the Generator’s policy.

clone (fp)

Return an independent clone of this BytesGenerator instance with the exact same options.

write (s)

Write the string s to the underlying file object. s is encoded using the ASCII codec and written to the write method of the outfp outfp passed to the BytesGenerator’s constructor. This provides just enough file-like API for BytesGenerator instances to be used in the print() function.

New in version 3.2.

The email.generator module also provides a derived class, called DecodedGenerator which is like the Generator base class, except that non-text parts are substituted with a format string representing the part.

class email.generator.DecodedGenerator (outfp, mangle_from_=True, maxheaderlen=78, fmt=None)

This class, derived from Generator walks through all the subparts of a message. If the subpart is of main type text, then it prints the decoded payload of the subpart. Optional _mangle_from_ and maxheaderlen are as with the Generator base class.

If the subpart is not of main type text, optional fmt is a format string that is used instead of the message payload. fmt is expanded with the following keywords, %(key)%s format:

• type – Full MIME type of the non-text part
The default value for fmt is None, meaning

[Non-text (%(type)s) part of message omitted, filename %(filename)s]

19.1.4 email.policy: Policy Objects

New in version 3.3. The email package’s prime focus is the handling of email messages as described by the various email and MIME RFCs. However, the general format of email messages (a block of header fields each consisting of a name followed by a colon followed by a value, the whole block followed by a blank line and an arbitrary ‘body’), is a format that has found utility outside of the realm of email. Some of these uses conform fairly closely to the main RFCs, some do not. And even when working with email, there are times when it is desirable to break strict compliance with the RFCs.

Policy objects give the email package the flexibility to handle all these disparate use cases.

A Policy object encapsulates a set of attributes and methods that control the behavior of various components of the email package during use. Policy instances can be passed to various classes and methods in the email package to alter the default behavior. The settable values and their defaults are described below.

There is a default policy used by all classes in the email package. This policy is named Compat32, with a corresponding pre-defined instance named compat32. It provides for complete backward compatibility (in some cases, including bug compatibility) with the pre-Python3.3 version of the email package.

The first part of this documentation covers the features of Policy, an abstract base class that defines the features that are common to all policy objects, including compat32. This includes certain hook methods that are called internally by the email package, which a custom policy could override to obtain different behavior.

When a Message object is created, it acquires a policy. By default this will be compat32, but a different policy can be specified. If the Message is created by a parser, a policy passed to the parser will be the policy used by the Message it creates. If the Message is created by the program, then the policy can be specified when it is created. When a Message is passed to a generator, the generator uses the policy from the Message by default, but you can also pass a specific policy to the generator that will override the one stored on the Message object.

Policy instances are immutable, but they can be cloned, accepting the same keyword arguments as the class constructor and returning a new Policy instance that is a copy of the original but with the specified attributes values changed.

As an example, the following code could be used to read an email message from a file on disk and pass it to the system sendmail program on a Unix system:

```python
>>> from email import message_from_binary_file
>>> from email.generator import BytesGenerator
>>> from email import policy
>>> from subprocess import Popen, PIPE
>>> with open('mymsg.txt', 'rb') as f:
...     msg = message_from_binary_file(f, policy=policy.default)
>>> p = Popen(['sendmail', msg['To'].addresses[0]], stdin=PIPE)
>>> g = BytesGenerator(p.stdin, policy=msg.policy.clone(linesep='\r
'))
>>> g.flatten(msg)
>>> p.stdin.close()
>>> rc = p.wait()
```
Here we are telling `BytesGenerator` to use the RFC correct line separator characters when creating the binary string to feed into `sendmail`'s stdin, where the default policy would use \n line separators.

Policy objects can also be combined using the addition operator, producing a policy object whose settings are a combination of the non-default values of the summed objects:

```python
>>> compat_SMTP = policy.compat32.clone(linesep='\r\n')
>>> compat_strict = policy.compat32.clone(raise_on_defect=True)
>>> compat_strict_SMTP = compat_SMTP + compat_strict
```

This operation is not commutative; that is, the order in which the objects are added matters. To illustrate:

```python
>>> policy100 = policy.compat32.clone(max_line_length=100)
>>> policy80 = policy.compat32.clone(max_line_length=80)
>>> apolicy = policy100 + policy80
>>> apolicy.max_line_length
80
>>> apolicy = policy80 + policy100
>>> apolicy.max_line_length
100
```

```python
class email.policy.Policy(**kw)
```

This is the abstract base class for all policy classes. It provides default implementations for a couple of trivial methods, as well as the implementation of the immutability property, the `clone()` method, and the constructor semantics.

The constructor of a policy class can be passed various keyword arguments. The arguments that may be specified are any non-method properties on this class, plus any additional non-method properties on the concrete class. A value specified in the constructor will override the default value for the corresponding attribute.

This class defines the following properties, and thus values for the following may be passed in the constructor of any policy class:

- **max_line_length**
  - The maximum length of any line in the serialized output, not counting the end of line character(s).
  - Default is 78, per RFC 5322. A value of 0 or None indicates that no line wrapping should be done at all.

- **linesep**
  - The string to be used to terminate lines in serialized output. The default is \n because that’s the internal end-of-line discipline used by Python, though \r\n is required by the RFCs.

- **cte_type**
  - Controls the type of Content Transfer Encodings that may be or are required to be used. The possible values are:
    - **7bit**
      - all data must be “7 bit clean” (ASCII-only). This means that where necessary data will be encoded using either quoted-printable or base64 encoding.
    - **8bit**
      - data is not constrained to be 7 bit clean. Data in headers is still required to be ASCII-only and so will be encoded (see ‘binary_fold’ below for an exception), but body parts may use the 8bit CTE.

A `cte_type` value of 8bit only works with `BytesGenerator`, not `Generator`, because strings cannot contain binary data. If a `Generator` is operating under a policy that specifies `cte_type=8bit`, it will act as if `cte_type` is 7bit.

- **raise_on_defect**
  - If True, any defects encountered will be raised as errors. If False (the default), defects will be passed to the `register_defect()` method.

The following `Policy` method is intended to be called by code using the email library to create policy instances with custom settings:
The Python Library Reference, Release 3.3.3

clone(**kw)

Return a new Policy instance whose attributes have the same values as the current instance, except where those attributes are given new values by the keyword arguments.

The remaining Policy methods are called by the email package code, and are not intended to be called by an application using the email package. A custom policy must implement all of these methods.

handle_defect(obj, defect)

Handle a defect found on obj. When the email package calls this method, defect will always be a subclass of Defect.

The default implementation checks the raise_on_defect flag. If it is True, defect is raised as an exception. If it is False (the default), obj and defect are passed to register_defect().

register_defect(obj, defect)

Register a defect on obj. In the email package, defect will always be a subclass of Defect.

The default implementation calls the append method of the defects attribute of obj. When the email package calls handle_defect, obj will normally have a defects attribute that has an append method. Custom object types used with the email package (for example, custom Message objects) should also provide such an attribute, otherwise defects in parsed messages will raise unexpected errors.

header_max_count(name)

Return the maximum allowed number of headers named name.

Called when a header is added to a Message object. If the returned value is not 0 or None, and there are already a number of headers with the name name equal to the value returned, a ValueError is raised.

Because the default behavior of Message.__setitem__ is to append the value to the list of headers, it is easy to create duplicate headers without realizing it. This method allows certain headers to be limited in the number of instances of that header that may be added to a Message programmatically. (The limit is not observed by the parser, which will faithfully produce as many headers as exist in the message being parsed.)

The default implementation returns None for all header names.

header_source_parse(sourcelines)

The email package calls this method with a list of strings, each string ending with the line separation characters found in the source being parsed. The first line includes the field header name and separator. All whitespace in the source is preserved. The method should return the (name, value) tuple that is to be stored in the Message to represent the parsed header.

If an implementation wishes to retain compatibility with the existing email package policies, name should be the case preserved name (all characters up to the ‘:’ separator), while value should be the unfolded value (all line separator characters removed, but whitespace kept intact), stripped of leading whitespace.

sourcelines may contain surrogateescaped binary data.

There is no default implementation

header_store_parse(name, value)

The email package calls this method with the name and value provided by the application program when the application program is modifying a Message programmatically (as opposed to a Message created by a parser). The method should return the (name, value) tuple that is to be stored in the Message to represent the header.

If an implementation wishes to retain compatibility with the existing email package policies, the name and value should be strings or string subclasses that do not change the content of the passed in arguments.

There is no default implementation

header_fetch_parse(name, value)

The email package calls this method with the name and value currently stored in the Message when
that header is requested by the application program, and whatever the method returns is what is passed back to the application as the value of the header being retrieved. Note that there may be more than one header with the same name stored in the Message; the method is passed the specific name and value of the header destined to be returned to the application.

$value$ may contain surrogateescaped binary data. There should be no surrogateescaped binary data in the value returned by the method.

There is no default implementation

`fold(name, value)`
The email package calls this method with the $name$ and $value$ currently stored in the Message for a given header. The method should return a string that represents that header “folded” correctly (according to the policy settings) by composing the $name$ with the $value$ and inserting $\text{linesep}$ characters at the appropriate places. See RFC 5322 for a discussion of the rules for folding email headers.

$value$ may contain surrogateescaped binary data. There should be no surrogateescaped binary data in the string returned by the method.

`fold_binary(name, value)`
The same as `fold()`, except that the returned value should be a bytes object rather than a string.

$value$ may contain surrogateescaped binary data. These could be converted back into binary data in the returned bytes object.

class `email.policyCompat32(**kw)`
This concrete Policy is the backward compatibility policy. It replicates the behavior of the email package in Python 3.2. The policy module also defines an instance of this class, compat32, that is used as the default policy. Thus the default behavior of the email package is to maintain compatibility with Python 3.2.

The class provides the following concrete implementations of the abstract methods of Policy:

`header_source_parse(source_lines)`
The name is parsed as everything up to the ‘:’ and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

`header_store_parse(name, value)`
The name and value are returned unmodified.

`header_fetch_parse(name, value)`
If the value contains binary data, it is converted into a Header object using the unknown-8bit charset. Otherwise it is returned unmodified.

`fold(name, value)`
Headers are folded using the Header folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the $\text{max_line_length}$. Non-ASCII binary data are CTE encoded using the unknown-8bit charset.

`fold_binary(name, value)`
Headers are folded using the Header folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the $\text{max_line_length}$. If $\text{cte_type}$ is 7bit, non-ascii binary data is CTE encoded using the unknown-8bit charset. Otherwise the original source header is used, with its existing line breaks and any (RFC invalid) binary data it may contain.

Note: The documentation below describes new policies that are included in the standard library on a provisional basis. Backwards incompatible changes (up to and including removal of the feature) may occur if deemed necessary by the core developers.

class `email.policy.EmailPolicy(**kw)`
This concrete Policy provides behavior that is intended to be fully compliant with the current email RFCs. These include (but are not limited to) RFC 5322, RFC 2047, and the current MIME RFCs.
This policy adds new header parsing and folding algorithms. Instead of simple strings, headers are custom objects with custom attributes depending on the type of the field. The parsing and folding algorithm fully implement RFC 2047 and RFC 5322.

In addition to the settable attributes listed above that apply to all policies, this policy adds the following additional attributes:

**refold_source**
If the value for a header in the Message object originated from a parser (as opposed to being set by a program), this attribute indicates whether or not a generator should refold that value when transforming the message back into stream form. The possible values are:

<table>
<thead>
<tr>
<th>None</th>
<th>All source values use original folding</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>Source values that have any line that is longer than max_line_length will be refolded</td>
</tr>
<tr>
<td>all</td>
<td>All values are refolded.</td>
</tr>
</tbody>
</table>

The default is long.

**header_factory**
A callable that takes two arguments, name and value, where name is a header field name and value is an unfolded header field value, and returns a string subclass that represents that header. A default header_factory (see headerregistry) is provided that understands some of the RFC 5322 header field types. (Currently address fields and date fields have special treatment, while all other fields are treated as unstructured. This list will be completed before the extension is marked stable.)

The class provides the following concrete implementations of the abstract methods of Policy:

**header_max_count (name)**
Returns the value of the max_count attribute of the specialized class used to represent the header with the given name.

**header_source_parse (sourcelines)**
The implementation of this method is the same as that for the Compat32 policy.

**header_store_parse (name, value)**
The name is returned unchanged. If the input value has a name attribute and it matches name ignoring case, the value is returned unchanged. Otherwise the name and value are passed to header_factory, and the resulting custom header object is returned as the value. In this case a ValueError is raised if the input value contains CR or LF characters.

**header_fetch_parse (name, value)**
If the value has a name attribute, it is returned to unmodified. Otherwise the name, and the value with any CR or LF characters removed, are passed to the header_factory, and the resulting custom header object is returned. Any surrogateescaped bytes get turned into the unicode unknown-character glyph.

**fold (name, value)**
Header folding is controlled by the refold_source policy setting. A value is considered to be a 'source value' if and only if it does not have a name attribute (having a name attribute means it is a header object of some sort). If a source value needs to be refolded according to the policy, it is converted into a custom header object by passing the name and the value with any CR and LF characters removed to the header_factory. Folding of a custom header object is done by calling its fold method with the current policy.

Source values are split into lines using splitlines(). If the value is not to be refolded, the lines are rejoined using the linesep from the policy and returned. The exception is lines containing non-ascii binary data. In that case the value is refolded regardless of the refold_source setting, which causes the binary data to be CTE encoded using the unknown-8bit charset.

**fold_binary (name, value)**
The same as fold() if cte_type is 7bit, except that the returned value is bytes.

19.1. email — An email and MIME handling package 735
If `cte_type` is 8bit, non-ASCII binary data is converted back into bytes. Headers with binary data are not refolded, regardless of the `refold_header` setting, since there is no way to know whether the binary data consists of single byte characters or multibyte characters.

The following instances of `EmailPolicy` provide defaults suitable for specific application domains. Note that in the future the behavior of these instances (in particular the `HTTP` instance) may be adjusted to conform even more closely to the RFCs relevant to their domains.

```python
e-mail.policy.default
    An instance of `EmailPolicy` with all defaults unchanged. This policy uses the standard Python `\n` line endings rather than the RFC-correct `\r\n`.

e-mail.policy.SMTP
    Suitable for serializing messages in conformance with the email RFCs. Like `default`, but with `linesep` set to `\r\n`, which is RFC compliant.

e-mail.policy.HTTP
    Suitable for serializing headers with for use in HTTP traffic. Like `SMTP` except that `max_line_length` is set to `None` (unlimited).

e-mail.policy.strict
    Convenience instance. The same as `default` except that `raise_on_defect` is set to `True`. This allows any policy to be made strict by writing:

    somepolicy + policy.strict
```

With all of these `EmailPolicies`, the effective API of the email package is changed from the Python 3.2 API in the following ways:

- Setting a header on a `Message` results in that header being parsed and a custom header object created.
- Fetching a header value from a `Message` results in that header being parsed and a custom header object created and returned.
- Any custom header object, or any header that is refolded due to the policy settings, is folded using an algorithm that fully implements the RFC folding algorithms, including knowing where encoded words are required and allowed.

From the application view, this means that any header obtained through the `Message` is a custom header object with custom attributes, whose string value is the fully decoded unicode value of the header. Likewise, a header may be assigned a new value, or a new header created, using a unicode string, and the policy will take care of converting the unicode string into the correct RFC encoded form.

The custom header objects and their attributes are described in `headerregistry`.

### 19.1.5 email.headerregistry: Custom Header Objects

**Note:** The headerregistry module has been included in the standard library on a provisional basis. Backwards incompatible changes (up to and including removal of the module) may occur if deemed necessary by the core developers.

New in version 3.3: as a provisional module. Headers are represented by customized subclasses of `str`. The particular class used to represent a given header is determined by the `header_factory` of the policy in effect when the headers are created. This section documents the particular `header_factory` implemented by the email package for handling RFC 5322 compliant email messages, which not only provides customized header objects for various header types, but also provides an extension mechanism for applications to add their own custom header types.

When using any of the policy objects derived from `EmailPolicy`, all headers are produced by `HeaderRegistry` and have `BaseHeader` as their last base class. Each header class has an additional base class that is determined by the type of the header. For example, many headers have the class `UnstructuredHeader` as their other base class. The specialized second class for a header is determined
by the name of the header, using a lookup table stored in the \texttt{HeaderRegistry}. All of this is managed transparently for the typical application program, but interfaces are provided for modifying the default behavior for use by more complex applications.

The sections below first document the header base classes and their attributes, followed by the API for modifying the behavior of \texttt{HeaderRegistry}, and finally the support classes used to represent the data parsed from structured headers.

\texttt{class email.headerregistry.BaseHeader(name, value)}

\texttt{name} and \texttt{value} are passed to \texttt{BaseHeader} from the \texttt{header\_factory} call. The string value of any header object is the \texttt{value} fully decoded to unicode.

This base class defines the following read-only properties:

\texttt{name}

The name of the header (the portion of the field before the `:`). This is exactly the value passed in the \texttt{header\_factory} call for \texttt{name}; that is, case is preserved.

\texttt{defects}

A tuple of \texttt{HeaderDefect} instances reporting any RFC compliance problems found during parsing. The email package tries to be complete about detecting compliance issues. See the \texttt{errors} module for a discussion of the types of defects that may be reported.

\texttt{max\_count}

The maximum number of headers of this type that can have the same \texttt{name}. A value of \texttt{None} means unlimited. The \texttt{BaseHeader} value for this attribute is \texttt{None}; it is expected that specialized header classes will override this value as needed.

\texttt{BaseHeader} also provides the following method, which is called by the email library code and should not in general be called by application programs:

\texttt{fold(*policy)}

Return a string containing \texttt{linesep} characters as required to correctly fold the header according to \texttt{policy}. A \texttt{cte\_type} of \texttt{8bit} will be treated as if it were \texttt{7bit}, since strings may not contain binary data.

\texttt{BaseHeader} by itself cannot be used to create a header object. It defines a protocol that each specialized header cooperates with in order to produce the header object. Specifically, \texttt{BaseHeader} requires that the specialized class provide a \texttt{classmethod()} named \texttt{parse}. This method is called as follows:

\texttt{parse(string, kwds)}

\texttt{kwds} is a dictionary containing one pre-initialized key, \texttt{defects}. \texttt{defects} is an empty list. The parse method should append any detected defects to this list. On return, the \texttt{kwds} dictionary must contain values for at least the keys \texttt{decoded} and \texttt{defects}. \texttt{decoded} should be the string value for the header (that is, the header value fully decoded to unicode). The parse method should assume that \texttt{string} may contain transport encoded parts, but should correctly handle all valid unicode characters as well so that it can parse un-encoded header values.

\texttt{BaseHeader}'s \texttt{__new__} then creates the header instance, and calls its \texttt{init} method. The specialized class only needs to provide an \texttt{init} method if it wishes to set additional attributes beyond those provided by \texttt{BaseHeader} itself. Such an \texttt{init} method should look like this:

\texttt{def init(self, *args, **kw):}

\hspace{1cm} self._myattr = kw.pop('myattr')

\hspace{1cm} super().init(*args, **kw)

That is, anything extra that the specialized class puts in to the \texttt{kwds} dictionary should be removed and handled, and the remaining contents of \texttt{kw} (and \texttt{args}) passed to the \texttt{BaseHeader init} method.

\texttt{class email.headerregistry.UnstructuredHeader}

An "unstructured" header is the default type of header in \texttt{RFC 5322}. Any header that does not have a specified syntax is treated as unstructured. The classic example of an unstructured header is the \texttt{Subject} header.
In RFC 5322, an unstructured header is a run of arbitrary text in the ASCII character set. RFC 2047, however, has an RFC 5322 compatible mechanism for encoding non-ASCII text as ASCII characters within a header value. When a value containing encoded words is passed to the constructor, the `UnstructuredHeader` parser converts such encoded words back in to the original unicode, following the RFC 2047 rules for unstructured text. The parser uses heuristics to attempt to decode certain non-compliant encoded words. Defects are registered in such cases, as well as defects for issues such as invalid characters within the encoded words or the non-encoded text.

This header type provides no additional attributes.

```python
class email.headerregistry.DateHeader
RFC 5322 specifies a very specific format for dates within email headers. The `DateHeader` parser recognizes that date format, as well as recognizing a number of variant forms that are sometimes found “in the wild”.

This header type provides the following additional attributes:

**datetime**

If the header value can be recognized as a valid date of one form or another, this attribute will contain a `datetime` instance representing that date. If the timezone of the input date is specified as -0000 (indicating it is in UTC but contains no information about the source timezone), then `datetime` will be a naive `datetime`. If a specific timezone offset is found (including +0000), then `datetime` will contain an aware `datetime` that uses `datetime.timezone` to record the timezone offset.

The decoded value of the header is determined by formatting the `datetime` according to the RFC 5322 rules; that is, it is set to:

```python
eMail.utils.format_datetime(self.datetime)
```

When creating a `DateHeader`, value may be `datetime` instance. This means, for example, that the following code is valid and does what one would expect:

```python
msg[’Date’] = datetime(2011, 7, 15, 21)
```

Because this is a naive `datetime` it will be interpreted as a UTC timestamp, and the resulting value will have a timezone of -0000. Much more useful is to use the `localtime()` function from the `utils` module:

```python
msg[’Date’] = utils.localtime()
```

This example sets the date header to the current time and date using the current timezone offset.

```python
class email.headerregistry.AddressHeader
Address headers are one of the most complex structured header types. The `AddressHeader` class provides a generic interface to any address header.

This header type provides the following additional attributes:

**groups**

A tuple of `Group` objects encoding the addresses and groups found in the header value. Addresses that are not part of a group are represented in this list as single-address `Groups` whose `display_name` is None.

**addresses**

A tuple of `Address` objects encoding all of the individual addresses from the header value. If the header value contains any groups, the individual addresses from the group are included in the list at the point where the group occurs in the value (that is, the list of addresses is “flattened” into a one dimensional list).

The decoded value of the header will have all encoded words decoded to unicode. `idna` encoded domain names are also decoded to unicode. The decoded value is set by joining the `str` value of the elements of the `groups` attribute with ‘, ’.
A list of `Address` and `Group` objects in any combination may be used to set the value of an address header. `Group` objects whose `display_name` is `None` will be interpreted as single addresses, which allows an address list to be copied with groups intact by using the list obtained `groups` attribute of the source header.

```python
class email.headerregistry.SingleAddressHeader
    A subclass of `AddressHeader` that adds one additional attribute:

    `address`
        The single address encoded by the header value. If the header value actually contains more than one address (which would be a violation of the RFC under the default `policy`), accessing this attribute will result in a `ValueError`.
```

Many of the above classes also have a `Unique` variant (for example, `UniqueUnstructuredHeader`). The only difference is that in the `Unique` variant, `max_count` is set to 1.

```python
class email.headerregistry.MIMEVersionHeader
    There is really only one valid value for the `MIME-Version` header, and that is 1.0. For future proofing, this header class supports other valid version numbers. If a version number has a valid value per RFC 2045, then the header object will have non-None values for the following attributes:

    `version`
        The version number as a string, with any whitespace and/or comments removed.

    `major`
        The major version number as an integer

    `minor`
        The minor version number as an integer
```

```python
class email.headerregistry.ParameterizedMIMEHeader
    MOME headers all start with the prefix `Content-`. Each specific header has a certain value, described under the class for that header. Some can also take a list of supplemental parameters, which have a common format. This class serves as a base for all the MIME headers that take parameters.

    `params`
        A dictionary mapping parameter names to parameter values.
```

```python
class email.headerregistry.ContentTypeHeader
    A `ParameterizedMIMEHeader` class that handles the `Content-Type` header.

    `content_type`
        The content type string, in the form `maintype/subtype`.

    `maintype`

    `subtype`
```

```python
class email.headerregistry.ContentDispositionHeader
    A `ParameterizedMIMEHeader` class that handles the `Content-Disposition` header.

    `content-disposition`
        inline and attachment are the only valid values in common use.
```

```python
class email.headerregistry.ContentTransferEncoding
    Handles the `Content-Transfer-Encoding` header.

    `cte`
        Valid values are 7bit, 8bit, base64, and quoted-printable. See RFC 2045 for more information.
```

```python
class email.headerregistry.HeaderRegistry (base_class=BaseHeader, default_class=UnstructuredHeader, use_default_map=True)
    This is the factory used by `EmailPolicy` by default. `HeaderRegistry` builds the class used to create a header instance dynamically, using `base_class` and a specialized class retrieved from a registry that it holds. When a given header name does not appear in the registry, the class specified by `default_class` is used as the
specialized class. When `use_default_map` is `True` (the default), the standard mapping of header names to
classes is copied in to the registry during initialization. `base_class` is always the last class in the generated
class’s `__bases__` list.

The default mappings are:

- `subject` UniqueUnstructuredHeader
- `date` UniqueDateHeader
- `resent-date` DateHeader
- `orig-date` UniqueDateHeader
- `sender` UniqueSingleAddressHeader
- `resent-sender` SingleAddressHeader
- `to` UniqueAddressHeader
- `resent-to` AddressHeader
- `cc` UniqueAddressHeader
- `resent-cc` AddressHeader
- `from` UniqueAddressHeader
- `resent-from` AddressHeader
- `reply-to` UniqueAddressHeader

`HeaderRegistry` has the following methods:

- `map_to_type(self, name, cls)`
  
  `name` is the name of the header to be mapped. It will be converted to lower case in the registry. `cls`
  is the specialized class to be used, along with `base_class`, to create the class used to instantiate headers
  that match `name`.

- `__getitem__(name)`
  
  Construct and return a class to handle creating a `name` header.

- `__call__(name, value)`
  
  Retrieves the specialized header associated with `name` from the registry (using `default_class` if `name`
  does not appear in the registry) and composes it with `base_class` to produce a class, calls the con-
  structed class’s constructor, passing it the same argument list, and finally returns the class instance
  created thereby.

The following classes are the classes used to represent data parsed from structured headers and can, in general, be
used by an application program to construct structured values to assign to specific headers.

```python
class email.headerregistry.Address(display_name='', username='', domain=''
                                  addr_spec=None):
```

The class used to represent an email address. The general form of an address is:

- `[display_name] <username@domain>`

  or:

- `username@domain`

  where each part must conform to specific syntax rules spelled out in

RFC 5322.

As a convenience `addr_spec` can be specified instead of `username` and `domain`, in which case `username` and
`domain` will be parsed from the `addr_spec`. An `addr_spec` must be a properly RFC quoted string; if it is not
`Address` will raise an error. Unicode characters are allowed and will be property encoded when serialized.
However, per the RFCs, unicode is not allowed in the username portion of the address.
**display_name**
The display name portion of the address, if any, with all quoting removed. If the address does not have a display name, this attribute will be an empty string.

**username**
The username portion of the address, with all quoting removed.

**domain**
The domain portion of the address.

**addr_spec**
The username@domain portion of the address, correctly quoted for use as a bare address (the second form shown above). This attribute is not mutable.

**__str__()**
The str value of the object is the address quoted according to RFC 5322 rules, but with no Content Transfer Encoding of any non-ASCII characters.

To support SMTP (RFC 5321), Address handles one special case: if username and domain are both the empty string (or None), then the string value of the Address is <>.

**class** email.headerregistry.Group(display_name=None, addresses=None)
The class used to represent an address group. The general form of an address group is:

```
display_name: [address-list];
```

As a convenience for processing lists of addresses that consist of a mixture of groups and single addresses, a Group may also be used to represent single addresses that are not part of a group by setting display_name to None and providing a list of the single address as addresses.

**display_name**
The display_name of the group. If it is None and there is exactly one Address in addresses, then the Group represents a single address that is not in a group.

**addresses**
A possibly empty tuple of Address objects representing the addresses in the group.

**__str__()**
The str value of a Group is formatted according to RFC 5322, but with no Content Transfer Encoding of any non-ASCII characters. If display_name is none and there is a single Address in the addresses list, the str value will be the same as the str of that single Address.

### 19.1.6 email.mime: Creating email and MIME objects from scratch

Ordinarily, you get a message object structure by passing a file or some text to a parser, which parses the text and returns the root message object. However you can also build a complete message structure from scratch, or even individual Message objects by hand. In fact, you can also take an existing structure and add new Message objects, move them around, etc. This makes a very convenient interface for slicing-and-dicing MIME messages.

You can create a new object structure by creating Message instances, adding attachments and all the appropriate headers manually. For MIME messages though, the email package provides some convenient subclasses to make things easier.

Here are the classes:

**class** email.mime.base.MIMEBase(_maintype, _subtype, **_params)
Module: email.mime.base

This is the base class for all the MIME-specific subclasses of Message. Ordinarily you won’t create instances specifically of MIMEBase, although you could. MIMEBase is provided primarily as a convenient base class for more specific MIME-aware subclasses.
_maintype is the Content-Type major type (e.g. text or image), and _subtype is the Content-Type minor type (e.g. plain or gif). _params is a parameter key/value dictionary and is passed directly to Message.add_header.

The MIMEBase class always adds a Content-Type header (based on _maintype, _subtype, and _params), and a MIME-Version header (always set to 1.0).

class email.mime.nonmultipart.MIMENonMultipart

    Module: email.mime.nonmultipart
    A subclass of MIMEBase, this is an intermediate base class for MIME messages that are not multipart. The primary purpose of this class is to prevent the use of the attach() method, which only makes sense for multipart messages. If attach() is called, a MultipartConversionError exception is raised.

class email.mime.multipart.MIMEMultipart(_subtype='mixed', boundary=None, _subparts=None, **_params)

    Module: email.mime.multipart
    A subclass of MIMEBase, this is an intermediate base class for MIME messages that are multipart. Optional _subtype defaults to mixed, but can be used to specify the subtype of the message. A Content-Type header of multipart/_subtype will be added to the message object. A MIME-Version header will also be added.

    Optional boundary is the multipart boundary string. When None (the default), the boundary is calculated when needed (for example, when the message is serialized).

    _subparts is a sequence of initial subparts for the payload. It must be possible to convert this sequence to a list. You can always attach new subparts to the message by using the Message.attach method.

    Additional parameters for the Content-Type header are taken from the keyword arguments, or passed into the _params argument, which is a keyword dictionary.

class email.mime.application.MIMEApplication(_data, _subtype='octet-stream', _encoder=email.encoders.encode_base64, **_params)

    Module: email.mime.application
    A subclass of MIMENonMultipart, the MIMEApplication class is used to represent MIME message objects of major type application. _data is a string containing the raw byte data. Optional _subtype specifies the MIME subtype and defaults to octet-stream.

    Optional _encoder is a callable (i.e. function) which will perform the actual encoding of the data for transport. This callable takes one argument, which is the MIMEApplication instance. It should use get_payload() and set_payload() to change the payload to encoded form. It should also add any Content-Transfer-Encoding or other headers to the message object as necessary. The default encoding is base64. See the email.encoders module for a list of the built-in encoders.

    _params are passed straight through to the base class constructor.

class email.mime.audio.MIMEAudio(_audiodata, _subtype=None, _encoder=email.encoders.encode_base64, **_params)

    Module: email.mime.audio
    A subclass of MIMENonMultipart, the MIMEAudio class is used to create MIME message objects of major type audio. _audiodata is a string containing the raw audio data. If this data can be decoded by the standard Python module sndhdr, then the subtype will be automatically included in the Content-Type header. Otherwise you can explicitly specify the audio subtype via the _subtype argument. If the minor type could not be guessed and _subtype was not given, then TypeError is raised.

    Optional _encoder is a callable (i.e. function) which will perform the actual encoding of the audio data for transport. This callable takes one argument, which is the MIMEAudio instance. It should use get_payload() and set_payload() to change the payload to encoded form. It should also add any Content-Transfer-Encoding or other headers to the message object as necessary. The default encoding is base64. See the email.encoders module for a list of the built-in encoders.

    _params are passed straight through to the base class constructor.
**email.mime.image.MIMEImage (_imagedata, _subtype=None, _encoder=email.encoders.encode_base64, **_params)**

Module: email.mime.image

A subclass of MIMENonMultipart, the MIMEImage class is used to create MIME message objects of major type image. _imagedata is a string containing the raw image data. If this data can be decoded by the standard Python module imghdr, then the subtype will be automatically included in the Content-Type header. Otherwise you can explicitly specify the image subtype via the _subtype argument. If the minor type could not be guessed and _subtype was not given, then TypeError is raised.

Optional _encoder is a callable (i.e. function) which will perform the actual encoding of the image data for transport. This callable takes one argument, which is the MIMEImage instance. It should use get_payload() and set_payload() to change the payload to encoded form. It should also add any Content-Transfer-Encoding or other headers to the message object as necessary. The default encoding is base64. See the email.encoders module for a list of the built-in encoders.

_params are passed straight through to the MIMEBase constructor.

**email.mime.message.MIMEMessage (_msg, _subtype='rfc822')**

Module: email.mime.message

A subclass of MIMENonMultipart, the MIMEMessage class is used to create MIME objects of main type message. _msg is used as the payload, and must be an instance of class Message (or a subclass thereof), otherwise a TypeError is raised.

Optional _subtype sets the subtype of the message; it defaults to rfc822.

**email.mime.text.MIMEText (_text, _subtype='plain', _charset=None)**

Module: email.mime.text

A subclass of MIMENonMultipart, the MIMEText class is used to create MIME objects of major type text. _text is the string for the payload. _subtype is the minor type and defaults to plain. _charset is the character set of the text and is passed as an argument to the MIMENonMultipart constructor; it defaults to us-ascii if the string contains only ascii codepoints, and utf-8 otherwise.

Unless the _charset argument is explicitly set to None, the MIMEText object created will have both a Content-Type header with a charset parameter, and a Content-Transfer-Encoding header. This means that a subsequent set_payload call will not result in an encoded payload, even if a charset is passed in the set_payload command. You can “reset” this behavior by deleting the Content-Transfer-Encoding header, after which a set_payload call will automatically encode the new payload (and add a new Content-Transfer-Encoding header).

### 19.1.7 email.header: Internationalized headers

RFC 2822 is the base standard that describes the format of email messages. It derives from the older RFC 822 standard which came into widespread use at a time when most email was composed of ASCII characters only. RFC 2822 is a specification written assuming email messages contain only 7-bit ASCII characters.

Of course, as email has been deployed worldwide, it has become internationalized, such that language specific character sets can now be used in email messages. The base standard still requires email messages to be transferred using only 7-bit ASCII characters, so a slew of RFCs have been written describing how to encode email containing non-ASCII characters into RFC 2822-compliant format. These RFCs include RFC 2045, RFC 2046, RFC 2047, and RFC 2231. The email package supports these standards in its email.header and email.charset modules.

If you want to include non-ASCII characters in your email headers, say in the Subject or To fields, you should use the Header class and assign the field in the Message object to an instance of Header instead of using a string for the header value. Import the Header class from the email.header module. For example:

```python
>>> from email.message import Message
>>> from email.header import Header
```
>>> msg = Message()
>>> h = Header('p\xf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> msg.as_string()
'Subject: =?iso-8859-1?q?p=F6stal?=\n\n'

Notice here how we wanted the Subject field to contain a non-ASCII character? We did this by creating a Header instance and passing in the character set that the byte string was encoded in. When the subsequent Message instance was flattened, the Subject field was properly RFC 2047 encoded. MIME-aware mail readers would show this header using the embedded ISO-8859-1 character.

Here is the Header class description:

class email.header.Header(s=None, charset=None, maxlinelen=None, header_name=None, continuation_ws=' ', errors='strict')

Create a MIME-compliant header that can contain strings in different character sets.

Optional s is the initial header value. If None (the default), the initial header value is not set. You can later append to the header with append() method calls. s may be an instance of bytes or str, but see the append() documentation for semantics.

Optional charset serves two purposes: it has the same meaning as the charset argument to the append() method. It also sets the default character set for all subsequent append() calls that omit the charset argument. If charset is not provided in the constructor (the default), the us-ascii character set is used both as s’s initial charset and as the default for subsequent append() calls.

The maximum line length can be specified explicitly via maxlinelen. For splitting the first line to a shorter value (to account for the field header which isn’t included in s, e.g. Subject) pass in the name of the field in header_name. The default maxlinelen is 76, and the default value for header_name is None, meaning it is not taken into account for the first line of a long, split header.

Optional continuation_ws must be RFC 2822-compliant folding whitespace, and is usually either a space or a hard tab character. This character will be prepended to continuation lines. continuation_ws defaults to a single space character.

Optional errors is passed straight through to the append() method.

append(s, charset=None, errors='strict')

Append the string s to the MIME header.

Optional charset, if given, should be a Charset instance (see email.charset) or the name of a character set, which will be converted to a Charset instance. A value of None (the default) means that the charset given in the constructor is used.

s may be an instance of bytes or str. If it is an instance of bytes, then charset is the encoding of that byte string, and a UnicodeError will be raised if the string cannot be decoded with that character set.

If s is an instance of str, then charset is a hint specifying the character set of the characters in the string.

In either case, when producing an RFC 2822-compliant header using

RFC 2047 rules, the string will be encoded using the output codec of the charset. If the string cannot be encoded using the output codec, a UnicodeError will be raised.

Optional errors is passed as the errors argument to the decode call if s is a byte string.

encode(splitchars='; \t', maxlinelen=None, linesep='\n')

Encode a message header into an RFC-compliant format, possibly wrapping long lines and encapsulating non-ASCII parts in base64 or quoted-printable encodings.

Optional splitchars is a string containing characters which should be given extra weight by the splitting algorithm during normal header wrapping. This is in very rough support of RFC 2822’s ‘higher level syntactic breaks’: split points preceded by a splitchar are preferred during line splitting, with the characters preferred in the order in which they appear in the string. Space and tab may be included in the string to indicate whether preference should be given to one over the other as a split point when
The Python Library Reference, Release 3.3.3

other split chars do not appear in the line being split. Splitchars does not affect RFC 2047 encoded
lines.
maxlinelen, if given, overrides the instance’s value for the maximum line length.
linesep specifies the characters used to separate the lines of the folded header. It defaults to the most
useful value for Python application code (\n), but \r\n can be specified in order to produce headers
with RFC-compliant line separators. Changed in version 3.2: Added the linesep argument.
The Header class also provides a number of methods to support standard operators and built-in functions.
__str__()
Returns an approximation of the Header as a string, using an unlimited line length. All pieces are
converted to unicode using the specified encoding and joined together appropriately. Any pieces with a
charset of ’unknown-8bit’ are decoded as ASCII using the ’replace’ error handler. Changed
in version 3.2: Added handling for the ’unknown-8bit’ charset.
__eq__(other)
This method allows you to compare two Header instances for equality.
__ne__(other)
This method allows you to compare two Header instances for inequality.
The email.header module also provides the following convenient functions.
email.header.decode_header(header)
Decode a message header value without converting the character set. The header value is in header.
This function returns a list of (decoded_string, charset) pairs containing each of the decoded
parts of the header. charset is None for non-encoded parts of the header, otherwise a lower case string
containing the name of the character set specified in the encoded string.
Here’s an example:
>>> from email.header import decode_header
[(b’p\xf6stal’, ’iso-8859-1’)]
email.header.make_header(decoded_seq, maxlinelen=None, header_name=None, continuation_ws=’ ‘)
Create a Header instance from a sequence of pairs as returned by decode_header().
decode_header() takes a header value string and returns a sequence of pairs of the format
(decoded_string, charset) where charset is the name of the character set.
This function takes one of those sequence of pairs and returns a Header instance. Optional maxlinelen,
header_name, and continuation_ws are as in the Header constructor.

19.1.8 email.charset: Representing character sets
This module provides a class Charset for representing character sets and character set conversions in email
messages, as well as a character set registry and several convenience methods for manipulating this registry.
Instances of Charset are used in several other modules within the email package.
Import this class from the email.charset module.
class email.charset.Charset(input_charset=DEFAULT_CHARSET)
Map character sets to their email properties.
This class provides information about the requirements imposed on email for a specific character set. It also
provides convenience routines for converting between character sets, given the availability of the applicable
codecs. Given a character set, it will do its best to provide information on how to use that character set in
an email message in an RFC-compliant way.

19.1. email — An email and MIME handling package

745


Certain character sets must be encoded with quoted-printable or base64 when used in email headers or bodies. Certain character sets must be converted outright, and are not allowed in email.

Optional `input_charset` is as described below; it is always coerced to lower case. After being alias normalized it is also used as a lookup into the registry of character sets to find out the header encoding, body encoding, and output conversion codec to be used for the character set. For example, if `input_charset` is `iso-8859-1`, then headers and bodies will be encoded using quoted-printable and no output conversion codec is necessary. If `input_charset` is `euc-jp`, then headers will be encoded with base64, bodies will not be encoded, but output text will be converted from the `euc-jp` character set to the `iso-2022-jp` character set.

`Charset` instances have the following data attributes:

- **input_charset**
  The initial character set specified. Common aliases are converted to their official email names (e.g. `latin_1` is converted to `iso-8859-1`). Defaults to 7-bit `us-ascii`.

- **header_encoding**
  If the character set must be encoded before it can be used in an email header, this attribute will be set to `Charset.QP` (for quoted-printable), `Charset.BASE64` (for base64 encoding), or `Charset.SHORTEST` for the shortest of QP or BASE64 encoding. Otherwise, it will be `None`.

- **body_encoding**
  Same as `header_encoding`, but describes the encoding for the mail message’s body, which indeed may be different than the header encoding. `Charset.SHORTEST` is not allowed for `body_encoding`.

- **output_charset**
  Some character sets must be converted before they can be used in email headers or bodies. If the `input_charset` is one of them, this attribute will contain the name of the character set output will be converted to. Otherwise, it will be `None`.

- **input_codec**
  The name of the Python codec used to convert the `input_charset` to Unicode. If no conversion codec is necessary, this attribute will be `None`.

- **output_codec**
  The name of the Python codec used to convert Unicode to the `output_charset`. If no conversion codec is necessary, this attribute will have the same value as the `input_codec`.

`Charset` instances also have the following methods:

- **get_body_encoding()**
  Return the content transfer encoding used for body encoding.
  
  This is either the string `quoted-printable` or `base64` depending on the encoding used, or it is a function, in which case you should call the function with a single argument, the Message object being encoded. The function should then set the `Content-Transfer-Encoding` header itself to whatever is appropriate.

  Returns the string `quoted-printable` if `body_encoding` is QP, returns the string `base64` if `body_encoding` is BASE64, and returns the string `7bit` otherwise.

- **get_output_charset()**
  Return the output character set.

  This is the `output_charset` attribute if that is not `None`, otherwise it is `input_charset`.

- **header_encode**(string)
  Header-encode the string `string`.

  The type of encoding (base64 or quoted-printable) will be based on the `header_encoding` attribute.

- **header_encode_lines**(string, maxlengths)
  Header-encode a `string` by converting it first to bytes.
This is similar to `header_encode()` except that the string is fit into maximum line lengths as given by the argument `maxlengths`, which must be an iterator: each element returned from this iterator will provide the next maximum line length.

```python
body_encode(string)
```

Body-encode the string `string`.

The type of encoding (base64 or quoted-printable) will be based on the `body_encoding` attribute.

The `Charset` class also provides a number of methods to support standard operations and built-in functions.

```python
__str__()
__repr__()
__eq__ (other)
__ne__ (other)
```

This method allows you to compare two `Charset` instances for equality.

This method allows you to compare two `Charset` instances for inequality.

The `email.charset` module also provides the following functions for adding new entries to the global character set, alias, and codec registries:

```python
email.charset.add_charset(charset, header_enc=None, body_enc=None, output_charset=None)
```

Add character properties to the global registry.

`charset` is the input character set, and must be the canonical name of a character set.

Optional `header_enc` and `body_enc` is either `Charset.QP` for quoted-printable, `Charset.BASE64` for base64 encoding, `Charset.SHORTEST` for the shortest of quoted-printable or base64 encoding, or `None` for no encoding. `SHORTEST` is only valid for `header_enc`. The default is `None` for no encoding.

Optional `output_charset` is the character set that the output should be in. Conversions will proceed from input charset, to Unicode, to the output charset when the method `Charset.convert()` is called. The default is to output in the same character set as the input.

Both `input_charset` and `output_charset` must have Unicode codec entries in the module’s character set-to-codec mapping; use `add_codec()` to add codecs the module does not know about. See the `codecs` module’s documentation for more information.

The global character set registry is kept in the module global dictionary `CHARSETS`.

```python
email.charset.add_alias(alias, canonical)
```

Add a character set alias. `alias` is the alias name, e.g. `latin-1`. `canonical` is the character set’s canonical name, e.g. `iso-8859-1`.

The global charset alias registry is kept in the module global dictionary `ALIASES`.

```python
email.charset.add_codec(charset, codecname)
```

Add a codec that map characters in the given character set to and from Unicode.

`charset` is the canonical name of a character set. `codecname` is the name of a Python codec, as appropriate for the second argument to the `str's encode()` method

---

19.1.9 `email.encoders`: Encoders

When creating `Message` objects from scratch, you often need to encode the payloads for transport through compliant mail servers. This is especially true for `image/*` and `text/*` type messages containing binary data.

The `email` package provides some convenient encodings in its `encoders` module. These encoders are actually used by the `MIMEAudio` and `MIMEImage` class constructors to provide default encodings. All encoder functions take exactly one argument, the message object to encode. They usually extract the payload, encode it, and reset the payload to this newly encoded value. They should also set the `Content-Transfer-Encoding` header as appropriate.
Note that these functions are not meaningful for a multipart message. They must be applied to individual subparts instead, and will raise a `TypeError` if passed a message whose type is multipart.

Here are the encoding functions provided:

```python
e-mail.encoders.encode_quopri(msg)
```

Encodes the payload into quoted-printable form and sets the `Content-Transfer-Encoding` header to quoted-printable³. This is a good encoding to use when most of your payload is normal printable data, but contains a few unprintable characters.

```python
e-mail.encoders.encode_base64(msg)
```

Encodes the payload into base64 form and sets the `Content-Transfer-Encoding` header to base64. This is a good encoding to use when most of your payload is unprintable data since it is a more compact form than quoted-printable. The drawback of base64 encoding is that it renders the text non-human readable.

```python
e-mail.encoders.encode_7or8bit(msg)
```

This doesn’t actually modify the message’s payload, but it does set the `Content-Transfer-Encoding` header to either 7bit or 8bit as appropriate, based on the payload data.

```python
e-mail.encoders.encode_noop(msg)
```

This does nothing; it doesn’t even set the `Content-Transfer-Encoding` header.

### 19.1.10 `email.errors`: Exception and Defect classes

The following exception classes are defined in the `email.errors` module:

#### exception `email.errors.MessageError`

This is the base class for all exceptions that the `email` package can raise. It is derived from the standard `Exception` class and defines no additional methods.

#### exception `email.errors.MessageParseError`

This is the base class for exceptions raised by the `Parser` class. It is derived from `MessageError`.

#### exception `email.errors.HeaderParseError`

Raised under some error conditions when parsing the RFC 2822 headers of a message, this class is derived from `MessageParseError`. It can be raised from the `Parser.parse` or `Parser.parsestr` methods.

Situations where it can be raised include finding an envelope header after the first RFC 2822 header of the message, finding a continuation line before the first RFC 2822 header is found, or finding a line in the headers which is neither a header or a continuation line.

#### exception `email.errors.BoundaryError`

Raised under some error conditions when parsing the RFC 2822 headers of a message, this class is derived from `MessageParseError`. It can be raised from the `Parser.parse` or `Parser.parsestr` methods.

Situations where it can be raised include not being able to find the starting or terminating boundary in a multipart/* message when strict parsing is used.

#### exception `email.errors.MultipartConversionError`

Raised when a payload is added to a `Message` object using `add_payload()`, but the payload is already a scalar and the message’s `Content-Type` main type is not either multipart or missing. `MultipartConversionError` multiply inherits from `MessageError` and the built-in `TypeError`.

Since `Message.add_payload()` is deprecated, this exception is rarely raised in practice. However the exception may also be raised if the `attach()` method is called on an instance of a class derived from `MIMENonMultipart` (e.g. `MIMEImage`).

Here’s the list of the defects that the `FeedParser` can find while parsing messages. Note that the defects are added to the message where the problem was found, so for example, if a message nested inside a

³ Note that encoding with `encode_quopri()` also encodes all tabs and space characters in the data.
**multipart/alternative** had a malformed header, that nested message object would have a defect, but the containing messages would not.

All defect classes are subclassed from `email.errors.MessageDefect`, but this class is *not* an exception!

- **NoBoundaryInMultipartDefect** – A message claimed to be a multipart, but had no `boundary` parameter.
- **StartBoundaryNotFoundDefect** – The start boundary claimed in the `Content-Type` header was never found.
- **CloseBoundaryNotFoundDefect** – A start boundary was found, but no corresponding close boundary was ever found. New in version 3.3.
- **FirstHeaderLineIsContinuationDefect** – The message had a continuation line as its first header line.
- **MisplacedEnvelopeHeaderDefect** - A “Unix From” header was found in the middle of a header block.
- **MissingHeaderBodySeparatorDefect** - A line was found while parsing headers that had no leading white space but contained no `:`. Parsing continues assuming that the line represents the first line of the body. New in version 3.3.
- **MalformedHeaderDefect** – A header was found that was missing a colon, or was otherwise malformed. Deprecated since version 3.3: This defect has not been used for several Python versions.
- **MultipartInvariantViolationDefect** – A message claimed to be a `multipart`, but no sub-parts were found. Note that when a message has this defect, its `is_multipart()` method may return false even though its content type claims to be `multipart`.
- **InvalidBase64PaddingDefect** – When decoding a block of base64 encoded bytes, the padding was not correct. Enough padding is added to perform the decode, but the resulting decoded bytes may be invalid.
- **InvalidBase64CharactersDefect** – When decoding a block of base64 encoded bytes, characters outside the base64 alphabet were encountered. The characters are ignored, but the resulting decoded bytes may be invalid.

### 19.1.11 email.utils: Miscellaneous utilities

There are several useful utilities provided in the `email.utils` module:

**email.utils.quote**(str)
Return a new string with backslashes in `str` replaced by two backslashes, and double quotes replaced by backslash-double quote.

**email.utils.unquote**(str)
Return a new string which is an unquoted version of `str`. If `str` ends and begins with double quotes, they are stripped off. Likewise if `str` ends and begins with angle brackets, they are stripped off.

**email.utils.parseaddr**(address)
Parse address – which should be the value of some address-containing field such as `To` or `Cc` – into its constituent `realname` and `email address` parts. Returns a tuple of that information, unless the parse fails, in which case a 2-tuple of `('', '')` is returned.

**email.utils.formataddr**(pair, charset=’utf-8’)
The inverse of `parseaddr()`, this takes a 2-tuple of the form `(realname, email_address)` and returns the string value suitable for a `To` or `Cc` header. If the first element of `pair` is false, then the second element is returned unmodified.

Optional `charset` is the character set that will be used in the RFC 2047 encoding of the `realname` if the `realname` contains non-ASCII characters. Can be an instance of `str` or a `Charset`. Defaults to `utf-8`. Changed in version 3.3: Added the `charset` option.
email.utils.getaddresses (fieldvalues)
This method returns a list of 2-tuples of the form returned by parseaddr(). fieldvalues is a sequence of header field values as might be returned by Message.get_all. Here’s a simple example that gets all the recipients of a message:

```python
from email.utils import getaddresses
tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

email.utils.parsedate (date)
Attempts to parse a date according to the rules in RFC 2822. However, some mailers don’t follow that format as specified, so parsedate() tries to guess correctly in such cases. date is a string containing an RFC 2822 date, such as "Mon, 20 Nov 1995 19:12:08 -0500". If it succeeds in parsing the date, parsedate() returns a 9-tuple that can be passed directly to time.mktime(); otherwise None will be returned. Note that indexes 6, 7, and 8 of the result tuple are not usable.

email.utils.parsedate_tz (date)
Performs the same function as parsedate(), but returns either None or a 10-tuple; the first 9 elements make up a tuple that can be passed directly to time.mktime(), and the tenth is the offset of the date’s timezone from UTC (which is the official term for Greenwich Mean Time). If the input string has no timezone, the last element of the tuple returned is None. Note that indexes 6, 7, and 8 of the result tuple are not usable.

email.utils.parsedate_to_datetime (date)
The inverse of format_datetime(). Performs the same function as parsedate(), but on success returns a datetime. If the input date has a timezone of -0000, the datetime will be a naive datetime, and if the date is conforming to the RFCs it will represent a time in UTC but with no indication of the actual source timezone of the message the date comes from. If the input date has any other valid timezone offset, the datetime will be an aware datetime with the corresponding a timezone tzinfo. New in version 3.3.

email.utils.mktime_tz (tuple)
Turn a 10-tuple as returned by parsedate_tz() into a UTC timestamp. It the timezone item in the tuple is None, assume local time. Minor deficiency: mktime_tz() interprets the first 8 elements of tuple as a local time and then compensates for the timezone difference. This may yield a slight error around changes in daylight savings time, though not worth worrying about for common use.

email.utils.formatdate (timeval=None, localtime=False, usegmt=False)
Returns a date string as per RFC 2822, e.g.:

```
Fri, 09 Nov 2001 01:08:47 -0000
```

Optional timeval if given is a floating point time value as accepted by time.gmtime() and time.localtime(), otherwise the current time is used.

Optional localtime is a flag that when True, interprets timeval, and returns a date relative to the local timezone instead of UTC, properly taking daylight savings time into account. The default is False meaning UTC is used.

Optional usegmt is a flag that when True, outputs a date string with the timezone as an ascii string GMT, rather than a numeric -0000. This is needed for some protocols (such as HTTP). This only applies when localtime is False. The default is False.

email.utils.format_datetime (dt, usegmt=False)
Like formatdate, but the input is a datetime instance. If it is a naive datetime, it is assumed to

---

4 Note that the sign of the timezone offset is the opposite of the sign of the time.timezone variable for the same timezone; the latter variable follows the POSIX standard while this module follows RFC 2822.
be “UTC with no information about the source timezone”, and the conventional -0000 is used for the timezone. If it is an aware datetime, then the numeric timezone offset is used. If it is an aware timezone with offset zero, then usegmt may be set to True, in which case the string GMT is used instead of the numeric timezone offset. This provides a way to generate standards conformant HTTP date headers. New in version 3.3.

email.utils.localtime(dt=None)
Return local time as an aware datetime object. If called without arguments, return current time. Otherwise dt argument should be a datetime instance, and it is converted to the local time zone according to the system time zone database. If dt is naive (that is, dt.tzinfo is None), it is assumed to be in local time. In this case, a positive or zero value for isdst causes localtime to presume initially that summer time (for example, Daylight Saving Time) is or is not (respectively) in effect for the specified time. A negative value for isdst causes the localtime to attempt to divine whether summer time is in effect for the specified time. New in version 3.3.

email.utils.make_msgid(idstring=None, domain=None)
Returns a string suitable for an RFC 2822-compliant Message-ID header. Optional idstring if given, is a string used to strengthen the uniqueness of the message id. Optional domain if given provides the portion of the msgid after the ‘@’. The default is the local hostname. It is not normally necessary to override this default, but may be useful certain cases, such as a constructing distributed system that uses a consistent domain name across multiple hosts. Changed in version 3.2: Added the domain keyword.

email.utils.decode_rfc2231(s)
Decode the string s according to RFC 2231.

email.utils.encode_rfc2231(s, charset=None, language=None)
Encode the string s according to RFC 2231. Optional charset and language, if given is the character set name and language name to use. If neither is given, s is returned as-is. If charset is given but language is not, the string is encoded using the empty string for language.

email.utils.collapse_rfc2231_value(value, errors='replace', fallback_charset='us-ascii')
When a header parameter is encoded in RFC 2231 format, Message.get_param may return a 3-tuple containing the character set, language, and value. collapse_rfc2231_value() turns this into a unicode string. Optional errors is passed to the errors argument of str's encode() method; it defaults to ‘replace’. Optional fallback_charset specifies the character set to use if the one in the RFC 2231 header is not known by Python; it defaults to ‘us-ascii’.

For convenience, if the value passed to collapse_rfc2231_value() is not a tuple, it should be a string and it is returned unquoted.

email.utils.decode_params(params)
Decode parameters list according to RFC 2231. params is a sequence of 2-tuples containing elements of the form (content-type, string-value).

19.1.12 email.iterators: Iterators

Iterating over a message object tree is fairly easy with the Message.walk method. The email.iterators module provides some useful higher level iterations over message object trees.

email.iterators.body_line_iterator(msg, decode=False)
This iterates over all the payloads in all the subparts of msg, returning the string payloads line-by-line. It skips over all the subpart headers, and it skips over any subpart with a payload that isn’t a Python string. This is somewhat equivalent to reading the flat text representation of the message from a file using readline(), skipping over all the intervening headers.

Optional decode is passed through to Message.get_payload.

email.iterators.typed_subpart_iterator(msg, maintype='text', subtype=None)
This iterates over all the subparts of msg, returning only those subparts that match the MIME type specified by maintype and subtype.
Note that *subtype* is optional; if omitted, then subpart MIME type matching is done only with the main type. *main_type* is optional too; it defaults to *text*.

Thus, by default *typed_subpart_iterator()* returns each subpart that has a MIME type of *text/*.

The following function has been added as a useful debugging tool. It should *not* be considered part of the supported public interface for the package.

```
email.iterators._structure(msg, fp=None, level=0, include_default=False)
```

Prints an indented representation of the content types of the message object structure. For example:

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
  text/plain
  multipart/digest
    message/rfc822
      text/plain
    message/rfc822
      text/plain
    message/rfc822
      text/plain
    message/rfc822
      text/plain

text/plain
```

Optional *fp* is a file-like object to print the output to. It must be suitable for Python’s *print()* function. *level* is used internally. *include_default*, if true, prints the default type as well.

### 19.1.13 email: Examples

Here are a few examples of how to use the *email* package to read, write, and send simple email messages, as well as more complex MIME messages.

First, let’s see how to create and send a simple text message:

```python
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.mime.text import MIMEText

# Open a plain text file for reading. For this example, assume that
# the text file contains only ASCII characters.
fp = open(textfile, 'rb')

# Create a text/plain message
msg = MIMEText(fp.read())
fp.close()

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = 'The contents of %s' % textfile
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server.
```
s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()

And parsing RFC822 headers can easily be done by the parse(filename) or parsestr(message_as_string) methods of the Parser() class:

```python
# Import the email modules we’ll need
from email.parser import Parser

# If the e-mail headers are in a file, uncomment this line:
# headers = Parser().parse(open(messagefile, 'r'))

# Or for parsing headers in a string, use:
headers = Parser().parsestr('From: <user@example.com>
To: <someone_else@example.com>
Subject: Test message

Body would go here')

# Now the header items can be accessed as a dictionary:
print('To: %s' % headers['to'])
print('From: %s' % headers['from'])
print('Subject: %s' % headers['subject'])
```

Here’s an example of how to send a MIME message containing a bunch of family pictures that may be residing in a directory:

```python
# Import smtplib for the actual sending function
import smtplib

# Here are the email package modules we’ll need
from email.mime.image import MIMEImage
from email.mime.multipart import MIMEMultipart

COMMASPACE = ', '

# Create the container (outer) email message.
msg = MIMEMultipart()
msg['Subject'] = 'Our family reunion'
# me == the sender’s email address
# family = the list of all recipients’ email addresses
msg['From'] = me
msg['To'] = COMMASPACE.join(family)
msg.preamble = 'Our family reunion'

# Assume we know that the image files are all in PNG format
for file in pngfiles:
    # Open the files in binary mode. Let the MIMEImage class automatically
    # guess the specific image type.
    fp = open(file, 'rb')
    img = MIMEImage(fp.read())
    fp.close()
    msg.attach(img)

# Send the email via our own SMTP server.
s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()
```
Here’s an example of how to send the entire contents of a directory as an email message:

```python
#!/usr/bin/env python3

"""Send the contents of a directory as a MIME message.""

import os
import sys
import smtplib

# For guessing MIME type based on file name extension
import mimetypes

from optparse import OptionParser
from email import encoders
from email.message import Message
from email.mime.audio import MIMEAudio
from email.mime.base import MIMEBase
from email.mime.image import MIMEImage
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

COMMASPACE = ', '

def main():
    parser = OptionParser(usage=""
    Send the contents of a directory as a MIME message.

    Usage: %prog [options]

    Unless the -o option is given, the email is sent by forwarding to your local
    SMTP server, which then does the normal delivery process. Your local machine
    must be running an SMTP server.
    ""
    parser.add_option('-d', '--directory',
                      type='string', action='store',
                      help="""Mail the contents of the specified directory,
                      otherwise use the current directory. Only the regular
                      files in the directory are sent, and we don’t recurse to
                      subdirectories.""
                      )
    parser.add_option('-o', '--output',
                      type='string', action='store',
                      help="""Print the composed message to FILE instead of
                      sending the message to the SMTP server.""
                      metavar='FILE',
                      )
    parser.add_option('-s', '--sender',
                      type='string', action='store',
                      help='The value of the From: header (required)
                      ')
    parser.add_option('-r', '--recipient',
                      type='string', action='append',
                      help='A To: header value (at least one required)
                      ')

    opts, args = parser.parse_args()
    if not opts.sender or not opts.recipients:
        parser.print_help()
        sys.exit(1)
    directory = opts.directory
```

5 Thanks to Matthew Dixon Cowles for the original inspiration and examples.
if not directory:
    directory = '.

# Create the enclosing (outer) message
outer = MIMEMultipart()
outer['Subject'] = 'Contents of directory %s' % os.path.abspath(directory)
outer['To'] = COMMASPACE.join(opts.recipients)
outer['From'] = opts.sender
outer.preamble = 'You will not see this in a MIME-aware mail reader.

for filename in os.listdir(directory):
    path = os.path.join(directory, filename)
    if not os.path.isfile(path):
        continue

    # Guess the content type based on the file's extension. Encoding
    # will be ignored, although we should check for simple things like
    # gzip'd or compressed files.
    ctype, encoding = mimetypes.guess_type(path)
    if ctype is None or encoding is not None:
        # No guess could be made, or the file is encoded (compressed), so
        # use a generic bag-of-bits type.
        ctype = 'application/octet-stream'
    maintype, subtype = ctype.split('/', 1)
    if maintype == 'text':
        fp = open(path)
        # Note: we should handle calculating the charset
        msg = MIMEText(fp.read(), _subtype=subtype)
        fp.close()
    elif maintype == 'image':
        fp = open(path, 'rb')
        msg = MIMEImage(fp.read(), _subtype=subtype)
        fp.close()
    elif maintype == 'audio':
        fp = open(path, 'rb')
        msg = MIMEAudio(fp.read(), _subtype=subtype)
        fp.close()
    else:
        fp = open(path, 'rb')
        msg = MIMEBase(maintype, subtype)
        msg.set_payload(fp.read())
        fp.close()
        # Encode the payload using Base64
        encoders.encode_base64(msg)

    # Set the filename parameter
    msg.add_header('Content-Disposition', 'attachment', filename=filename)
    outer.attach(msg)

# Now send or store the message
composed = outer.as_string()
if opts.output:
    fp = open(opts.output, 'w')
    fp.write(composed)
    fp.close()
else:
    s = smtplib.SMTP('localhost')
    s.sendmail(opts.sender, opts.recipients, composed)
    s.quit()

if __name__ == '__main__':
Here's an example of how to unpack a MIME message like the one above, into a directory of files:

```
#!/usr/bin/env python3

"""Unpack a MIME message into a directory of files."""

import os
import sys
import email
import errno
import mimetypes

from optparse import OptionParser

def main():
    parser = OptionParser(usage="
    Unpack a MIME message into a directory of files.
    Usage: %prog [options] msgfile
    ""
    parser.add_option(’-d’, ’--directory’,
                      type=’string’, action=’store’,
                      help="""Unpack the MIME message into the named
directory, which will be created if it doesn’t already exist."""
    )

    opts, args = parser.parse_args()
    if not opts.directory:
        parser.print_help()
        sys.exit(1)

    try:
        msgfile = args[0]
    except IndexError:
        parser.print_help()
        sys.exit(1)

    try:
        os.mkdir(opts.directory)
    except OSError as e:
        # Ignore directory exists error
        if e.errno != errno.EEXIST:
            raise

    fp = open(msgfile)
    msg = email.message_from_file(fp)
    fp.close()

    counter = 1
    for part in msg.walk():
        # multipart/* are just containers
        if part.get_content_maintype() == ’multipart’:
            continue
        # Applications should really sanitize the given filename so that an
        # email message can’t be used to overwrite important files
        filename = part.get_filename()
        if not filename:
            continue
    ```
ext = mimetypes.guess_extension(part.get_content_type())
if not ext:
    # Use a generic bag-of-bits extension
    ext = '.bin'
filename = 'part-%03d%s' % (counter, ext)
counter += 1
fp = open(os.path.join(opts.directory, filename), 'wb')
fp.write(part.get_payload(decode=True))
fp.close()

if __name__ == '__main__':
    main()

Here’s an example of how to create an HTML message with an alternative plain text version:

```
#!/usr/bin/env python3

import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

# me == my email address
# you == recipient's email address
me = "my@email.com"
you = "your@email.com"

# Create message container - the correct MIME type is multipart/alternative.
msg = MIMEMultipart('alternative')
msg['Subject'] = "Link"
msg['From'] = me
msg['To'] = you

# Create the body of the message (a plain-text and an HTML version).
text = "Hi!

How are you?

Here is the link you wanted:

http://www.python.org"
html = "<html><head></head><body><p>Hi!<br>
How are you?<br>
Here is the <a href="http://www.python.org">link</a> you wanted.</p></body></html>"

# Record the MIME types of both parts - text/plain and text/html.
part1 = MIMEText(text, 'plain')
part2 = MIMEText(html, 'html')

# Attach parts into message container.
# According to RFC 2046, the last part of a multipart message, in this case
# the HTML message, is best and preferred.
msg.attach(part1)
msg.attach(part2)
```

---

19.1. `email` — An email and MIME handling package

---

6 Contributed by Martin Matejek.
# Send the message via local SMTP server.
s = smtplib.SMTP('localhost')
# sendmail function takes 3 arguments: sender’s address, recipient’s address
# and message to send – here it is sent as one string.
s.sendmail(me, you, msg.as_string())
s.quit()

See Also:
Module `smtplib` SMTP protocol client
Module `nntplib` NNTP protocol client

## 19.1.14 Package History

This table describes the release history of the email package, corresponding to the version of Python that the package was released with. For purposes of this document, when you see a note about change or added versions, these refer to the Python version the change was made in, *not* the email package version. This table also describes the Python compatibility of each version of the package.

<table>
<thead>
<tr>
<th>email version</th>
<th>distributed with</th>
<th>compatible with</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.x</td>
<td>Python 2.2.0 to Python 2.2.1</td>
<td>no longer supported</td>
</tr>
<tr>
<td>2.5</td>
<td>Python 2.2.2+ and Python 2.3</td>
<td>Python 2.1 to 2.5</td>
</tr>
<tr>
<td>3.0</td>
<td>Python 2.4</td>
<td>Python 2.3 to 2.5</td>
</tr>
<tr>
<td>4.0</td>
<td>Python 2.5</td>
<td>Python 2.3 to 2.5</td>
</tr>
<tr>
<td>5.0</td>
<td>Python 3.0 and Python 3.1</td>
<td>Python 3.0 to 3.2</td>
</tr>
<tr>
<td>5.1</td>
<td>Python 3.2</td>
<td>Python 3.0 to 3.2</td>
</tr>
</tbody>
</table>

Here are the major differences between email version 5.1 and version 5.0:

- It is once again possible to parse messages containing non-ASCII bytes, and to reproduce such messages if the data containing the non-ASCII bytes is not modified.
- New functions `message_from_bytes()` and `message_from_binary_file()`, and new classes `BytesFeedParser` and `BytesParser` allow binary message data to be parsed into model objects.
- Given bytes input to the model, `get_payload()` will by default decode a message body that has a `Content-Transfer-Encoding` of 8bit using the charset specified in the MIME headers and return the resulting string.
- Given bytes input to the model, `Generator` will convert message bodies that have a `Content-Transfer-Encoding` of 8bit to instead have a 7bit `Content-Transfer-Encoding`.
- New class `BytesGenerator` produces bytes as output, preserving any unchanged non-ASCII data that was present in the input used to build the model, including message bodies with a `Content-Transfer-Encoding` of 8bit.

Here are the major differences between email version 5.0 and version 4:

- All operations are on unicode strings. Text inputs must be strings, text outputs are strings. Outputs are limited to the ASCII character set and so can be encoded to ASCII for transmission. Inputs are also limited to ASCII; this is an acknowledged limitation of email 5.0 and means it can only be used to parse email that is 7bit clean.

Here are the major differences between email version 4 and version 3:

- All modules have been renamed according to PEP 8 standards. For example, the version 3 module `email.Message` was renamed to `email.message` in version 4.
- A new subpackage `email.mime` was added and all the version 3 `email.MIME*` modules were renamed and situated into the `email.mime` subpackage. For example, the version 3 module `email.MIMEText` was renamed to `email.mime.text`.

*Note that the version 3 names will continue to work until Python 2.6.*

- The `email.mime.application` module was added, which contains the `MIMEApplication` class.
• Methods that were deprecated in version 3 have been removed. These include 
  Generator.__call__(), Message.get_type(), Message.get_main_type(), 
  Message.get_subtype().

• Fixes have been added for RFC 2231 support which can change some of the return types for 
  Message.get_param and friends. Under some circumstances, values which used to return a 3-tuple 
  now return simple strings (specifically, if all extended parameter segments were unencoded, there is no 
  language and charset designation expected, so the return type is now a simple string). Also, %-decoding 
  used to be done for both encoded and unencoded segments; this decoding is now done only for encoded 
  segments.

Here are the major differences between email version 3 and version 2:

• The FeedParser class was introduced, and the Parser class was implemented in terms of the 
  FeedParser. All parsing therefore is non-strict, and parsing will make a best effort never to raise an 
  exception. Problems found while parsing messages are stored in the message’s defect attribute.

• All aspects of the API which raised DeprecationWarnings in version 2 have been removed. 
  These include the _encoder argument to the MIMEText constructor, the Message.add_payload() 
  method, the Utils.dump_address_pair() function, and the functions Utils.decode() and 
  Utils.encode().

• New DeprecationWarnings have been added to: Generator.__call__(), 
  Message.get_type(), Message.get_main_type(), Message.get_subtype(), and 
  the strict argument to the Parser class. These are expected to be removed in future versions.

• Support for Pythons earlier than 2.3 has been removed.

Here are the differences between email version 2 and version 1:

• The email.Header and email.Charset modules have been added.

• The pickle format for Message instances has changed. Since this was never (and still isn’t) formally 
  defined, this isn’t considered a backward incompatibility. However if your application pickles and unpickles 
  Message instances, be aware that in email version 2, Message instances now have private variables 
  _charset and _default_type.

• Several methods in the Message class have been deprecated, or their signatures changed. Also, many new 
  methods have been added. See the documentation for the Message class for details. The changes should 
  be completely backward compatible.

• The object structure has changed in the face of message/rfc822 content types. In email version 1, 
  such a type would be represented by a scalar payload, i.e. the container message’s is_multipart() 
  returned false, get_payload() was not a list object, but a single Message instance. 
  This structure was inconsistent with the rest of the package, so the object representation for 
  message/rfc822 content types was changed. In email version 2, the container does return True 
  from is_multipart(), and get_payload() returns a list containing a single Message item. 
  Note that this is one place that backward compatibility could not be completely maintained. However, if 
  you’re already testing the return type of get_payload(), you should be fine. You just need to make sure 
  your code doesn’t do a set_payload() with a Message instance on a container with a content type of 
  message/rfc822.

• The Parser constructor’s strict argument was added, and its parse() and parsestr() methods grew 
  a headersonly argument. The strict flag was also added to functions email.message_from_file() 
  and email.message_from_string().

• Generator.__call__() is deprecated; use Generator.flatten instead. The Generator class 
  has also grown the clone() method.

• The DecodedGenerator class in the email.generator module was added.

• The intermediate base classes MIMENonMultipart and MIMEMultipart have been added, and inter- 
  posed in the class hierarchy for most of the other MIME-related derived classes.
• The _encoder argument to the MIMEText constructor has been deprecated. Encoding now happens implicitly based on the _charset argument.

• The following functions in the email.Utils module have been deprecated: dump_address_pairs(), decode(), and encode(). The following functions have been added to the module: make_msid(), decode_rfc2231(), encode_rfc2231(), and decode_params().

• The non-public function email.Iterators._structure() was added.

19.1.15 Differences from mimelib

The email package was originally prototyped as a separate library called mimelib. Changes have been made so that method names are more consistent, and some methods or modules have either been added or removed. The semantics of some of the methods have also changed. For the most part, any functionality available in mimelib is still available in the email package, albeit often in a different way. Backward compatibility between the mimelib package and the email package was not a priority.

Here is a brief description of the differences between the mimelib and the email packages, along with hints on how to port your applications.

Of course, the most visible difference between the two packages is that the package name has been changed to email. In addition, the top-level package has the following differences:

• messageFromString() has been renamed to message_from_string().
• messageFromFile() has been renamed to message_from_file().

The Message class has the following differences:

• The method asString() was renamed to as_string().
• The method ismultipart() was renamed to is_multipart().
• The get_payload() method has grown a decode optional argument.
• The method getall() was renamed to get_all().
• The method addheader() was renamed to add_header().
• The method gettype() was renamed to get_type().
• The method getmaintype() was renamed to get_main_type().
• The method getsubtype() was renamed to get_subtype().
• The method getparams() was renamed to get_params(). Also, whereas getparams() returned a list of strings, get_params() returns a list of 2-tuples, effectively the key/value pairs of the parameters, split on the '=' sign.
• The method getparam() was renamed to get_param().
• The method getcharsets() was renamed to get_charsets().
• The method getfilename() was renamed to get_filename().
• The method getboundary() was renamed to get_boundary().
• The method setboundary() was renamed to set_boundary().
• The method getdecodedpayload() was removed. To get similar functionality, pass the value 1 to the decode flag of the get_payload() method.
• The method getpayloadastext() was removed. Similar functionality is supported by the DecodedGenerator class in the email.generator module.
• The method getbodyastext() was removed. You can get similar functionality by creating an iterator with typed_subpart_iterator() in the email.iterators module.
The **Parser** class has no differences in its public interface. It does have some additional smarts to recognize `message/delivery-status` type messages, which it represents as a `Message` instance containing separate `Message` subparts for each header block in the delivery status notification.

The **Generator** class has no differences in its public interface. There is a new class in the `email.generator` module though, called `DecodedGenerator` which provides most of the functionality previously available in the `Message.get_payload(text)` method.

The following modules and classes have been changed:

- The `MIMEBase` class constructor arguments `_major` and `_minor` have changed to `_maintype` and `_subtype` respectively.
- The `Image` class/module has been renamed to `MIMEImage`. The `_minor` argument has been renamed to `_subtype`.
- The `Text` class/module has been renamed to `MIMEText`. The `_minor` argument has been renamed to `_subtype`.
- The `MessageRFC822` class/module has been renamed to `MIMEMessage`. Note that an earlier version of `mimelib` called this class/module `RFC822`, but that clashed with the Python standard library module `rfc822` on some case-insensitive file systems.

Also, the `MIMEMessage` class now represents any kind of MIME message with main type `message`. It takes an optional argument `_subtype` which is used to set the MIME subtype. `_subtype` defaults to `rfc822`.

`mimelib` provided some utility functions in its `address` and `date` modules. All of these functions have been moved to the `email.utils` module.

The `MsgReader` class/module has been removed. Its functionality is most closely supported in the `body_line_iterator()` function in the `email.iterators` module.

## 19.2 `json` — JSON encoder and decoder

**JSON (JavaScript Object Notation)**, specified by

*RFC 4627*, is a lightweight data interchange format based on a subset of JavaScript syntax (ECMA-262 3rd edition).

`json` exposes an API familiar to users of the standard library `marshal` and `pickle` modules.

**Encoding basic Python object hierarchies:**

```python
>>> import json
>>> json.dumps(['foo', {'bar': {'baz': (None, 1.0, 2)}}])
'[
  "foo",
  {
    "bar": {
      "baz": [null, 1.0, 2]
    }
  }
]
```

```python
>>> print(json.dumps("\"foo\b\")
"\"foo\b"
```

```python
>>> print(json.dumps("\u1234")
"\u1234"
```

```python
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
```

```python
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'[
  "streaming API"
]
```

**Compact encoding:**

---

7 Delivery Status Notifications (DSN) are defined in [RFC 1894](https://tools.ietf.org/html/rfc1894).

19.2. `json` — JSON encoder and decoder 761
The Python Library Reference, Release 3.3.3

>>> import json
>>> json.dumps([1,2,3,{’4’: 5, ’6’: 7}], separators=(’,’, ’:’))
’[1,2,3,{"4":5,"6":7}]’
Pretty printing:
>>> import json
>>> print(json.dumps({’4’: 5, ’6’: 7}, sort_keys=True,
...
indent=4, separators=(’,’, ’: ’)))
{
"4": 5,
"6": 7
}
Decoding JSON:
>>> import json
>>> json.loads(’["foo", {"bar":["baz", null, 1.0, 2]}]’)
[’foo’, {’bar’: [’baz’, None, 1.0, 2]}]
>>> json.loads(’"\\"foo\\bar"’)
’"foo\x08ar’
>>> from io import StringIO
>>> io = StringIO(’["streaming API"]’)
>>> json.load(io)
[’streaming API’]
Specializing JSON object decoding:
>>> import json
>>> def as_complex(dct):
...
if ’__complex__’ in dct:
...
return complex(dct[’real’], dct[’imag’])
...
return dct
...
>>> json.loads(’{"__complex__": true, "real": 1, "imag": 2}’,
...
object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads(’1.1’, parse_float=decimal.Decimal)
Decimal(’1.1’)
Extending JSONEncoder:
>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
...
def default(self, obj):
...
if isinstance(obj, complex):
...
return [obj.real, obj.imag]
...
# Let the base class default method raise the TypeError
...
return json.JSONEncoder.default(self, obj)
...
>>> json.dumps(2 + 1j, cls=ComplexEncoder)
’[2.0, 1.0]’
>>> ComplexEncoder().encode(2 + 1j)
’[2.0, 1.0]’
>>> list(ComplexEncoder().iterencode(2 + 1j))
[’[2.0’, ’, 1.0’, ’]’]
Using json.tool from the shell to validate and pretty-print:
$ echo ’{"json":"obj"}’ | python -mjson.tool
{
"json": "obj"

762

Chapter 19. Internet Data Handling


$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)

Note: JSON is a subset of YAML 1.2. The JSON produced by this module’s default settings (in particular, the default separators value) is also a subset of YAML 1.0 and 1.1. This module can thus also be used as a YAML serializer.

### 19.2.1 Basic Usage

```python
json.dump(obj, fp, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)
```

Serialize `obj` as a JSON formatted stream to `fp` (a `write()`-supporting file-like object) using this conversion table.

If `skipkeys` is `True` (default: `False`), then dict keys that are not of a basic type (`str`, `int`, `float`, `bool`, `None`) will be skipped instead of raising a `TypeError`.

The `json` module always produces `str` objects, not `bytes` objects. Therefore, `fp.write()` must support `str` input.

If `ensure_ascii` is `True` (the default), the output is guaranteed to have all incoming non-ASCII characters escaped. If `ensure_ascii` is `False`, these characters will be output as-is.

If `check_circular` is `False` (default: `True`), then the circular reference check for container types will be skipped and a circular reference will result in an `OverflowError` (or worse).

If `allow_nan` is `False` (default: `True`), then it will be a `ValueError` to serialize out of range `float` values (`nan`, `inf`, `-inf`) in strict compliance of the JSON specification, instead of using the JavaScript equivalents (`NaN`, `Infinity`, `-Infinity`).

If `indent` is a non-negative integer or string, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0, negative, or `""` will only insert newlines. None (the default) selects the most compact representation. Using a positive integer indent indents that many spaces per level. If `indent` is a string (such as `"\t"`), that string is used to indent each level. Changed in version 3.2: Allow strings for `indent` in addition to integers.

Note: Since the default item separator is `', '`, the output might include trailing whitespace when `indent` is specified. You can use `separators=(' ', ',', ' : ')` to avoid this.

If `separators` is an `(item_separator, dict_separator)` tuple, then it will be used instead of the default `(' ', ',', ' : ') separators`. `(' ', ',', ' : ')` is the most compact JSON representation.

`default(obj)` is a function that should return a serializable version of `obj` or raise `TypeError`. The default simply raises `TypeError`.

If `sort_keys` is `True` (default: `False`), then the output of dictionaries will be sorted by key.

To use a custom `JSONEncoder` subclass (e.g. one that overrides the `default()` method to serialize additional types), specify it with the `cls` kwarg; otherwise `JSONEncoder` is used.

```python
json.dumps(obj, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)
```

Serialize `obj` to a JSON formatted `str` using this conversion table. The arguments have the same meaning as in `dump()`.

Note: Unlike `pickle` and `marshal`, JSON is not a framed protocol, so trying to serialize multiple objects with repeated calls to `dump()` using the same `fp` will result in an invalid JSON file.
Note: Keys in key/value pairs of JSON are always of the type `str`. When a dictionary is converted into JSON, all the keys of the dictionary are coerced to strings. As a result of this, if a dictionary is converted into JSON and then back into a dictionary, the dictionary may not equal the original one. That is, `loads(dumps(x)) != x` if `x` has non-string keys.

```python
json.load(fp, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)
```

Deserializes `fp (a `.read()`-supporting `file-like object` containing a JSON document) to a Python object using this conversion table.

`object_hook` is an optional function that will be called with the result of any object literal decoded (a `dict`). The return value of `object_hook` will be used instead of the `dict`. This feature can be used to implement custom decoders (e.g. JSON-RPC class hinting).

`object_pairs_hook` is an optional function that will be called with the result of any object literal decoded with an ordered list of pairs. The return value of `object_pairs_hook` will be used instead of the `dict`. This feature can be used to implement custom decoders that rely on the order that the key and value pairs are decoded (for example, `collections.OrderedDict()` will remember the order of insertion). If `object_hook` is also defined, the `object_pairs_hook` takes priority. Changed in version 3.1: Added support for `object_pairs_hook`, `parse_float`, if specified, will be called with the string of every JSON float to be decoded. By default, this is equivalent to `float(num_str)`. This can be used to use another datatype or parser for JSON floats (e.g. `decimal.Decimal`).

`parse_int`, if specified, will be called with the string of every JSON int to be decoded. By default, this is equivalent to `int(num_str)`. This can be used to use another datatype or parser for JSON integers (e.g. `float`).

`parse_constant`, if specified, will be called with one of the following strings: ‘-Infinity’, ‘Infinity’, ‘NaN’. This can be used to raise an exception if invalid JSON numbers are encountered. Changed in version 3.1: `parse_constant` doesn’t get called on ‘null’, ‘true’, ‘false’ anymore. To use a custom `JSONDecoder` subclass, specify it with the `cls` kwarg; otherwise `JSONDecoder` is used. Additional keyword arguments will be passed to the constructor of the class.

```python
json.loads(s, encoding=None, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)
```

Deserializes `s (a `str` instance containing a JSON document) to a Python object using this conversion table. The other arguments have the same meaning as in `load()`, except `encoding` which is ignored and deprecated.

### 19.2.2 Encoders and Decoders

```python
class json.JSONDecoder (object_hook=None, parse_float=None, parse_int=None, parse_constant=None, strict=True, object_pairs_hook=None)
```

Simple JSON decoder.

<table>
<thead>
<tr>
<th>JSON</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>dict</td>
</tr>
<tr>
<td>array</td>
<td>list</td>
</tr>
<tr>
<td>string</td>
<td>str</td>
</tr>
<tr>
<td>number (int)</td>
<td>int</td>
</tr>
<tr>
<td>number (real)</td>
<td>float</td>
</tr>
<tr>
<td>true</td>
<td>True</td>
</tr>
<tr>
<td>false</td>
<td>False</td>
</tr>
<tr>
<td>null</td>
<td>None</td>
</tr>
</tbody>
</table>

It also understands `NaN`, `Infinity`, and `-Infinity` as their corresponding `float` values, which is outside the JSON spec.

`object_hook`, if specified, will be called with the result of every JSON object decoded and its return value will be used in place of the given `dict`. This can be used to provide custom deserializations (e.g. to support JSON-RPC class hinting).
object_pairs_hook, if specified will be called with the result of every JSON object decoded with an ordered list of pairs. The return value of object_pairs_hook will be used instead of the dict. This feature can be used to implement custom decoders that rely on the order that the key and value pairs are decoded (for example, collections.OrderedDict() will remember the order of insertion). If object_hook is also defined, the object_pairs_hook takes priority. Changed in version 3.1: Added support for object_pairs_hook.

parse_float, if specified, will be called with the string of every JSON float to be decoded. By default, this is equivalent to float(num_str). This can be used to use another datatype or parser for JSON floats (e.g. decimal.Decimal).

parse_int, if specified, will be called with the string of every JSON int to be decoded. By default, this is equivalent to int(num_str). This can be used to use another datatype or parser for JSON integers (e.g. float).

parse_constant, if specified, will be called with one of the following strings: '-Infinity', 'Infinity', 'NaN', 'null', 'true', 'false'. This can be used to raise an exception if invalid JSON numbers are encountered.

If strict is False (True is the default), then control characters will be allowed inside strings. Control characters in this context are those with character codes in the 0-31 range, including \t (tab), \n, \r and \0.

If the data being deserialized is not a valid JSON document, a ValueError will be raised.

decode (s)
Return the Python representation of s (a str instance containing a JSON document)

raw_decode (s)
Decode a JSON document from s (a str beginning with a JSON document) and return a 2-tuple of the Python representation and the index in s where the document ended.

This can be used to decode a JSON document from a string that may have extraneous data at the end.

class json.JSONEncoder (skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None)
Extensible JSON encoder for Python data structures.

<table>
<thead>
<tr>
<th>Python</th>
<th>JSON</th>
</tr>
</thead>
<tbody>
<tr>
<td>dict</td>
<td>object</td>
</tr>
<tr>
<td>list, tuple</td>
<td>array</td>
</tr>
<tr>
<td>str</td>
<td>string</td>
</tr>
<tr>
<td>int, float</td>
<td>number</td>
</tr>
<tr>
<td>True</td>
<td>true</td>
</tr>
<tr>
<td>False</td>
<td>false</td>
</tr>
<tr>
<td>None</td>
<td>null</td>
</tr>
</tbody>
</table>

Supports the following objects and types by default:

To extend this to recognize other objects, subclass and implement a default() method with another method that returns a serializable object for o if possible, otherwise it should call the superclass implementation (to raise TypeError).

If skipkeys is False (the default), then it is a TypeError to attempt encoding of keys that are not str, int, float or None. If skipkeys is True, such items are simply skipped.

If ensure_ascii is True (the default), the output is guaranteed to have all incoming non-ASCII characters escaped. If ensure_ascii is False, these characters will be output as-is.

If check_circular is True (the default), then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise, no such check takes place.

If allow_nan is True (the default), then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is True (default False), then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

19.2. json — JSON encoder and decoder 765
If `indent` is a non-negative integer or string, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0, negative, or "" will only insert newlines. `None` (the default) selects the most compact representation. Using a positive integer indent indents that many spaces per level. If `indent` is a string (such as "\t"), that string is used to indent each level. Changed in version 3.2: Allow strings for `indent` in addition to integers.

**Note:** Since the default item separator is ‘, ‘, the output might include trailing whitespace when `indent` is specified. You can use `separators=('', ': ')` to avoid this.

If specified, `separators` should be an `(item_separator, key_separator)` tuple. The default is `('', ': ')`. To get the most compact JSON representation, you should specify `('', ': ')` to eliminate whitespace.

If specified, `default` is a function that gets called for objects that can’t otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`.

```python
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return json.JSONEncoder.default(self, o)
```

`encode(o)`

Return a JSON string representation of a Python data structure, `o`. For example:

```python
>>> json.JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}"
```

`iterencode(o)`

Encode the given object, `o`, and yield each string representation as available. For example:

```python
for chunk in json.JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

### 19.2.3 Standard Compliance

The JSON format is specified by RFC 4627. This section details this module’s level of compliance with the RFC. For simplicity, `JSONEncoder` and `JSONDecoder` subclasses, and parameters other than those explicitly mentioned, are not considered.

This module does not comply with the RFC in a strict fashion, implementing some extensions that are valid JavaScript but not valid JSON. In particular:

- Top-level non-object, non-array values are accepted and output;
- Infinite and NaN number values are accepted and output;
- Repeated names within an object are accepted, and only the value of the last name-value pair is used.

Since the RFC permits RFC-compliant parsers to accept input texts that are not RFC-compliant, this module’s deserializer is technically RFC-compliant under default settings.
Character Encodings

The RFC recommends that JSON be represented using either UTF-8, UTF-16, or UTF-32, with UTF-8 being the default.

As permitted, though not required, by the RFC, this module’s serializer sets `ensure_ascii=True` by default, thus escaping the output so that the resulting strings only contain ASCII characters.

Other than the `ensure_ascii` parameter, this module is defined strictly in terms of conversion between Python objects and Unicode strings, and thus does not otherwise address the issue of character encodings.

Top-level Non-Object, Non-Array Values

The RFC specifies that the top-level value of a JSON text must be either a JSON object or array (Python `dict` or `list`). This module’s deserializer also accepts input texts consisting solely of a JSON null, boolean, number, or string value:

```python
>>> just_a_json_string = "'spam and eggs'"   # Not by itself a valid JSON text
>>> json.loads(just_a_json_string)
'spam and eggs'
```

This module itself does not include a way to request that such input texts be regarded as illegal. Likewise, this module’s serializer also accepts single Python `None`, `bool`, numeric, and `str` values as input and will generate output texts consisting solely of a top-level JSON null, boolean, number, or string value without raising an exception:

```python
>>> neither_a_list_nor_a_dict = "'spam and eggs'"
>>> json.dumps(neither_a_list_nor_a_dict)   # The result is not a valid JSON text
"'spam and eggs'"
```

This module’s serializer does not itself include a way to enforce the aforementioned constraint.

Infinite and NaN Number Values

The RFC does not permit the representation of infinite or NaN number values. Despite that, by default, this module accepts and outputs `Infinity`, `-Infinity`, and `NaN` as if they were valid JSON number literal values:

```python
>>> # Neither of these calls raises an exception, but the results are not valid JSON
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # Same when deserializing
>>> json.loads('-Infinity')
-inf
>>> json.loads('NaN')
nan
```

In the serializer, the `allow_nan` parameter can be used to alter this behavior. In the deserializer, the `parse_constant` parameter can be used to alter this behavior.

Repeated Names Within an Object

The RFC specifies that the names within a JSON object should be unique, but does not specify how repeated names in JSON objects should be handled. By default, this module does not raise an exception; instead, it ignores all but the last name-value pair for a given name:

```python
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{'x': 3}
```
The object_pairs_hook parameter can be used to alter this behavior.

## 19.3 mailcap — Mailcap file handling

Source code: Lib/mailcap.py

Mailcap files are used to configure how MIME-aware applications such as mail readers and Web browsers react to files with different MIME types. (The name “mailcap” is derived from the phrase “mail capability”.) For example, a mailcap file might contain a line like video/mpeg; xmpeg %s. Then, if the user encounters an email message or Web document with the MIME type video/mpeg, %s will be replaced by a filename (usually one belonging to a temporary file) and the xmpeg program can be automatically started to view the file.

The mailcap format is documented in RFC 1524, “A User Agent Configuration Mechanism For Multimedia Mail Format Information,” but is not an Internet standard. However, mailcap files are supported on most Unix systems.

```python
import mailcap

d = mailcap.getcaps()
mailcap.findmatch(d, 'video/mpeg', filename='tmp1223')
```

In a mailcap file, the “test” field can optionally be specified to test some external condition (such as the machine architecture, or the window system in use) to determine whether or not the mailcap line applies. findmatch() will automatically check such conditions and skip the entry if the check fails.

```python
mailcap.getcaps()
```

Returns a dictionary mapping MIME types to a list of mailcap file entries. This dictionary must be passed to the findmatch() function. An entry is stored as a list of dictionaries, but it shouldn’t be necessary to know the details of this representation.

The information is derived from all of the mailcap files found on the system. Settings in the user’s mailcap file $HOME/.mailcap will override settings in the system mailcap files /etc/mailcap, /usr/etc/mailcap, and /usr/local/etc/mailcap.

An example usage:

```python
>>> import mailcap
>>> d = mailcap.getcaps()
>>> mailcap.findmatch(d, 'video/mpeg', filename='tmp1223')
('xmpeg tmp1223', {'view': 'xmpeg %s'})
```

## 19.4 mailbox — Manipulate mailboxes in various formats

This module defines two classes, Mailbox and Message, for accessing and manipulating on-disk mailboxes and the messages they contain. Mailbox offers a dictionary-like mapping from keys to messages. Message
extends the `email.message` module's `Message` class with format-specific state and behavior. Supported mailbox formats are Maildir, mbox, MH, Babyl, and MMDF.

See Also:

Module `email` Represent and manipulate messages.

19.4.1 Mailbox objects

class mailbox.Mailbox

A mailbox, which may be inspected and modified.

The `Mailbox` class defines an interface and is not intended to be instantiated. Instead, format-specific subclasses should inherit from `Mailbox` and your code should instantiate a particular subclass.

The `Mailbox` interface is dictionary-like, with small keys corresponding to messages. Keys are issued by the `Mailbox` instance with which they will be used and are only meaningful to that `Mailbox` instance. A key continues to identify a message even if the corresponding message is modified, such as by replacing it with another message.

Messages may be added to a `Mailbox` instance using the set-like method `add()` and removed using a `del` statement or the set-like methods `remove()` and `discard()`.

`Mailbox` interface semantics differ from dictionary semantics in some noteworthy ways. Each time a message is requested, a new representation (typically a `Message` instance) is generated based upon the current state of the mailbox. Similarly, when a message is added to a `Mailbox` instance, the provided message representation's contents are copied. In neither case is a reference to the message representation kept by the `Mailbox` instance.

The default `Mailbox` iterator iterates over message representations, not keys as the default dictionary iterator does. Moreover, modification of a mailbox during iteration is safe and well-defined. Messages added to the mailbox after an iterator is created will not be seen by the iterator. Messages removed from the mailbox before the iterator yields them will be silently skipped, though using a key from an iterator may result in a `KeyError` exception if the corresponding message is subsequently removed.

**Warning:** Be very cautious when modifying mailboxes that might be simultaneously changed by some other process. The safest mailbox format to use for such tasks is Maildir; try to avoid using single-file formats such as mbox for concurrent writing. If you’re modifying a mailbox, you *must* lock it by calling the `lock()` and `unlock()` methods *before* reading any messages in the file or making any changes by adding or deleting a message. Failing to lock the mailbox runs the risk of losing messages or corrupting the entire mailbox.

`Mailbox` instances have the following methods:

`add(message)`
Add message to the mailbox and return the key that has been assigned to it.

Parameter `message` may be a `Message` instance, an `email.message.Message` instance, a string, a byte string, or a file-like object (which should be open in binary mode). If `message` is an instance of the appropriate format-specific `Message` subclass (e.g., if it’s an `mboxMessage` instance and this is an `mbox` instance), its format-specific information is used. Otherwise, reasonable defaults for format-specific information are used. Changed in version 3.2: Support for binary input was added.

`remove(key)`
`__delitem__(key)`
`discard(key)`
Delete the message corresponding to `key` from the mailbox.

If no such message exists, a `KeyError` exception is raised if the method was called as `remove()` or `__delitem__()` but no exception is raised if the method was called as `discard()`. The behavior of `discard()` may be preferred if the underlying mailbox format supports concurrent modification by other processes.
__setitem__(key, message)
Replace the message corresponding to key with message. Raise a KeyError exception if no message already corresponds to key.

As with add(), parameter message may be a Message instance, an email.message.Message instance, a string, a byte string, or a file-like object (which should be open in binary mode). If message is an instance of the appropriate format-specific Message subclass (e.g., if it’s an mboxMessage instance and this is an mbox instance), its format-specific information is used. Otherwise, the format-specific information of the message that currently corresponds to key is left unchanged.

iterkeys()
keys()
Return an iterator over all keys if called as iterkeys() or return a list of keys if called as keys().

itervalues()
__iter__()
values()
Return an iterator over representations of all messages if called as itervalues() or __iter__() or return a list of such representations if called as values(). The messages are represented as instances of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized.

Note: The behavior of __iter__() is unlike that of dictionaries, which iterate over keys.

iteritems()
items()
Return an iterator over (key, message) pairs, where key is a key and message is a message representation, if called as iteritems() or return a list of such pairs if called as items(). The messages are represented as instances of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized.

get(key, default=None)
__getitem__(key)
Return a representation of the message corresponding to key. If no such message exists, default is returned if the method was called as get() and a KeyError exception is raised if the method was called as __getitem__(). The message is represented as an instance of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized.

get_message(key)
Return a representation of the message corresponding to key as an instance of the appropriate format-specific Message subclass, or raise a KeyError exception if no such message exists.

get_bytes(key)
Return a byte representation of the message corresponding to key, or raise a KeyError exception if no such message exists. New in version 3.2.

get_string(key)
Return a string representation of the message corresponding to key, or raise a KeyError exception if no such message exists. The message is processed through email.message.Message to convert it to a 7bit clean representation.

get_file(key)
Return a file-like representation of the message corresponding to key, or raise a KeyError exception if no such message exists. The file-like object behaves as if open in binary mode. This file should be closed once it is no longer needed. Changed in version 3.2: The file object really is a binary file; previously it was incorrectly returned in text mode. Also, the file-like object now supports the context manager protocol: you can use a with statement to automatically close it.

Note: Unlike other representations of messages, file-like representations are not necessarily independent of the Mailbox instance that created them or of the underlying mailbox. More specific
__contains__(key)
    Return True if key corresponds to a message, False otherwise.

__len__()
    Return a count of messages in the mailbox.

clear()
    Delete all messages from the mailbox.

pop(key, default=None)
    Return a representation of the message corresponding to key and delete the message. If no such message exists, return default. The message is represented as an instance of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized.

popitem()
    Return an arbitrary (key, message) pair, where key is a key and message is a message representation, and delete the corresponding message. If the mailbox is empty, raise a KeyError exception. The message is represented as an instance of the appropriate format-specific Message subclass unless a custom message factory was specified when the Mailbox instance was initialized.

update(arg)
    Parameter arg should be a key-to-message mapping or an iterable of (key, message) pairs. Updates the mailbox so that, for each given key and message, the message corresponding to key is set to message as if by using __setitem__(). As with __setitem__(), each key must already correspond to a message in the mailbox or else a KeyError exception will be raised, so in general it is incorrect for arg to be a Mailbox instance.

    Note: Unlike with dictionaries, keyword arguments are not supported.

flush()
    Write any pending changes to the filesystem. For some Mailbox subclasses, changes are always written immediately and flush() does nothing, but you should still make a habit of calling this method.

lock()
    Acquire an exclusive advisory lock on the mailbox so that other processes know not to modify it. An ExternalClashError is raised if the lock is not available. The particular locking mechanisms used depend upon the mailbox format. You should always lock the mailbox before making any modifications to its contents.

unlock()
    Release the lock on the mailbox, if any.

close()
    Flush the mailbox, unlock it if necessary, and close any open files. For some Mailbox subclasses, this method does nothing.

Maildir

class Maildir(dirname, factory=None, create=True)
    A subclass of Mailbox for mailboxes in Maildir format. Parameter factory is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If factory is None, MaildirMessage is used as the default message representation. If create is True, the mailbox is created if it does not exist.

    It is for historical reasons that dirname is named as such rather than path.
Maildir is a directory-based mailbox format invented for the qmail mail transfer agent and now widely supported by other programs. Messages in a Maildir mailbox are stored in separate files within a common directory structure. This design allows Maildir mailboxes to be accessed and modified by multiple unrelated programs without data corruption, so file locking is unnecessary.

Maildir mailboxes contain three subdirectories, namely: tmp, new, and cur. Messages are created momentarily in the tmp subdirectory and then moved to the new subdirectory to finalize delivery. A mail user agent may subsequently move the message to the cur subdirectory and store information about the state of the message in a special “info” section appended to its file name.

Folders of the style introduced by the Courier mail transfer agent are also supported. Any subdirectory of the main mailbox is considered a folder if ‘.’ is the first character in its name. Folder names are represented by Maildir without the leading ‘.’. Each folder is itself a Maildir mailbox but should not contain other folders. Instead, a logical nesting is indicated using ‘.’ to delimit levels, e.g., “Archived.2005.07”.

Note: The Maildir specification requires the use of a colon (’ :’) in certain message file names. However, some operating systems do not permit this character in file names. If you wish to use a Maildir-like format on such an operating system, you should specify another character to use instead. The exclamation point (‘ !’) is a popular choice. For example:

```python
import mailbox
mailbox.Maildir.colon = '!
```

The colon attribute may also be set on a per-instance basis.

Maildir instances have all of the methods of Mailbox in addition to the following:

- **list_folders()**
  - Return a list of the names of all folders.

- **get_folder(folder)**
  - Return a Maildir instance representing the folder whose name is folder. A NoSuchMailboxError exception is raised if the folder does not exist.

- **add_folder(folder)**
  - Create a folder whose name is folder and return a Maildir instance representing it.

- **remove_folder(folder)**
  - Delete the folder whose name is folder. If the folder contains any messages, a NotEmptyError exception will be raised and the folder will not be deleted.

- **clean()**
  - Delete temporary files from the mailbox that have not been accessed in the last 36 hours. The Maildir specification says that mail-reading programs should do this occasionally.

Some Mailbox methods implemented by Maildir deserve special remarks:

- **add(message)**
- **__setitem__(key, message)**
- **update(arg)**

  **Warning:** These methods generate unique file names based upon the current process ID. When using multiple threads, undetected name clashes may occur and cause corruption of the mailbox unless threads are coordinated to avoid using these methods to manipulate the same mailbox simultaneously.

- **flush()**
  - All changes to Maildir mailboxes are immediately applied, so this method does nothing.

- **lock()**
- **unlock()**
  - Maildir mailboxes do not support (or require) locking, so these methods do nothing.
close()
    Maildir instances do not keep any open files and the underlying mailboxes do not support locking,
    so this method does nothing.

get_file(key)
    Depending upon the host platform, it may not be possible to modify or remove the underlying message
    while the returned file remains open.

See Also:

maildir man page from qmail  The original specification of the format.

Using maildir format  Notes on Maildir by its inventor. Includes an updated name-creation scheme and details
    on “info” semantics.

maildir man page from Courier  Another specification of the format. Describes a common extension for supporting folders.

mbox

class mailbox.mbox(path, factory=None, create=True)
    A subclass of Mailbox for mailboxes in mbox format. Parameter factory is a callable object that accepts
    a file-like message representation (which behaves as if opened in binary mode) and returns a custom rep-
    resentation. If factory is None, mboxMessage is used as the default message representation. If create is
    True, the mailbox is created if it does not exist.

    The mbox format is the classic format for storing mail on Unix systems. All messages in an mbox mailbox
    are stored in a single file with the beginning of each message indicated by a line whose first five characters
    are “From ”.

    Several variations of the mbox format exist to address perceived shortcomings in the original. In the interest
    of compatibility, mbox implements the original format, which is sometimes referred to as mboxo. This
    means that the Content-Length header, if present, is ignored and that any occurrences of “From ” at
    the beginning of a line in a message body are transformed to “>From ” when storing the message, although
    occurrences of “>From ” are not transformed to “From ” when reading the message.

    Some Mailbox methods implemented by mbox deserve special remarks:

    get_file(key)
        Using the file after calling flush() or close() on the mbox instance may yield unpredictable
        results or raise an exception.

    lock()
    unlock()
        Three locking mechanisms are used—dot locking and, if available, the flock() and lockf()
        system calls.

See Also:

mbox man page from qmail  A specification of the format and its variations.

mbox man page from tin  Another specification of the format, with details on locking.

Configuring Netscape Mail on Unix: Why The Content-Length Format is Bad  An argument for using the
    original mbox format rather than a variation.

“mbox” is a family of several mutually incompatible mailbox formats  A history of mbox variations.

MH

class mailbox.MH(path, factory=None, create=True)
    A subclass of Mailbox for mailboxes in MH format. Parameter factory is a callable object that accepts a
file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If factory is None, MHMessage is used as the default message representation. If create is True, the mailbox is created if it does not exist.

MH is a directory-based mailbox format invented for the MH Message Handling System, a mail user agent. Each message in an MH mailbox resides in its own file. An MH mailbox may contain other MH mailboxes (called folders) in addition to messages. Folders may be nested indefinitely. MH mailboxes also support sequences, which are named lists used to logically group messages without moving them to sub-folders. Sequences are defined in a file called .mh_sequences in each folder.

The MH class manipulates MH mailboxes, but it does not attempt to emulate all of mh’s behaviors. In particular, it does not modify and is not affected by the context or .mh_profile files that are used by mh to store its state and configuration.

MH instances have all of the methods of Mailbox in addition to the following:

- list_folders()  
  Return a list of the names of all folders.

- get_folder(folder)  
  Return an MH instance representing the folder whose name is folder. A NoSuchMailboxError exception is raised if the folder does not exist.

- add_folder(folder)  
  Create a folder whose name is folder and return an MH instance representing it.

- remove_folder(folder)  
  Delete the folder whose name is folder. If the folder contains any messages, aNotEmptyError exception will be raised and the folder will not be deleted.

- get_sequences()  
  Return a dictionary of sequence names mapped to key lists. If there are no sequences, the empty dictionary is returned.

- set_sequences(sequences)  
  Re-define the sequences that exist in the mailbox based upon sequences, a dictionary of names mapped to key lists, like returned by get_sequences().

- pack()  
  Rename messages in the mailbox as necessary to eliminate gaps in numbering. Entries in the sequences list are updated correspondingly.

Note: Already-issued keys are invalidated by this operation and should not be subsequently used.

Some Mailbox methods implemented by MH deserve special remarks:

- remove(key)  
- __delitem__(key)  
- discard(key)  
  These methods immediately delete the message. The MH convention of marking a message for deletion by prepending a comma to its name is not used.

- lock()  
- unlock()  
  Three locking mechanisms are used—dot locking and, if available, the flock() and lockf() system calls. For MH mailboxes, locking the mailbox means locking the .mh_sequences file and, only for the duration of any operations that affect them, locking individual message files.

- get_file(key)  
  Depending upon the host platform, it may not be possible to remove the underlying message while the returned file remains open.

- flush()  
  All changes to MH mailboxes are immediately applied, so this method does nothing.
close()

MH instances do not keep any open files, so this method is equivalent to unlock().

See Also:

nmh - Message Handling System  Home page of nmh, an updated version of the original mh.

MH & nmh: Email for Users & Programmers  A GPL-licensed book on mh and nmh, with some information on the mailbox format.

Babyl

class mailbox.Babyl (path, factory=None, create=True)

A subclass of Mailbox for mailboxes in Babyl format. Parameter factory is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If factory is None, BabylMessage is used as the default message representation. If create is True, the mailbox is created if it does not exist.

Babyl is a single-file mailbox format used by the Rmail mail user agent included with Emacs. The beginning of a message is indicated by a line containing the two characters Control-Underscore (\'\037\') and Control-L (\'\014\'). The end of a message is indicated by the start of the next message or, in the case of the last message, a line containing a Control-Underscore (\'\037\') character.

Messages in a Babyl mailbox have two sets of headers, original headers and so-called visible headers. Visible headers are typically a subset of the original headers that have been reformatted or abridged to be more attractive. Each message in a Babyl mailbox also has an accompanying list of labels, or short strings that record extra information about the message, and a list of all user-defined labels found in the mailbox is kept in the Babyl options section.

Babyl instances have all of the methods of Mailbox in addition to the following:

get_labels()

Return a list of the names of all user-defined labels used in the mailbox.

Note: The actual messages are inspected to determine which labels exist in the mailbox rather than consulting the list of labels in the Babyl options section, but the Babyl section is updated whenever the mailbox is modified.

Some Mailbox methods implemented by Babyl deserve special remarks:

get_file (key)

In Babyl mailboxes, the headers of a message are not stored contiguously with the body of the message. To generate a file-like representation, the headers and body are copied together into a io.BytesIO instance, which has an API identical to that of a file. As a result, the file-like object is truly independent of the underlying mailbox but does not save memory compared to a string representation.

lock()
unlock()

Three locking mechanisms are used—dot locking and, if available, the flock() and lockf() system calls.

See Also:

Format of Version 5 Babyl Files  A specification of the Babyl format.

Reading Mail with Rmail  The Rmail manual, with some information on Babyl semantics.

MMDF

class mailbox.MMDF (path, factory=None, create=True)

A subclass of Mailbox for mailboxes in MMDF format. Parameter factory is a callable object that accepts
a file-like message representation (which behaves as if opened in binary mode) and returns a custom rep-

resentation. If factory is None, MMDFMessage is used as the default message representation. If create is

True, the mailbox is created if it does not exist.

MMDF is a single-file mailbox format invented for the Multichannel Memorandum Distribution Facility, a

mail transfer agent. Each message is in the same form as an mbox message but is bracketed before and after

by lines containing four Control-A (‘\001’) characters. As with the mbox format, the beginning of each

message is indicated by a line whose first five characters are “From ”, but additional occurrences of “From

” are not transformed to “>From ” when storing messages because the extra message separator lines prevent

mistaking such occurrences for the starts of subsequent messages.

Some Mailbox methods implemented by MMDF deserve special remarks:

get_file(key)

Using the file after calling flush() or close() on the MMDF instance may yield unpredictable

results or raise an exception.

lock()
unlock()

Three locking mechanisms are used—dot locking and, if available, the flock() and lockf() system calls.

See Also:

mmdf man page from tin A specification of MMDF format from the documentation of tin, a newsreader.

MMDF A Wikipedia article describing the Multichannel Memorandum Distribution Facility.

19.4.2 Message objects

class mailbox.Message (message=None)

A subclass of the email.message module’s Message. Subclasses of mailbox.Message add

mailbox-format-specific state and behavior.

If message is omitted, the new instance is created in a default, empty state. If message is an

email.message.Message instance, its contents are copied; furthermore, any format-specific infor-

mation is converted insofar as possible if message is a Message instance. If message is a string, a byte

string, or a file, it should contain an RFC 2822-compliant message, which is read and parsed. Files should

be open in binary mode, but text mode files are accepted for backward compatibility.

The format-specific state and behaviors offered by subclasses vary, but in general it is only the properties that

are not specific to a particular mailbox that are supported (although presumably the properties are specific
to a particular mailbox format). For example, file offsets for single-file mailbox formats and file names for
directory-based mailbox formats are not retained, because they are only applicable to the original mailbox.
But state such as whether a message has been read by the user or marked as important is retained, because
it applies to the message itself.

There is no requirement that Message instances be used to represent messages retrieved using Mailbox
instances. In some situations, the time and memory required to generate Message representations might
not be acceptable. For such situations, Mailbox instances also offer string and file-like representations,
and a custom message factory may be specified when a Mailbox instance is initialized.

MaildirMessage

class mailbox.MaildirMessage (message=None)

A message with Maildir-specific behaviors. Parameter message has the same meaning as with the Message
constructor.

Typically, a mail user agent application moves all of the messages in the new subdirectory to the cur
subdirectory after the first time the user opens and closes the mailbox, recording that the messages are old
whether or not they’ve actually been read. Each message in cur has an “info” section added to its file name
to store information about its state. (Some mail readers may also add an “info” section to messages in new.)
The “info” section may take one of two forms: it may contain “2,” followed by a list of standardized flags (e.g., “2,FR”) or it may contain “1,” followed by so-called experimental information. Standard flags for Maildir messages are as follows:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Draft</td>
<td>Under composition</td>
</tr>
<tr>
<td>F</td>
<td>Flagged</td>
<td>Marked as important</td>
</tr>
<tr>
<td>P</td>
<td>Passed</td>
<td>Forwarded, resent, or bounced</td>
</tr>
<tr>
<td>R</td>
<td>Replied</td>
<td>Replied to</td>
</tr>
<tr>
<td>S</td>
<td>Seen</td>
<td>Read</td>
</tr>
<tr>
<td>T</td>
<td>Trashed</td>
<td>Marked for subsequent deletion</td>
</tr>
</tbody>
</table>

MaildirMessage instances offer the following methods:

- **get_subdir()**
  - Return either “new” (if the message should be stored in the new subdirectory) or “cur” (if the message should be stored in the cur subdirectory).
  
  **Note:** A message is typically moved from new to cur after its mailbox has been accessed, whether or not the message has been read. A message msg has been read if "S" in msg.get_flags() is True.

- **set_subdir(subdir)**
  - Set the subdirectory the message should be stored in. Parameter subdir must be either “new” or “cur”.

- **get_flags()**
  - Return a string specifying the flags that are currently set. If the message complies with the standard Maildir format, the result is the concatenation in alphabetical order of zero or one occurrence of each of 'D', 'F', 'P', 'R', 'S', and 'T'. The empty string is returned if no flags are set or if “info” contains experimental semantics.

- **set_flags(flags)**
  - Set the flags specified by flags and unset all others.

- **add_flag(flag)**
  - Set the flag(s) specified by flag without changing other flags. To add more than one flag at a time, flag may be a string of more than one character. The current “info” is overwritten whether or not it contains experimental information rather than flags.

- **remove_flag(flag)**
  - Unset the flag(s) specified by flag without changing other flags. To remove more than one flag at a time, flag maybe a string of more than one character. If “info” contains experimental information rather than flags, the current “info” is not modified.

- **get_date()**
  - Return the delivery date of the message as a floating-point number representing seconds since the epoch.

- **set_date(date)**
  - Set the delivery date of the message to date, a floating-point number representing seconds since the epoch.

- **get_info()**
  - Return a string containing the “info” for a message. This is useful for accessing and modifying “info” that is experimental (i.e., not a list of flags).

- **set_info(info)**
  - Set “info” to info, which should be a string.

When a MaildirMessage instance is created based upon a mboxMessage or MMDFMessage instance, the Status and X-Status headers are omitted and the following conversions take place:
mboxMessage

class mailbox.mboxMessage (message=None)
A message with mbox-specific behaviors. Parameter message has the same meaning as with the Message constructor.

Messages in an mbox mailbox are stored together in a single file. The sender’s envelope address and the time of delivery are typically stored in a line beginning with “From ” that is used to indicate the start of a message, though there is considerable variation in the exact format of this data among mbox implementations. Flags that indicate the state of the message, such as whether it has been read or marked as important, are typically stored in Status and X-Status headers.

Conventional flags for mbox messages are as follows:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Read</td>
<td>Read</td>
</tr>
<tr>
<td>O</td>
<td>Old</td>
<td>Previously detected by MUA</td>
</tr>
<tr>
<td>D</td>
<td>Deleted</td>
<td>Marked for subsequent deletion</td>
</tr>
<tr>
<td>F</td>
<td>Flagged</td>
<td>Marked as important</td>
</tr>
<tr>
<td>A</td>
<td>Answered</td>
<td>Replied to</td>
</tr>
</tbody>
</table>

The “R” and “O” flags are stored in the Status header, and the “D”, “F”, and “A” flags are stored in the X-Status header. The flags and headers typically appear in the order mentioned.

mboxMessage instances offer the following methods:

get_from()
Return a string representing the “From ” line that marks the start of the message in an mbox mailbox. The leading “From ” and the trailing newline are excluded.

set_from(from_, time_=None)
Set the “From ” line to from_, which should be specified without a leading “From ” or trailing newline. For convenience, time_ may be specified and will be formatted appropriately and appended to from_. If time_ is specified, it should be a time.struct_time instance, a tuple suitable for passing to time.strftime(), or True (to use time.gmtime()).
get_flags()
    Return a string specifying the flags that are currently set. If the message complies with the conventional
    format, the result is the concatenation in the following order of zero or one occurrence of each of 'R',
    'O', 'D', 'F', and 'A'.

set_flags(flags)
    Set the flags specified by flags and unset all others. Parameter flags should be the concatenation in any
    order of zero or more occurrences of each of 'R', 'O', 'D', 'F', and 'A'.

add_flag(flag)
    Set the flag(s) specified by flag without changing other flags. To add more than one flag at a time, flag
    may be a string of more than one character.

remove_flag(flag)
    Unset the flag(s) specified by flag without changing other flags. To remove more than one flag at a
    time, flag maybe a string of more than one character.

When an mboxMessage instance is created based upon a MaildirMessage instance, a “From ” line is gen-
erated based upon the MaildirMessage instance’s delivery date, and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MaildirMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag</td>
<td>S flag</td>
</tr>
<tr>
<td>O flag</td>
<td>“cur” subdirectory</td>
</tr>
<tr>
<td>D flag</td>
<td>T flag</td>
</tr>
<tr>
<td>F flag</td>
<td>F flag</td>
</tr>
<tr>
<td>A flag</td>
<td>R flag</td>
</tr>
</tbody>
</table>

When an mboxMessage instance is created based upon an MHMessage instance, the following conversions
take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MHMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag and O flag</td>
<td>no “unseen” sequence</td>
</tr>
<tr>
<td>O flag</td>
<td>“unseen” sequence</td>
</tr>
<tr>
<td>F flag</td>
<td>“flagged” sequence</td>
</tr>
<tr>
<td>A flag</td>
<td>“replied” sequence</td>
</tr>
</tbody>
</table>

When an mboxMessage instance is created based upon a BabylMessage instance, the following conversions
take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>BabylMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag and O flag</td>
<td>no “unseen” label</td>
</tr>
<tr>
<td>O flag</td>
<td>“unseen” label</td>
</tr>
<tr>
<td>D flag</td>
<td>“deleted” label</td>
</tr>
<tr>
<td>A flag</td>
<td>“answered” label</td>
</tr>
</tbody>
</table>

When a Message instance is created based upon an MMDFMessage instance, the “From ” line is copied and all
flags directly correspond:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MMDFMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag</td>
<td>R flag</td>
</tr>
<tr>
<td>O flag</td>
<td>O flag</td>
</tr>
<tr>
<td>D flag</td>
<td>D flag</td>
</tr>
<tr>
<td>F flag</td>
<td>F flag</td>
</tr>
<tr>
<td>A flag</td>
<td>A flag</td>
</tr>
</tbody>
</table>

MHMessage

class mailbox.MHMessage(message=None)
    A message with MH-specific behaviors. Parameter message has the same meaning as with the Message
    constructor.

    MH messages do not support marks or flags in the traditional sense, but they do support sequences, which
    are logical groupings of arbitrary messages. Some mail reading programs (although not the standard mh
and `nmh`) use sequences in much the same way flags are used with other formats, as follows:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>unseen</td>
<td>Not read, but previously detected by MUA</td>
</tr>
<tr>
<td>replied</td>
<td>Replied to</td>
</tr>
<tr>
<td>flagged</td>
<td>Marked as important</td>
</tr>
</tbody>
</table>

`MHMessage` instances offer the following methods:

- `get_sequences()`
  - Return a list of the names of sequences that include this message.
- `set_sequences(sequences)`
  - Set the list of sequences that include this message.
- `add_sequence(sequence)`
  - Add `sequence` to the list of sequences that include this message.
- `remove_sequence(sequence)`
  - Remove `sequence` from the list of sequences that include this message.

When an `MHMessage` instance is created based upon a `MaildirMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MaildirMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“unseen” sequence</td>
<td>no S flag</td>
</tr>
<tr>
<td>“replied” sequence</td>
<td>R flag</td>
</tr>
<tr>
<td>“flagged” sequence</td>
<td>F flag</td>
</tr>
</tbody>
</table>

When an `MHMessage` instance is created based upon an `mboxMessage` or `MMDFMessage` instance, the `Status` and `X-Status` headers are omitted and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>mboxMessage or MMDFMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“unseen” sequence</td>
<td>no R flag</td>
</tr>
<tr>
<td>“replied” sequence</td>
<td>A flag</td>
</tr>
<tr>
<td>“flagged” sequence</td>
<td>F flag</td>
</tr>
</tbody>
</table>

When an `MHMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>BabylMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“unseen” sequence</td>
<td>“unseen” label</td>
</tr>
<tr>
<td>“replied” sequence</td>
<td>“answered” label</td>
</tr>
</tbody>
</table>

`BabylMessage`

class `mailbox.BabylMessage` *(message=None)*

A message with Babyl-specific behaviors. Parameter `message` has the same meaning as with the `Message` constructor.

Certain message labels, called attributes, are defined by convention to have special meanings. The attributes are as follows:

<table>
<thead>
<tr>
<th>Label</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>unseen</td>
<td>Not read, but previously detected by MUA</td>
</tr>
<tr>
<td>deleted</td>
<td>Marked for subsequent deletion</td>
</tr>
<tr>
<td>filed</td>
<td>Copied to another file or mailbox</td>
</tr>
<tr>
<td>answered</td>
<td>Replied to</td>
</tr>
<tr>
<td>forwarded</td>
<td>Forwarded</td>
</tr>
<tr>
<td>edited</td>
<td>Modified by the user</td>
</tr>
<tr>
<td>resent</td>
<td>Resent</td>
</tr>
</tbody>
</table>

By default, Rmail displays only visible headers. The `BabylMessage` class, though, uses the original headers because they are more complete. Visible headers may be accessed explicitly if desired.
BabylMessage instances offer the following methods:

get_labels()
Return a list of labels on the message.

set_labels(labels)
Set the list of labels on the message to labels.

add_label(label)
Add label to the list of labels on the message.

remove_label(label)
Remove label from the list of labels on the message.

get_visible()
Return an Message instance whose headers are the message’s visible headers and whose body is empty.

set_visible(visible)
Set the message’s visible headers to be the same as the headers in message. Parameter visible should be a Message instance, an email.message.Message instance, a string, or a file-like object (which should be open in text mode).

update_visible()
When a BabylMessage instance’s original headers are modified, the visible headers are not automatically modified to correspond. This method updates the visible headers as follows: each visible header with a corresponding original header is set to the value of the original header, each visible header without a corresponding original header is removed, and any of Date, From, Reply-To, To, CC, and Subject that are present in the original headers but not the visible headers are added to the visible headers.

When a BabylMessage instance is created based upon a MaildirMessage instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MaildirMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“unseen” label</td>
<td>no S flag</td>
</tr>
<tr>
<td>“deleted” label</td>
<td>T flag</td>
</tr>
<tr>
<td>“answered” label</td>
<td>R flag</td>
</tr>
<tr>
<td>“forwarded” label</td>
<td>P flag</td>
</tr>
</tbody>
</table>

When a BabylMessage instance is created based upon an mboxMessage or MMDFMessage instance, the Status and X>Status headers are omitted and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>mboxMessage or MMDFMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“unseen” label</td>
<td>no R flag</td>
</tr>
<tr>
<td>“deleted” label</td>
<td>D flag</td>
</tr>
<tr>
<td>“answered” label</td>
<td>A flag</td>
</tr>
</tbody>
</table>

When a BabylMessage instance is created based upon an MMMessage instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MMMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>“unseen” label</td>
<td>“unseen” sequence</td>
</tr>
<tr>
<td>“answered” label</td>
<td>“replied” sequence</td>
</tr>
</tbody>
</table>

**MMDFMessage**

class mailbox.MMDFMessage(message=None)
A message with MMDF-specific behaviors. Parameter message has the same meaning as with the Message constructor.

As with message in an mbox mailbox, MMDF messages are stored with the sender’s address and the delivery date in an initial line beginning with “From”. Likewise, flags that indicate the state of the message are typically stored in Status and X>Status headers.
Conventional flags for MMDF messages are identical to those of mbox message and are as follows:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Read</td>
<td>Read</td>
</tr>
<tr>
<td>O</td>
<td>Old</td>
<td>Previously detected by MUA</td>
</tr>
<tr>
<td>D</td>
<td>Deleted</td>
<td>Marked for subsequent deletion</td>
</tr>
<tr>
<td>F</td>
<td>Flagged</td>
<td>Marked as important</td>
</tr>
<tr>
<td>A</td>
<td>Answered</td>
<td>Replied to</td>
</tr>
</tbody>
</table>

The “R” and “O” flags are stored in the `Status` header, and the “D”, “F”, and “A” flags are stored in the `X-Status` header. The flags and headers typically appear in the order mentioned.

`MMDFMessage` instances offer the following methods, which are identical to those offered by `mboxMessage`:

- `get_from()`  
  Return a string representing the “From ” line that marks the start of the message in an mbox mailbox. The leading “From ” and the trailing newline are excluded.

- `set_from(from_, time_=None)`  
  Set the “From ” line to `from_`, which should be specified without a leading “From ” or trailing newline. For convenience, `time_` may be specified and will be formatted appropriately and appended to `from_`. If `time_` is specified, it should be a `time.struct_time` instance, a tuple suitable for passing to `time.strftime()`, or `True` (to use `time.gmtime()`).

- `get_flags()`  
  Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of ‘R’, ‘O’, ‘D’, ‘F’, and ‘A’.

- `set_flags(flags)`  
  Set the flags specified by `flags` and unset all others. Parameter `flags` should be the concatenation in any order of zero or more occurrences of each of ‘R’, ‘O’, ‘D’, ‘F’, and ‘A’.

- `add_flag(flag)`  
  Set the flag(s) specified by `flag` without changing other flags. To add more than one flag at a time, `flag` may be a string of more than one character.

- `remove_flag(flag)`  
  Unset the flag(s) specified by `flag` without changing other flags. To remove more than one flag at a time, `flag` maybe a string of more than one character.

When an `MMDFMessage` instance is created based upon a `MaildirMessage` instance, a “From ” line is generated based upon the `MaildirMessage` instance’s delivery date, and the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MaildirMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag</td>
<td>S flag</td>
</tr>
<tr>
<td>O flag</td>
<td>“cur” subdirectory</td>
</tr>
<tr>
<td>D flag</td>
<td>T flag</td>
</tr>
<tr>
<td>F flag</td>
<td>F flag</td>
</tr>
<tr>
<td>A flag</td>
<td>R flag</td>
</tr>
</tbody>
</table>

When an `MMDFMessage` instance is created based upon an `MHMessage` instance, the following conversions take place:

<table>
<thead>
<tr>
<th>Resulting state</th>
<th>MHMessage state</th>
</tr>
</thead>
<tbody>
<tr>
<td>R flag and O flag</td>
<td>no “unseen” sequence</td>
</tr>
<tr>
<td>O flag</td>
<td>“unseen” sequence</td>
</tr>
<tr>
<td>F flag</td>
<td>“flagged” sequence</td>
</tr>
<tr>
<td>A flag</td>
<td>“replied” sequence</td>
</tr>
</tbody>
</table>

When an `MMDFMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place:
## 19.4.3 Exceptions

The following exception classes are defined in the `mailbox` module:

**exception mailbox.Error**

The base class for all other module-specific exceptions.

**exception mailbox.NoSuchMailboxError**

Raised when a mailbox is expected but is not found, such as when instantiating a `Mailbox` subclass with a path that does not exist (and with the `create` parameter set to `False`), or when opening a folder that does not exist.

**exception mailbox.NotEmptyError**

Raised when a mailbox is not empty but is expected to be, such as when deleting a folder that contains messages.

**exception mailbox.ExternalClashError**

Raised when some mailbox-related condition beyond the control of the program causes it to be unable to proceed, such as when failing to acquire a lock that another program already holds a lock, or when a uniquely-generated file name already exists.

**exception mailbox.FormatError**

Raised when the data in a file cannot be parsed, such as when an `MH` instance attempts to read a corrupted `.mh_sequences` file.

## 19.4.4 Examples

A simple example of printing the subjects of all messages in a mailbox that seem interesting:

```python
import mailbox
for message in mailbox.mbox '~/mbox':
    subject = message['subject']
               # Could possibly be None.
    if subject and 'python' in subject.lower():
        print(subject)
```

To copy all mail from a Babyl mailbox to an MH mailbox, converting all of the format-specific information that can be converted:

```python
import mailbox
destination = mailbox.MH '~/Mail')
destination.lock()
for message in mailbox.Babyl '~/RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```
This example sorts mail from several mailing lists into different mailboxes, being careful to avoid mail corruption due to concurrent modification by other programs, mail loss due to interruption of the program, or premature termination due to malformed messages in the mailbox:

```python
import mailbox
import email.Errors

list_names = ('python-list', 'python-dev', 'python-bugs')

boxes = {name: mailbox.mbox('~/email/%s' % name) for name in list_names}
inbox = mailbox.Maildir('~/Maildir', factory=None)

for key in inbox.iterkeys():
    try:
        message = inbox[key]
    except email.Errors.MessageParseError:
        continue  # The message is malformed. Just leave it.

for name in list_names:
    list_id = message['list-id']
    if list_id and name in list_id:
        # Get mailbox to use
        box = boxes[name]

        # Write copy to disk before removing original.
        # If there’s a crash, you might duplicate a message, but
        # that’s better than losing a message completely.
        box.lock()
        box.add(message)
        box.flush()
        box.unlock()

        # Remove original message
        inbox.lock()
        inbox.discard(key)
        inbox.flush()
        inbox.unlock()
        break  # Found destination, so stop looking.

for box in boxes.itervalues():
    box.close()
```

### 19.5 mimetypes — Map filenames to MIME types

**Source code:** Lib/mimetypes.py

The `mimetypes` module converts between a filename or URL and the MIME type associated with the filename extension. Conversions are provided from filename to MIME type and from MIME type to filename extension; encodings are not supported for the latter conversion.

The module provides one class and a number of convenience functions. The functions are the normal interface to this module, but some applications may be interested in the class as well.

The functions described below provide the primary interface for this module. If the module has not been initialized, they will call `init()` if they rely on the information `init()` sets up.

```python
mimetypes.guess_type(url, strict=True)
```

Guess the type of a file based on its filename or URL, given by `url`. The return value is a tuple `(type, `
encoding) where type is None if the type can’t be guessed (missing or unknown suffix) or a string of the form ‘type/subtype’, usable for a MIME content-type header.

encoding is None for no encoding or the name of the program used to encode (e.g. compress or gzip). The encoding is suitable for use as a Content-Encoding header, not as a Content-Transfer-Encoding header. The mappings are table driven. Encoding suffixes are case sensitive; type suffixes are first tried case sensitively, then case insensitively.

The optional strict argument is a flag specifying whether the list of known MIME types is limited to only the official types registered with IANA. When strict is True (the default), only the IANA types are supported; when strict is False, some additional non-standard but commonly used MIME types are also recognized.

mimetypes.guess_all_extensions(type, strict=True)
Guess the extensions for a file based on its MIME type, given by type. The return value is a list of strings giving all possible filename extensions, including the leading dot (‘.’). The extensions are not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type type by guess_type().

The optional strict argument has the same meaning as with the guess_type() function.

mimetypes.guess_extension(type, strict=True)
Guess the extension for a file based on its MIME type, given by type. The return value is a string giving a filename extension, including the leading dot (‘.’). The extension is not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type type by guess_type(). If no extension can be guessed for type, None is returned.

The optional strict argument has the same meaning as with the guess_type() function.

Some additional functions and data items are available for controlling the behavior of the module.

mimetypes.init(files=None)
Initialize the internal data structures. If given, files must be a sequence of file names which should be used to augment the default type map. If omitted, the file names to use are taken from knownfiles; on Windows, the current registry settings are loaded. Each file named in files or knownfiles takes precedence over those named before it. Calling init() repeatedly is allowed.

Specifying an empty list for files will prevent the system defaults from being applied: only the well-known values will be present from a built-in list. Changed in version 3.2: Previously, Windows registry settings were ignored.

mimetypes.read_mime_types(filename)
Load the type map given in the file filename, if it exists. The type map is returned as a dictionary mapping filename extensions, including the leading dot (‘.’), to strings of the form ‘type/subtype’. If the file filename does not exist or cannot be read, None is returned.

mimetypes.add_type(type, ext, strict=True)
Add a mapping from the MIME type type to the extension ext. When the extension is already known, the new type will replace the old one. When the type is already known the extension will be added to the list of known extensions.

When strict is True (the default), the mapping will added to the official MIME types, otherwise to the non-standard ones.

mimetypes.inited
Flag indicating whether or not the global data structures have been initialized. This is set to True by init().

mimetypes.knownfiles
List of type map file names commonly installed. These files are typically named mime.types and are installed in different locations by different packages.

mimetypes.suffix_map
Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the .tgz extension is mapped to .tar.gz to allow the encoding and type to be recognized separately.
mimetypes.encodings_map
Dictionary mapping filename extensions to encoding types.

mimetypes.types_map
Dictionary mapping filename extensions to MIME types.

mimetypes.common_types
Dictionary mapping filename extensions to non-standard, but commonly found MIME types.

An example usage of the module:

```python
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

19.5.1 MimeTypes Objects

The `MimeTypes` class may be useful for applications which may want more than one MIME-type database; it provides an interface similar to the one of the `mimetypes` module.

```python
class mimetypes.MimeTypes (filenames=(), strict=True)
```

This class represents a MIME-types database. By default, it provides access to the same database as the rest of this module. The initial database is a copy of that provided by the module, and may be extended by loading additional `mime.types`-style files into the database using the `read()` or `readfp()` methods. The mapping dictionaries may also be cleared before loading additional data if the default data is not desired.

The optional `filenames` parameter can be used to cause additional files to be loaded “on top” of the default database.

```python
MimeTypes.suffix_map
Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the `.tgz` extension is mapped to `.tar.gz` to allow the encoding and type to be recognized separately. This is initially a copy of the global `suffix_map` defined in the module.

MimeTypes.encodings_map
Dictionary mapping filename extensions to encoding types. This is initially a copy of the global `encodings_map` defined in the module.

MimeTypes.types_map
Tuple containing two dictionaries, mapping filename extensions to MIME types: the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by `common_types` and `types_map`.

MimeTypes.types_map_inv
Tuple containing two dictionaries, mapping MIME types to a list of filename extensions: the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by `common_types` and `types_map`.

MimeTypes.guess_extension (type, strict=True)
Similar to the `guess_extension()` function, using the tables stored as part of the object.

MimeTypes.guess_type (url, strict=True)
Similar to the `guess_type()` function, using the tables stored as part of the object.

MimeTypes.guess_all_extensions (type, strict=True)
Similar to the `guess_all_extensions()` function, using the tables stored as part of the object.
MimeTypes.read (filename, strict=True)
Load MIME information from a file named filename. This uses readfp () to parse the file.
If strict is True, information will be added to list of standard types, else to the list of non-standard types.

MimeTypes.readfp (fp, strict=True)
Load MIME type information from an open file fp. The file must have the format of the standard mime.types files.
If strict is True, information will be added to the list of standard types, else to the list of non-standard types.

MimeTypes.read_windows_registry (strict=True)
Load MIME type information from the Windows registry. Availability: Windows.
If strict is True, information will be added to the list of standard types, else to the list of non-standard types. New in version 3.2.

19.6 base64 — RFC 3548: Base16, Base32, Base64 Data Encodings

This module provides data encoding and decoding as specified in RFC 3548. This standard defines the Base16, Base32, and Base64 algorithms for encoding and decoding arbitrary binary strings into ASCII-only byte strings that can be safely sent by email, used as parts of URLs, or included as part of an HTTP POST request. The encoding algorithm is not the same as the uuencode program.

There are two interfaces provided by this module. The modern interface supports encoding and decoding ASCII byte string objects using all three alphabets. Additionally, the decoding functions of the modern interface also accept Unicode strings containing only ASCII characters. The legacy interface provides for encoding and decoding to and from file-like objects as well as byte strings, but only using the Base64 standard alphabet. Changed in version 3.3: ASCII-only Unicode strings are now accepted by the decoding functions of the modern interface. The modern interface provides:

base64.b64encode (s, altchars=None)
Encode a byte string using Base64.

    s is the string to encode. Optional altchars must be a string of at least length 2 (additional characters are ignored) which specifies an alternative alphabet for the + and / characters. This allows an application to e.g. generate URL or filesystem safe Base64 strings. The default is None, for which the standard Base64 alphabet is used.

    The encoded byte string is returned.

base64.b64decode (s, altchars=None, validate=False)
Decode a Base64 encoded byte string.

    s is the byte string to decode. Optional altchars must be a string of at least length 2 (additional characters are ignored) which specifies the alternative alphabet used instead of the + and / characters.

    The decoded string is returned. A binascii.Error exception is raised if s is incorrectly padded.

    If validate is False (the default), non-base64-alphabet characters are discarded prior to the padding check.
    If validate is True, non-base64-alphabet characters in the input result in a binascii.Error.

base64.standard_b64encode (s)
Encode byte string s using the standard Base64 alphabet.

base64.standard_b64decode (s)
Decode byte string s using the standard Base64 alphabet.

base64.urlsafe_b64encode (s)
Encode byte string s using a URL-safe alphabet, which substitutes _ instead of + and / instead of / in the standard Base64 alphabet. The result can still contain _.
The Python Library Reference, Release 3.3.3

base64.urlsafe_b64decode(s)
    Decode byte string s using a URL-safe alphabet, which substitutes - instead of + and _ instead of / in the standard Base64 alphabet.

base64.b32encode(s)
    Encode a byte string using Base32. s is the string to encode. The encoded string is returned.

base64.b32decode(s, casefold=False, map01=None)
    Decode a Base32 encoded byte string.

    s is the byte string to decode. Optional casefold is a flag specifying whether a lowercase alphabet is acceptable as input. For security purposes, the default is False.

    RFC 3548 allows for optional mapping of the digit 0 (zero) to the letter O (oh), and for optional mapping of the digit 1 (one) to either the letter I (eye) or letter L (el). The optional argument map01 when not None, specifies which letter the digit 1 should be mapped to (when map01 is not None, the digit 0 is always mapped to the letter O). For security purposes the default is None, so that 0 and 1 are not allowed in the input.

    The decoded byte string is returned. A binascii.Error is raised if s were incorrectly padded or if there are non-alphabet characters present in the string.

base64.b16encode(s)
    Encode a byte string using Base16.

    s is the string to encode. The encoded byte string is returned.

base64.b16decode(s, casefold=False)
    Decode a Base16 encoded byte string.

    s is the string to decode. Optional casefold is a flag specifying whether a lowercase alphabet is acceptable as input. For security purposes, the default is False.

    The decoded byte string is returned. A TypeError is raised if s were incorrectly padded or if there are non-alphabet characters present in the string.

The legacy interface:

base64.decode(input, output)
    Decode the contents of the binary input file and write the resulting binary data to the output file. input and output must be file objects. input will be read until input.read() returns an empty bytes object.

base64.decodestring(s)
    Decodes the byte string s, which must contain one or more lines of base64 encoded data, and return a byte string containing the resulting binary data. decodestring is a deprecated alias. New in version 3.1.

base64.encode(input, output)
    Encode the contents of the binary input file and write the resulting base64 encoded data to the output file. input and output must be file objects. input will be read until input.read() returns an empty bytes object. encode() returns the encoded data plus a trailing newline character (b’\n’).

base64.encodebytes(s)
    Encode the byte string s, which can contain arbitrary binary data, and return a byte string containing one or more lines of base64-encoded data. encodebytes() returns a string containing one or more lines of base64-encoded data always including an extra trailing newline (b’\n’). encodestring is a deprecated alias.

An example usage of the module:

>>> import base64
>>> encoded = base64.b64encode(b’data to be encoded’)
>>> encoded
b’ZGF0YSB0byBiZSBlbmNvZGVk’
>>> data = base64.b64decode(encoded)
>>> data
b'data to be encoded'

See Also:
Module `binascii` Support module containing ASCII-to-binary and binary-to-ASCII conversions.

RFC 1521 - MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies

Section 5.2, “Base64 Content-Transfer-Encoding,” provides the definition of the base64 encoding.

### 19.7 `binhex` — Encode and decode binhex4 files

This module encodes and decodes files in binhex4 format, a format allowing representation of Macintosh files in ASCII. Only the data fork is handled.

The `binhex` module defines the following functions:

```python
binhex.binhex(input, output)
```

Convert a binary file with filename `input` to binhex file `output`. The `output` parameter can either be a filename or a file-like object (any object supporting a `write()` and `close()` method).

```python
binhex.hexbin(input, output)
```

Decode a binhex file `input`. `input` may be a filename or a file-like object supporting `read()` and `close()` methods. The resulting file is written to a file named `output`, unless the argument is `None` in which case the output filename is read from the binhex file.

The following exception is also defined:

```python
exception binhex.Error
```

Exception raised when something can’t be encoded using the binhex format (for example, a filename is too long to fit in the filename field), or when input is not properly encoded binhex data.

See Also:
Module `binascii` Support module containing ASCII-to-binary and binary-to-ASCII conversions.

### 19.7.1 Notes

There is an alternative, more powerful interface to the coder and decoder, see the source for details.

If you code or decode textfiles on non-Macintosh platforms they will still use the old Macintosh newline convention (carriage-return as end of line).

As of this writing, `hexbin()` appears to not work in all cases.

### 19.8 `binascii` — Convert between binary and ASCII

The `binascii` module contains a number of methods to convert between binary and various ASCII-encoded binary representations. Normally, you will not use these functions directly but use wrapper modules like `uu`, `base64`, or `binhex` instead. The `binascii` module contains low-level functions written in C for greater speed that are used by the higher-level modules.

Note: `a2b_*` functions accept Unicode strings containing only ASCII characters. Other functions only accept `bytes-like objects` (such as `bytes`, `bytearray` and other objects that support the buffer protocol). Changed in version 3.3: ASCII-only unicode strings are now accepted by the `a2b_*` functions.

The `binascii` module defines the following functions:
The Python Library Reference, Release 3.3.3

`binascii.a2b_uu` *(string)*

Convert a single line of uuencoded data back to binary and return the binary data. Lines normally contain 45 (binary) bytes, except for the last line. Line data may be followed by whitespace.

`binascii.b2a_uu` *(data)*

Convert binary data to a line of ASCII characters, the return value is the converted line, including a newline char. The length of `data` should be at most 45.

`binascii.a2b_base64` *(string)*

Convert a block of base64 data back to binary and return the binary data. More than one line may be passed at a time.

`binascii.b2a_base64` *(data)*

Convert binary data to a line of ASCII characters in base64 coding. The return value is the converted line, including a newline char. The length of `data` should be at most 57 to adhere to the base64 standard.

`binascii.a2b_qp` *(string, header=False)*

Convert a block of quoted-printable data back to binary and return the binary data. More than one line may be passed at a time. If the optional argument `header` is present and true, underscores will be decoded as spaces. Changed in version 3.2: Accept only bytestring or bytearray objects as input.

`binascii.b2a_qp` *(data, quotetabs=False, istext=True, header=False)*

Convert binary data to a line(s) of ASCII characters in quoted-printable encoding. The return value is the converted line(s). If the optional argument `quotetabs` is present and true, all tabs and spaces will be encoded. If the optional argument `istext` is present and true, newlines are not encoded but trailing whitespace will be encoded. If the optional argument `header` is present and true, spaces will be encoded as underscores per RFC1522. If the optional argument `header` is present and false, newline characters will be encoded as well; otherwise linefeed conversion might corrupt the binary data stream.

`binascii.a2b_hqx` *(string)*

Convert binhex4 formatted ASCII data to binary, without doing RLE-decompression. The string should contain a complete number of binary bytes, or (in case of the last portion of the binhex4 data) have the remaining bits zero.

`binascii.rledecode_hqx` *(data)*

Perform RLE-decompression on the data, as per the binhex4 standard. The algorithm uses 0x90 after a byte as a repeat indicator, followed by a count. A count of 0 specifies a byte value of 0x90. The routine returns the decompressed data, unless data input data ends in an orphaned repeat indicator, in which case the `Incomplete` exception is raised. Changed in version 3.2: Accept only bytestring or bytearray objects as input.

`binascii.rlecode_hqx` *(data)*

Perform binhex4 style RLE-compression on `data` and return the result.

`binascii.crc_hqx` *(data, crc)*

Compute the binhex4 crc value of `data`, starting with an initial `crc` and returning the result.

`binascii.crc32` *(data, crc)*

Compute CRC-32, the 32-bit checksum of data, starting with an initial crc. This is consistent with the ZIP file checksum. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm. Use as follows:

```python
print(binascii.crc32(b"hello world"))
# Or, in two pieces:
crc = binascii.crc32(b"hello")
crc = binascii.crc32(b" world", crc) & 0xffffffff
print('crc32 = {:#010x}'.format(crc))
```

Chapter 19. Internet Data Handling
Note: To generate the same numeric value across all Python versions and platforms use \texttt{crc32(data) \& 0xffffffff}. If you are only using the checksum in packed binary format this is not necessary as the return value is the correct 32bit binary representation regardless of sign.

\begin{verbatim}
binascii.b2a_hex(data)
\end{verbatim}

\begin{verbatim}
binascii.hexlify(data)
\end{verbatim}

Return the hexadecimal representation of the binary \texttt{data}. Every byte of \texttt{data} is converted into the corresponding 2-digit hex representation. The resulting string is therefore twice as long as the length of \texttt{data}.

\begin{verbatim}
binascii.a2b_hex(hexstr)
\end{verbatim}

\begin{verbatim}
binascii.unhexlify(hexstr)
\end{verbatim}

Return the binary data represented by the hexadecimal string \texttt{hexstr}. This function is the inverse of \texttt{b2a_hex()}. \texttt{hexstr} must contain an even number of hexadecimal digits (which can be upper or lower case), otherwise a \texttt{TypeError} is raised. Changed in version 3.2: Accept only bytestring or bytearray objects as input.

\begin{verbatim}
exception binascii.Error
\end{verbatim}

Exception raised on errors. These are usually programming errors.

\begin{verbatim}
exception binascii.Incomplete
\end{verbatim}

Exception raised on incomplete data. These are usually not programming errors, but may be handled by reading a little more data and trying again.

See Also:

Module \texttt{base64} Support for base64 encoding used in MIME email messages.

Module \texttt{binhex} Support for the binhex format used on the Macintosh.

Module \texttt{uu} Support for UU encoding used on Unix.

Module \texttt{quopri} Support for quoted-printable encoding used in MIME email messages.

\section{19.9 \texttt{quopri} — Encode and decode MIME quoted-printable data}

Source code: Lib/quopri.py

This module performs quoted-printable transport encoding and decoding, as defined in RFC 1521: “MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies”. The quoted-printable encoding is designed for data where there are relatively few nonprintable characters; the base64 encoding scheme available via the \texttt{base64} module is more compact if there are many such characters, as when sending a graphics file.

\begin{verbatim}
quopri.decode(input, output, header=False)
\end{verbatim}

Decode the contents of the \texttt{input} file and write the resulting decoded binary data to the \texttt{output} file. \texttt{input} and \texttt{output} must be \texttt{file} objects. \texttt{input} will be read until \texttt{input.readline()} returns an empty string. If the optional argument \texttt{header} is present and true, underscore will be decoded as space. This is used to decode “Q”-encoded headers as described in RFC 1522: “MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text”.

\begin{verbatim}
quopri.encode(input, output, quotetabs, header=False)
\end{verbatim}

Encode the contents of the \texttt{input} file and write the resulting quoted-printable data to the \texttt{output} file. \texttt{input} and \texttt{output} must be \texttt{file objects}. \texttt{input} will be read until \texttt{input.readline()} returns an empty string. \texttt{quotetabs} is a flag which controls whether to encode embedded spaces and tabs; when true it encodes such embedded whitespace, and when false it leaves them unencoded. Note that spaces and tabs appearing at the end of lines are always encoded, as per RFC 1521. \texttt{header} is a flag which controls if spaces are encoded as underscores as per RFC 1522.

\begin{verbatim}
quopri.decodestring(s, header=False)
\end{verbatim}

Like \texttt{decode()}, except that it accepts a source string and returns the corresponding decoded string.
**quopri.**

**`encodestring(s, quotetabs=False, header=False)`**

Like `encode()`, except that it accepts a source string and returns the corresponding encoded string. `quotetabs` and `header` are optional (defaulting to `False`), and are passed straight through to `encode()`.

**See Also:**

*Module base64*  
Encode and decode MIME base64 data

---

**19.10  uu — Encode and decode uuencode files**

**Source code:** Lib/uu.py

This module encodes and decodes files in uuencode format, allowing arbitrary binary data to be transferred over ASCII-only connections. Wherever a file argument is expected, the methods accept a file-like object. For backwards compatibility, a string containing a pathname is also accepted, and the corresponding file will be opened for reading and writing; the pathname `'-'` is understood to mean the standard input or output. However, this interface is deprecated; it’s better for the caller to open the file itself, and be sure that, when required, the mode is `'rb'` or `'wb'` on Windows.

This code was contributed by Lance Ellinghouse, and modified by Jack Jansen.

The **uu** module defines the following functions:

**uu.**

**`encode(in_file, out_file, name=None, mode=None)`**

Uuencode file `in_file` into file `out_file`. The uuencoded file will have the header specifying `name` and `mode` as the defaults for the results of decoding the file. The default defaults are taken from `in_file`, or `'- '` and `0o666` respectively.

**uu.**

**`decode(in_file, out_file=None, mode=None, quiet=False)`**

This call decodes uuencoded file `in_file` placing the result on file `out_file`. If `out_file` is a pathname, `mode` is used to set the permission bits if the file must be created. Defaults for `out_file` and `mode` are taken from the uuencode header. However, if the file specified in the header already exists, a **uu.Error** is raised.

`decode()` may print a warning to standard error if the input was produced by an incorrect uuencoder and Python could recover from that error. Setting `quiet` to a true value silences this warning.

**exception uu.**

**`Error`**  
Subclass of **`Exception`**, this can be raised by `uu.decode()` under various situations, such as described above, but also including a badly formatted header, or truncated input file.

**See Also:**

*Module binascii*  
Support module containing ASCII-to-binary and binary-to-ASCII conversions.
CHAPTER
TWENTY

STRUCTURED MARKUP
PROCESSING TOOLS

Python supports a variety of modules to work with various forms of structured data markup. This includes modules to work with the Standard Generalized Markup Language (SGML) and the Hypertext Markup Language (HTML), and several interfaces for working with the Extensible Markup Language (XML).

20.1 html — HyperText Markup Language support

Source code: Lib/html/__init__.py

This module defines utilities to manipulate HTML.

```
html.escape(s, quote=True)
```

Convert the characters &, < and > in string s to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. If the optional flag quote is true, the characters (") and (') are also translated; this helps for inclusion in an HTML attribute value delimited by quotes, as in <a href="...">. New in version 3.2.

Submodules in the html package are:

- html.parser – HTML/XHTML parser with lenient parsing mode
- html.entities – HTML entity definitions

20.2 html.parser — Simple HTML and XHTML parser

Source code: Lib/html/parser.py

This module defines a class HTMLParser which serves as the basis for parsing text files formatted in HTML (HyperText Mark-up Language) and XHTML.

```
class html.parser.HTMLParser (strict=False)
```

Create a parser instance. If strict is False (the default), the parser will accept and parse invalid markup. If strict is True the parser will raise an HTMLParseError exception instead \(^1\) when it’s not able to parse the markup. The use of strict=True is discouraged and the strict argument is deprecated.

\(^1\) For backward compatibility reasons strict mode does not raise exceptions for all non-compliant HTML. That is, some invalid HTML is tolerated even in strict mode.
An `HTMLParser` instance is fed HTML data and calls handler methods when start tags, end tags, text, comments, and other markup elements are encountered. The user should subclass `HTMLParser` and override its methods to implement the desired behavior.

This parser does not check that end tags match start tags or call the end-tag handler for elements which are closed implicitly by closing an outer element. Changed in version 3.2: `strict` keyword added. Deprecated since version 3.3, will be removed in version 3.5: The `strict` argument and the strict mode have been deprecated. The parser is now able to accept and parse invalid markup too.

An exception is defined as well:

```python
exception html.parser.HTMLParseError

Exception raised by the `HTMLParser` class when it encounters an error while parsing and `strict` is True. This exception provides three attributes: `msg` is a brief message explaining the error, `lineno` is the number of the line on which the broken construct was detected, and `offset` is the number of characters into the line at which the construct starts. Deprecated since version 3.3, will be removed in version 3.5: This exception has been deprecated because it’s never raised by the parser (when the default non-strict mode is used).
```

### 20.2.1 Example HTML Parser Application

As a basic example, below is a simple HTML parser that uses the `HTMLParser` class to print out start tags, end tags, and data as they are encountered:

```python
from html.parser import HTMLParser

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Encountered a start tag: ", tag)
    def handle_endtag(self, tag):
        print("Encountered an end tag: ", tag)
    def handle_data(self, data):
        print("Encountered some data : ", data)

parser = MyHTMLParser(strict=False)
parsed = parser.feed('<html><head><title>Test</title></head>'
                     '<body><h1>Parse me!</h1></body></html>')
```

The output will then be:

- Encountered a start tag: html
- Encountered a start tag: head
- Encountered a start tag: title
- Encountered some data : Test
- Encountered an end tag : title
- Encountered an end tag : head
- Encountered a start tag: body
- Encountered a start tag: h1
- Encountered some data : Parse me!
- Encountered an end tag : h1
- Encountered an end tag : body
- Encountered an end tag : html

### 20.2.2 HTMLParser Methods

`HTMLParser` instances have the following methods:

- `HTMLParser()`
  - `feed(data)`
    - Feed some text to the parser. It is processed insofar as it consists of complete elements; incomplete data is buffered until more data is fed or `close()` is called. `data` must be `str`.

---

Chapter 20. Structured Markup Processing Tools
HTMLParser.close()

Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call the HTMLParser base class method close().

HTMLParser.reset()

Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.

HTMLParser.getpos()

Return current line number and offset.

HTMLParser.get_starttag_text()

Return the text of the most recently opened start tag. This should not normally be needed for structured processing, but may be useful in dealing with HTML "as deployed" or for re-generating input with minimal changes (whitespace between attributes can be preserved, etc.).

The following methods are called when data or markup elements are encountered and they are meant to be over-ridden in a subclass. The base class implementations do nothing (except for handle_startendtag()):

HTMLParser.handle_starttag(tag, attrs)

This method is called to handle the start of a tag (e.g. <div id="main">).

The tag argument is the name of the tag converted to lower case. The attrs argument is a list of (name, value) pairs containing the attributes found inside the tag’s <> brackets. The name will be translated to lower case, and quotes in the value have been removed, and character and entity references have been replaced.

For instance, for the tag <A HREF="http://www.cwi.nl/">, this method would be called as handle_starttag('a', [('href', 'http://www.cwi.nl/')]).

All entity references from html.entities are replaced in the attribute values.

HTMLParser.handle_endtag(tag)

This method is called to handle the end tag of an element (e.g. </div>).

The tag argument is the name of the tag converted to lower case.

HTMLParser.handle_startendtag(tag, attrs)

Similar to handle_starttag(), but called when the parser encounters an XHTML-style empty tag (<img ... />). This method may be overridden by subclasses which require this particular lexical information; the default implementation simply calls handle_starttag() and handle_endtag().

HTMLParser.handle_data(data)

This method is called to process arbitrary data (e.g. text nodes and the content of <script>...</script> and <style>...</style>).

HTMLParser.handle_entityref(name)

This method is called to process a named character reference of the form &name; (e.g. &gt;), where name is a general entity reference (e.g. ‘qt’).

HTMLParser.handle_charref(name)

This method is called to process decimal and hexadecimal numeric character references of the form &#NNN; and &#xNNN;. For example, the decimal equivalent for &gt; is &#62;, whereas the hexadecimal is &#x3E; in this case the method will receive ‘62’ or ‘x3E’.

HTMLParser.handle_comment(data)

This method is called when a comment is encountered (e.g. <!--comment-->).

For example, the comment <!-- comment --> will cause this method to be called with the argument ’comment’.

The content of Internet Explorer conditional comments (condcoms) will also be sent to this method, so, for <!--[if IE 9]>IE9-specific content<![endif]-->, this method will receive ’[if IE 9]>IE-specific content<![endif]’.

HTMLParser.handle_decl(decl)

This method is called to handle an HTML doctype declaration (e.g. <!DOCTYPE html>).

20.2. html.parser — Simple HTML and XHTML parser 795
The `decl` parameter will be the entire contents of the declaration inside the `<!...>` markup (e.g. `DOCTYPE html`).

**HTMLParser.handle_pi(data)**

Method called when a processing instruction is encountered. The `data` parameter will contain the entire processing instruction. For example, for the processing instruction `<?proc color='red'>`, this method would be called as `handle_pi("proc color='red' ")`. It is intended to be overridden by a derived class; the base class implementation does nothing.

**Note:** The `HTMLParser` class uses the SGML syntactic rules for processing instructions. An XHTML processing instruction using the trailing `?' will cause the `'?' to be included in `data`.

**HTMLParser.unknown_decl(data)**

This method is called when an unrecognized declaration is read by the parser.

The `data` parameter will be the entire contents of the declaration inside the `<![...]>` markup. It is sometimes useful to be overridden by a derived class. The base class implementation raises an `HTMLParseError` when `strict` is True.

### 20.2.3 Examples

The following class implements a parser that will be used to illustrate more examples:

```python
from html.parser import HTMLParser
from html.entities import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Start tag:", tag)
        for attr in attrs:
            print(" attr:", attr)
    def handle_endtag(self, tag):
        print("End tag:", tag)
    def handle_data(self, data):
        print("Data:", data)
    def handle_comment(self, data):
        print("Comment:", data)
    def handle_entityref(self, name):
        c = chr(name2codepoint[name])
        print("Named ent:", c)
    def handle_charref(self, name):
        if name.startswith('x'):
            c = chr(int(name[1:], 16))
        else:
            c = chr(int(name))
        print("Num ent:", c)
    def handle_decl(self, data):
        print("Decl:", data)

parser = MyHTMLParser(strict=False)
```

Parsing a doctype:

```python
>>> parser.feed("<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" 
... "http://www.w3.org/TR/html4/strict.dtd">
Decl : DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd"
```

Parsing an element with a few attributes and a title:

```python
```
>>> parser.feed('<img src="python-logo.png" alt="The Python logo">')
Start tag: img
  attr: ('src', 'python-logo.png')
  attr: ('alt', 'The Python logo')
>>> parser.feed('<h1>Python</h1>
Start tag: h1
Data : Python
End tag : h1
The content of script and style elements is returned as is, without further parsing:
>>> parser.feed('<style type="text/css">#python { color: green }</style>')
Start tag: style
  attr: ('type', 'text/css')
Data : #python { color: green }
End tag : style
>>> parser.feed('<script type="text/javascript">alert("hello!");</script>')
Start tag: script
  attr: ('type', 'text/javascript')
Data : alert("hello!");
End tag : script
Parsing comments:
>>> parser.feed('<!-- a comment -->
...<!--[if IE 9]>IE-specific content<![endif]-->
Comment : a comment
Comment : [if IE 9]>IE-specific content<![endif]
Parsing named and numeric character references and converting them to the correct char (note: these 3 references are all equivalent to ‘>’):
>>> parser.feed('&gt; '&#62;' &#3E;')
Named ent: >
Num ent : >
Num ent : >
Feeding incomplete chunks to feed() works, but handle_data() might be called more than once:
>>> for chunk in ['<span>buffer', 'ered', 'text</span>', 'pan']:
...    parser.feed(chunk)
...Start tag: span
Data : buffer
Data : ered
Data : text
End tag : span
Parsing invalid HTML (e.g. unquoted attributes) also works:
>>> parser.feed('<p><a class=link href=#main>tag soup</p ></a>')
Start tag: p
Start tag: a
  attr: ('class', 'link')
  attr: ('href', '#main')
Data : tag soup
End tag : p
End tag : a
20.3 html.entities — Definitions of HTML general entities

This module defines four dictionaries, html5, name2codepoint, codepoint2name, and entitydefs.

html.entities.html5
A dictionary that maps HTML5 named character references to the equivalent Unicode character(s), e.g. html5[‘gt;’] == ‘>’. Note that the trailing semicolon is included in the name (e.g. ‘gt;’), however some of the names are accepted by the standard even without the semicolon: in this case the name is present with and without the ‘;’. New in version 3.3.

html.entities.entitydefs
A dictionary mapping XHTML 1.0 entity definitions to their replacement text in ISO Latin-1.

html.entities.name2codepoint
A dictionary that maps HTML entity names to the Unicode codepoints.

html.entities.codepoint2name
A dictionary that maps Unicode codepoints to HTML entity names.

20.4 XML Processing Modules

Python’s interfaces for processing XML are grouped in the xml package.

Warning: The XML modules are not secure against erroneous or maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

It is important to note that modules in the xml package require that there be at least one SAX-compliant XML parser available. The Expat parser is included with Python, so the xml.parsers.expat module will always be available.

The documentation for the xml.dom and xml.sax packages are the definition of the Python bindings for the DOM and SAX interfaces.

The XML handling submodules are:

• xml.etree.ElementTree: the ElementTree API, a simple and lightweight
• xml.dom: the DOM API definition
• xml.dom.minidom: a lightweight DOM implementation
• xml.dom.pulldom: support for building partial DOM trees
• xml.sax: SAX2 base classes and convenience functions
• xml.parsers.expat: the Expat parser binding

20.5 XML vulnerabilities

The XML processing modules are not secure against maliciously constructed data. An attacker can abuse vulnerabilities for e.g. denial of service attacks, to access local files, to generate network connections to other machines, or to or circumvent firewalls. The attacks on XML abuse unfamiliar features like inline DTD (document type definition) with entities.

The following table gives an overview of the known attacks and if the various modules are vulnerable to them.

See http://www.w3.org/TR/html5/named-character-references.html
<table>
<thead>
<tr>
<th>kind</th>
<th>sax</th>
<th>etree</th>
<th>minidom</th>
<th>pulldom</th>
<th>xmlrpc</th>
</tr>
</thead>
<tbody>
<tr>
<td>billion laughs</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>quadratic blowup</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>external entity expansion</td>
<td>Yes</td>
<td>No (1)</td>
<td>Yes</td>
<td>No (2)</td>
<td>Yes</td>
</tr>
<tr>
<td>DTD retrieval</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>decompression bomb</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

1. `xml.etree.ElementTree` doesn’t expand external entities and raises a `ParserError` when an entity occurs.

2. `xml.dom.minidom` doesn’t expand external entities and simply returns the unexpanded entity verbatim.

3. `xmlrpc` doesn’t expand external entities and omits them.

**billion laughs / exponential entity expansion** The Billion Laughs attack – also known as exponential entity expansion – uses multiple levels of nested entities. Each entity refers to another entity several times, the final entity definition contains a small string. Eventually the small string is expanded to several gigabytes. The exponential expansion consumes lots of CPU time, too.

**quadratic blowup entity expansion** A quadratic blowup attack is similar to a Billion Laughs attack; it abuses entity expansion, too. Instead of nested entities it repeats one large entity with a couple of thousand chars over and over again. The attack isn’t as efficient as the exponential case but it avoids triggering countermeasures of parsers against heavily nested entities.

**external entity expansion** Entity declarations can contain more than just text for replacement. They can also point to external resources by public identifiers or system identifiers. System identifiers are standard URIs or can refer to local files. The XML parser retrieves the resource with e.g. HTTP or FTP requests and embeds the content into the XML document.

**DTD retrieval** Some XML libraries like Python’s mod:`xml.dom.pulldom` retrieve document type definitions from remote or local locations. The feature has similar implications as the external entity expansion issue.

**decompression bomb** The issue of decompression bombs (aka ZIP bomb) apply to all XML libraries that can parse compressed XML stream like gzipped HTTP streams or LZMA-ed files. For an attacker it can reduce the amount of transmitted data by three magnitudes or more.

The documentation of `defusedxml` on PyPI has further information about all known attack vectors with examples and references.

### 20.5.1 defused packages

`defusedxml` is a pure Python package with modified subclasses of all stdlib XML parsers that prevent any potentially malicious operation. The courses of action are recommended for any server code that parses untrusted XML data. The package also ships with example exploits and an extended documentation on more XML exploits like xpath injection.

`defusedexpat` provides a modified libexpat and patched replacement `pyexpat` extension module with countermeasures against entity expansion DoS attacks. Defusedexpat still allows a sane and configurable amount of entity expansions. The modifications will be merged into future releases of Python.

The workarounds and modifications are not included in patch releases as they break backward compatibility. After all inline DTD and entity expansion are well-defined XML features.

### 20.6 xml.etree.ElementTree — The ElementTree XML API

The `xml.etree.ElementTree` module implements a simple and efficient API for parsing and creating XML data. Changed in version 3.3: This module will use a fast implementation whenever available. The `xml.etree.cElementTree` module is deprecated.

**Warning:** The `xml.etree.ElementTree` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see `XML vulnerabilities`. 

---

20.6. *xml.etree.ElementTree* — The ElementTree XML API 799
20.6.1 Tutorial

This is a short tutorial for using xml.etree.ElementTree (ET in short). The goal is to demonstrate some of the building blocks and basic concepts of the module.

XML tree and elements

XML is an inherently hierarchical data format, and the most natural way to represent it is with a tree. ET has two classes for this purpose - ElementTree represents the whole XML document as a tree, and Element represents a single node in this tree. Interactions with the whole document (reading and writing to/from files) are usually done on the ElementTree level. Interactions with a single XML element and its sub-elements are done on the Element level.

Parsing XML

We’ll be using the following XML document as the sample data for this section:

```xml
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank>68</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

We can import this data by reading from a file:

```python
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

Or directly from a string:

```python
root = ET.fromstring(country_data_as_string)
```

`fromstring()` parses XML from a string directly into an Element, which is the root element of the parsed tree. Other parsing functions may create an ElementTree. Check the documentation to be sure.

As an Element, root has a tag and a dictionary of attributes:

```python
>>> root.tag
'data'
>>> root.attrib
{}
```
It also has children nodes over which we can iterate:

```python
>>> for child in root:
...     print(child.tag, child.attrib)
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

Children are nested, and we can access specific child nodes by index:

```python
>>> root[0][1].text
'2008'
```

### Finding interesting elements

`Element` has some useful methods that help iterate recursively over all the sub-tree below it (its children, their children, and so on). For example, `Element.iter()`:

```python
>>> for neighbor in root.iter('neighbor'):
...     print(neighbor.attrib)
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

`Element.findall()` finds only elements with a tag which are direct children of the current element. `Element.find()` finds the first child with a particular tag, and `Element.text` accesses the element’s text content. `Element.get()` accesses the element’s attributes:

```python
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print(name, rank)
...
Liechtenstein 1
Singapore 4
Panama 68
```

More sophisticated specification of which elements to look for is possible by using *XPath*.

### Modifying an XML File

`ElementTree` provides a simple way to build XML documents and write them to files. The `ElementTree.write()` method serves this purpose.

Once created, an `Element` object may be manipulated by directly changing its fields (such as `Element.text`), adding and modifying attributes (`Element.set()` method), as well as adding new children (for example with `Element.append()`).

Let’s say we want to add one to each country’s rank, and add an `updated` attribute to the rank element:

```python
>>> for rank in root.iter('rank'):
...     new_rank = int(rank.text) + 1
...     rank.text = str(new_rank)
...     rank.set('updated', 'yes')
...     tree.write('output.xml')
```

Our XML now looks like this:
We can remove elements using `Element.remove()`. Let’s say we want to remove all countries with a rank higher than 50:

```python
>>> for country in root.findall('country'):
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
... >>>
```

Our XML now looks like this:

```xml
<?xml version="1.0"?>
<data>
  <country name="Liechtenstein">
    <rank updated="yes">2</rank>
    <year>2008</year>
    <gdppc>141100</gdppc>
    <neighbor name="Austria" direction="E"/>
    <neighbor name="Switzerland" direction="W"/>
  </country>
  <country name="Singapore">
    <rank updated="yes">5</rank>
    <year>2011</year>
    <gdppc>59900</gdppc>
    <neighbor name="Malaysia" direction="N"/>
  </country>
  <country name="Panama">
    <rank updated="yes">69</rank>
    <year>2011</year>
    <gdppc>13600</gdppc>
    <neighbor name="Costa Rica" direction="W"/>
    <neighbor name="Colombia" direction="E"/>
  </country>
</data>
```

Building XML documents

The `SubElement()` function also provides a convenient way to create new sub-elements for a given element:

```python
>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
>>> c = ET.SubElement(a, 'c')
```
```python
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /></c><d /></a>
```

### Additional resources

See [http://effbot.org/zone/element-index.htm](http://effbot.org/zone/element-index.htm) for tutorials and links to other docs.

#### 20.6.2 XPath support

This module provides limited support for XPath expressions for locating elements in a tree. The goal is to support a small subset of the abbreviated syntax; a full XPath engine is outside the scope of the module.

**Example**

Here’s an example that demonstrates some of the XPath capabilities of the module. We’ll be using the countrydata XML document from the Parsing XML section:

```python
import xml.etree.ElementTree as ET
root = ET.fromstring(countrydata)

# Top-level elements
root.findall(".")

# All ‘neighbor’ grand-children of ‘country’ children of the top-level elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall("./year/.[@name='Singapore']")

# ‘year’ nodes that are children of nodes with name='Singapore'
root.findall("./*[@name='Singapore']/year")

# All ‘neighbor’ nodes that are the second child of their parent
root.findall("./neighbor[2]")
```
Supported XPath syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tag</code></td>
<td>Selects all child elements with the given tag. For example, <code>spam</code> selects all child elements named <code>spam</code>, <code>spam/egg</code> selects all grandchildren named <code>egg</code> in all children named <code>spam</code>.</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Selects all child elements. For example, <code>*/egg</code> selects all grandchildren named <code>egg</code>.</td>
</tr>
<tr>
<td><code>.</code></td>
<td>Selects the current node. This is mostly useful at the beginning of the path, to indicate that it’s a relative path.</td>
</tr>
<tr>
<td><code>//</code></td>
<td>Selects all subelements, on all levels beneath the current element. For example, <code>../egg</code> selects all egg elements in the entire tree.</td>
</tr>
<tr>
<td><code>[@attrib]</code></td>
<td>Selects all elements that have the given attribute.</td>
</tr>
<tr>
<td><code>[@attrib='value']</code></td>
<td>Selects all elements for which the given attribute has the given value. The value cannot contain quotes.</td>
</tr>
<tr>
<td><code>[tag]</code></td>
<td>Selects all elements that have a child named <code>tag</code>. Only immediate children are supported.</td>
</tr>
<tr>
<td><code>[position]</code></td>
<td>Selects all elements that are located at the given position. The position can be either an integer (1 is the first position), the expression <code>last()</code> (for the last position), or a position relative to the last position (e.g. <code>last()-1</code>).</td>
</tr>
</tbody>
</table>

Predicates (expressions within square brackets) must be preceded by a tag name, an asterisk, or another predicate. `position` predicates must be preceded by a tag name.

20.6.3 Reference

Functions

`xml.etree.ElementTree.Comment (text=None)`
Comment element factory. This factory function creates a special element that will be serialized as an XML comment by the standard serializer. The comment string can be either a bytestring or a Unicode string. `text` is a string containing the comment string. Returns an element instance representing a comment.

`xml.etree.ElementTree.dump (elem)`
Writes an element tree or element structure to sys.stdout. This function should be used for debugging only.

The exact output format is implementation dependent. In this version, it’s written as an ordinary XML file. `elem` is an element tree or an individual element.

`xml.etree.ElementTree.fromstring (text)`
Parses an XML section from a string constant. Same as `XML()`. `text` is a string containing XML data. Returns an `Element` instance.

`xml.etree.ElementTree.fromstringlist (sequence, parser=None)`
Parses an XML document from a sequence of string fragments. `sequence` is a list or other sequence containing XML data fragments. `parser` is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `Element` instance. New in version 3.2.

`xml.etree.ElementTree.iselement (element)`
Checks if an object appears to be a valid element object. `element` is an element instance. Returns a true value if this is an element object.

`xml.etree.ElementTree.iterparse (source, events=None, parser=None)`
Parses an XML section into an element tree incrementally, and reports what’s going on to the user. `source` is a filename or `file object` containing XML data. `events` is a list of events to report back. The supported events are the strings "start", "end", "start-ns" and "end-ns" (the “ns” events are used to get detailed namespace information). If `events` is omitted, only "end" events are reported. `parser` is an
optional parser instance. If not given, the standard `XMLParser` parser is used. `parser` can only use the default `TreeBuilder` as a target. Returns an `iterator` providing `(event, elem)` pairs.

**Note:** `iterparse()` only guarantees that it has seen the “>” character of a starting tag when it emits a “start” event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present.

If you need a fully populated element, look for “end” events instead.

```python
xml.etree.ElementTree.parse(source, parser=None)
```

Parses an XML section into an element tree. `source` is a filename or file object containing XML data. `parser` is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `Element` instance.

```python
xml.etree.ElementTree.ProcessingInstruction(target, text=None)
```

PI element factory. This factory function creates a special element that will be serialized as an XML processing instruction. `target` is a string containing the PI target. `text` is a string containing the PI contents, if given. Returns an element instance, representing a processing instruction.

```python
xml.etree.ElementTree.register_namespace(prefix, uri)
```

Registers a namespace prefix. The registry is global, and any existing mapping for either the given prefix or the namespace URI will be removed. `prefix` is a namespace prefix. `uri` is a namespace URI. Tags and attributes in this namespace will be serialized with the given prefix, if at all possible. New in version 3.2.

```python
xml.etree.ElementTree.SubElement(parent, tag, attrib={}, **extra)
```

Subelement factory. This function creates an element instance, and appends it to an existing element.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. `parent` is the parent element. `tag` is the subelement name. `attrib` is an optional dictionary, containing element attributes. `extra` contains additional attributes, given as keyword arguments. Returns an element instance.

```python
xml.etree.ElementTree.tostring(element, encoding="us-ascii", method="xml")
```

Generates a string representation of an XML element, including all subelements. `element` is an `Element` instance. `encoding` is the output encoding (default is US-ASCII). Use `encoding="unicode"` to generate a Unicode string (otherwise, a bytestring is generated). `method` is either "xml", "html" or "text" (default is "xml"). Returns an (optionally) encoded string containing the XML data.

```python
xml.etree.ElementTree.tostringlist(element, encoding="us-ascii", method="xml")
```

Generates a string representation of an XML element, including all subelements. `element` is an `Element` instance. `encoding` is the output encoding (default is US-ASCII). Use `encoding="unicode"` to generate a Unicode string (otherwise, a bytestring is generated). `method` is either "xml", "html" or "text" (default is "xml"). Returns a list of (optionally) encoded strings containing the XML data. It does not guarantee any specific sequence, except that `".join(tostringlist(element)) == tostring(element)` (New in version 3.2).

```python
xml.etree.ElementTree.XML(text, parser=None)
```

Parses an XML section from a string constant. This function can be used to embed “XML literals” in Python code. `text` is a string containing XML data. `parser` is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `Element` instance.

```python
xml.etree.ElementTree.XMLID(text, parser=None)
```

Parses an XML section from a string constant, and also returns a dictionary which maps from element ids to elements. `text` is a string containing XML data. `parser` is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns a tuple containing an `Element` instance and a dictionary.

**Element Objects**

```python
class xml.etree.ElementTree.Element(tag, attrib={}, **extra)
```

Element class. This class defines the Element interface, and provides a reference implementation of this

---

3 The encoding string included in XML output should conform to the appropriate standards. For example, “UTF-8” is valid, but “UTF8” is not. See [http://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl](http://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl) and [http://www.iana.org/assignments/character-sets](http://www.iana.org/assignments/character-sets).
interface.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. `tag` is the element name. `attrib` is an optional dictionary, containing element attributes. `extra` contains additional attributes, given as keyword arguments.

**tag**

A string identifying what kind of data this element represents (the element type, in other words).

**text**

The `text` attribute can be used to hold additional data associated with the element. As the name implies this attribute is usually a string but may be any application-specific object. If the element is created from an XML file the attribute will contain any text found between the element tags.

**tail**

The `tail` attribute can be used to hold additional data associated with the element. This attribute is usually a string but may be any application-specific object. If the element is created from an XML file the attribute will contain any text found after the element’s end tag and before the next tag.

**attrib**

A dictionary containing the element’s attributes. Note that while the `attrib` value is always a real mutable Python dictionary, an ElementTree implementation may choose to use another internal representation, and create the dictionary only if someone asks for it. To take advantage of such implementations, use the dictionary methods below whenever possible.

The following dictionary-like methods work on the element attributes.

**clear()**

Resets an element. This function removes all subelements, clears all attributes, and sets the text and tail attributes to `None`.

**get(key, default=None)**

Gets the element attribute named `key`.

Returns the attribute value, or `default` if the attribute was not found.

**items()**

Returns the element attributes as a sequence of (name, value) pairs. The attributes are returned in an arbitrary order.

**keys()**

Returns the elements attribute names as a list. The names are returned in an arbitrary order.

**set(key, value)**

Set the attribute `key` on the element to `value`.

The following methods work on the element’s children (subelements).

**append(subelement)**

Adds the element `subelement` to the end of this element’s internal list of subelements. Raises `TypeError` if `subelement` is not an `Element`.

**extend(subelements)**

Appends `subelements` from a sequence object with zero or more elements. Raises `TypeError` if a subelement is not an `Element`. New in version 3.2.

**find(match, namespaces=None)**

Finds the first subelement matching `match`. `match` may be a tag name or a `path`. Returns an element instance or `None`. `namespaces` is an optional mapping from namespace prefix to full name.

**findall(match, namespaces=None)**

Finds all matching subelements, by tag name or `path`. Returns a list containing all matching elements in document order. `namespaces` is an optional mapping from namespace prefix to full name.

**findtext(match, default=None, namespaces=None)**

Finds text for the first subelement matching `match`. `match` may be a tag name or a `path`. Returns the text content of the first matching element, or `default` if no element was found. Note that if the
matching element has no text content an empty string is returned. *namespaces* is an optional mapping from namespace prefix to full name.

**getchildren()**
Deprecated since version 3.2: Use `list(elem)` or iteration.

**getiterator**(tag=None)
Deprecated since version 3.2: Use method `Element.iter()` instead.

**insert**(index, subelement)
Inserts `subelement` at the given position in this element. Raises `TypeError` if `subelement` is not an `Element`.

**iter**(tag=None)
Creates a tree iterator with the current element as the root. The iterator iterates over this element and all elements below it, in document (depth first) order. If `tag` is not `None` or `'*'`, only elements whose tag equals `tag` are returned from the iterator. If the tree structure is modified during iteration, the result is undefined. New in version 3.2.

**iterfind**(match, namespaces=None)
Finds all matching subelements, by tag name or *path*. Returns an iterable yielding all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name. New in version 3.2.

**itertext**()
Creates a text iterator. The iterator loops over this element and all subelements, in document order, and returns all inner text. New in version 3.2.

**makeelement**(tag, attrib)
Creates a new element object of the same type as this element. Do not call this method, use the `SubElement()` factory function instead.

**remove**(subelement)
Removes `subelement` from the element. Unlike the find* methods this method compares elements based on the instance identity, not on tag value or contents.

*Element* objects also support the following sequence type methods for working with subelements: `__delitem__()`, `__getitem__()`, `__setitem__()`, `__len__()`. Caution: Elements with no subelements will test as `False`. This behavior will change in future versions. Use specific `len(elem)` or `elem is None` test instead.

```python
element = root.find('foo')

if not element:  # careful!
    print("element not found, or element has no subelements")

if element is None:
    print("element not found")
```

**ElementTree Objects**

**class** `xml.etree.ElementTree.ElementTree(element=None, file=None)`
ElementTree wrapper class. This class represents an entire element hierarchy, and adds some extra support for serialization to and from standard XML.

*element* is the root element. The tree is initialized with the contents of the XML *file* if given.

**_setroot**(element)
Replaces the root element for this tree. This discards the current contents of the tree, and replaces it with the given element. Use with care. *element* is an element instance.

**find**(match, namespaces=None)
Same as `Element.find()`, starting at the root of the tree.
**findall** *(match, namespaces=None)*

Same as `Element.findall()`, starting at the root of the tree.

**findtext** *(match, default=None, namespaces=None)*

Same as `Element.findtext()`, starting at the root of the tree.

**getiterator** *(tag=None)*

Deprecated since version 3.2: Use method `ElementTree.iter()` instead.

**getroot**

Returns the root element for this tree.

**iter** *(tag=None)*

Creates and returns a tree iterator for the root element. The iterator loops over all elements in this tree, in section order. `tag` is the tag to look for (default is to return all elements)

**iterfind** *(match, namespaces=None)*

Same as `Element.iterfind()`, starting at the root of the tree. New in version 3.2.

**parse** *(source, parser=None)*

Loads an external XML section into this element tree. `source` is a file name or file object, `parser` is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns the section root element.

**write** *(file, encoding="us-ascii", xml_declaration=None, default_namespace=None, method="xml")*

Writes the element tree to a file, as XML. `file` is a file name, or a file object opened for writing. `encoding` is the output encoding (default is US-ASCII). `xml_declaration` controls if an XML declaration should be added to the file. Use `False` for never, `True` for always, `None` for only if not US-ASCII or UTF-8 or Unicode (default is `None`). `default_namespace` sets the default XML namespace (for "xmlns"). `method` is either "xml", "html" or "text" (default is "xml").

The output is either a string (str) or binary (bytes). This is controlled by the `encoding` argument. If `encoding` is "unicode", the output is a string; otherwise, it’s binary. Note that this may conflict with the type of `file` if it’s an open file object; make sure you do not try to write a string to a binary stream and vice versa.

This is the XML file that is going to be manipulated:

```xml
<html>
  <head>
    <title>Example page</title>
  </head>
  <body>
    <p>Moved to <a href="http://example.org/">example.org</a> or <a href="http://example.com/">example.com</a>.</p>
  </body>
</html>
```

Example of changing the attribute “target” of every link in first paragraph:

```python
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")  # Finds first occurrence of tag p in body
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))  # Returns list of all links
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77ec1cc>]
>>> for i in links:  # Iterates through all found links
...   i.attrib["target"] = "blank"
>>> tree.write("output.xhtml")
```
QName Objects

class xml.etree.ElementTree.QName (text_or_uri, tag=None)

QName wrapper. This can be used to wrap a QName attribute value, in order to get proper namespace handling on output. text_or_uri is a string containing the QName value, in the form {uri}local, or, if the tag argument is given, the URI part of a QName. If tag is given, the first argument is interpreted as an URI, and this argument is interpreted as a local name. QName instances are opaque.

TreeBuilder Objects

class xml.etree.ElementTree.TreeBuilder (element_factory=None)

Generic element structure builder. This builder converts a sequence of start, data, and end method calls to a well-formed element structure. You can use this class to build an element structure using a custom XML parser, or a parser for some other XML-like format. element_factory, when given, must be a callable accepting two positional arguments: a tag and a dict of attributes. It is expected to return a new element instance.

close ()

Flushes the builder buffers, and returns the toplevel document element. Returns an Element instance.

data (data)

Adds text to the current element. data is a string. This should be either a bytestring, or a Unicode string.

der (tag)

Closes the current element. tag is the element name. Returns the closed element.

start (tag, attrs)

Opens a new element. tag is the element name. attrs is a dictionary containing element attributes. Returns the opened element.

In addition, a custom TreeBuilder object can provide the following method:

doctype (name, pubid, system)

Handles a doctype declaration. name is the doctype name. pubid is the public identifier. system is the system identifier. This method does not exist on the default TreeBuilder class. New in version 3.2.

XMLParser Objects

class xml.etree.ElementTree.XMLParser (html=0, target=None, encoding=None)

Element structure builder for XML source data, based on the expat parser. html are predefined HTML entities. This flag is not supported by the current implementation. target is the target object. If omitted, the builder uses an instance of the standard TreeBuilder class. encoding is optional. If given, the value overrides the encoding specified in the XML file.

close ()

Finishes feeding data to the parser. Returns the result of calling the close() method of the target passed during construction; by default, this is the toplevel document element.

doctype (name, pubid, system)

Deprecated since version 3.2: Define the TreeBuilder.doctype() method on a custom TreeBuilder target.

feed (data)

Feeds data to the parser. data is encoded data.

XMLParser.feed() calls target's start() method for each opening tag, its end() method for each closing tag, and data is processed by method data(). XMLParser.close() calls target's method close(). XMLParser can be used not only for building a tree structure. This is an example of counting the maximum depth of an XML file:
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:
...     # The target object of the parser
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):
...         # Called for each opening tag.
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):
...         # Called for each closing tag.
...         self.depth -= 1
...     def data(self, data):
...         pass
...     # We do not need to do anything with data.
...     def close(self):
...         # Called when all data has been parsed.
...         return self.maxDepth
...     
>>> target = MaxDepth()
>>> parser = XMLParser(target=target)
>>> exampleXml = """"""
...     <a>
...         <b>
...             <c>
...                 <d>
...             </d>
...         </c>
...     </b>
"""
>>> parser.feed(exampleXml)
>>> parser.close()
4

Exceptions

class xml.etree.ElementTree.ParseError
XML parse error, raised by the various parsing methods in this module when parsing fails. The string representation of an instance of this exception will contain a user-friendly error message. In addition, it will have the following attributes available:

    code
A numeric error code from the expat parser. See the documentation of xml.parsers.expat for the list of error codes and their meanings.

    position
A tuple of line, column numbers, specifying where the error occurred.

20.7 xml.dom — The Document Object Model API

The Document Object Model, or “DOM,” is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying XML documents. A DOM implementation presents an XML document as a tree structure, or allows client code to build such a structure from scratch. It then gives access to the structure through a set of objects which provided well-known interfaces.

The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another. If you are looking at a text node, you have no access to a containing element. When you write a SAX application, you need to keep track
of your program’s position in the document somewhere in your own code. SAX does not do it for you. Also, if you need to look ahead in the XML document, you are just out of luck.

Some applications are simply impossible in an event driven model with no access to a tree. Of course you could build some sort of tree yourself in SAX events, but the DOM allows you to avoid writing that code. The DOM is a standard tree representation for XML data.

The Document Object Model is being defined by the W3C in stages, or “levels” in their terminology. The Python mapping of the API is substantially based on the DOM Level 2 recommendation.

DOM applications typically start by parsing some XML into a DOM. How this is accomplished is not covered at all by DOM Level 1, and Level 2 provides only limited improvements: There is a DOMImplementation object class which provides access to Document creation methods, but no way to access an XML reader/parser/Document builder in an implementation-independent way. There is also no well-defined way to access these methods without an existing Document object. In Python, each DOM implementation will provide a function getDOMImplementation(). DOM Level 3 adds a Load/Store specification, which defines an interface to the reader, but this is not yet available in the Python standard library.

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification; this portion of the reference manual describes the interpretation of the specification in Python.

The specification provided by the W3C defines the DOM API for Java, ECMAScript, and OMG IDL. The Python mapping defined here is based in large part on the IDL version of the specification, but strict compliance is not required (though implementations are free to support the strict mapping from IDL). See section Conformance for a detailed discussion of mapping requirements.

See Also:

Document Object Model (DOM) Level 2 Specification The W3C recommendation upon which the Python DOM API is based.

Document Object Model (DOM) Level 1 Specification The W3C recommendation for the DOM supported by xml.dom.minidom.

Python Language Mapping Specification This specifies the mapping from OMG IDL to Python.

20.7.1 Module Contents

The xml.dom contains the following functions:

xml.dom.registerDOMImplementation(name, factory)
   Register the factory function with the name name. The factory function should return an object which implements the DOMImplementation interface. The factory function can return the same object every time, or a new one for each call, as appropriate for the specific implementation (e.g. if that implementation supports some customization).

xml.dom.getDOMImplementation(name=None, features=())
   Return a suitable DOM implementation. The name is either well-known, the module name of a DOM implementation, or None. If it is not None, imports the corresponding module and returns a DOMImplementation object if the import succeeds. If no name is given, and if the environment variable PYTHON_DOM is set, this variable is used to find the implementation.

   If name is not given, this examines the available implementations to find one with the required feature set. If no implementation can be found, raise an ImportError. The features list must be a sequence of (feature, version) pairs which are passed to the hasFeature() method on available DOMImplementation objects.

Some convenience constants are also provided:

xml.dom.EMPTY_NAMESPACE
   The value used to indicate that no namespace is associated with a node in the DOM. This is typically found as the namespaceURI of a node, or used as the namespaceURI parameter to a namespaces-specific method.
xml.dom.XML_NAMESPACE

The namespace URI associated with the reserved prefix xml, as defined by Namespaces in XML (section 4).

xml.dom.XMLNS_NAMESPACE

The namespace URI for namespace declarations, as defined by Document Object Model (DOM) Level 2 Core Specification (section 1.1.8).

xml.dom.XHTML_NAMESPACE

The URI of the XHTML namespace as defined by XHTML 1.0: The Extensible HyperText Markup Language (section 3.1.1).

In addition, xml.dom contains a base Node class and the DOM exception classes. The Node class provided by this module does not implement any of the methods or attributes defined by the DOM specification; concrete DOM implementations must provide those. The Node class provided as part of this module does provide the constants used for the nodeType attribute on concrete Node objects; they are located within the class rather than at the module level to conform with the DOM specifications.

### 20.7.2 Objects in the DOM

The definitive documentation for the DOM is the DOM specification from the W3C.

Note that DOM attributes may also be manipulated as nodes instead of as simple strings. It is fairly rare that you must do this, however, so this usage is not yet documented.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Section</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMImplementation</td>
<td>DOMImplementation</td>
<td>Interface to the underlying implementation.</td>
</tr>
<tr>
<td>Node</td>
<td>Node Objects</td>
<td>Base interface for most objects in a document.</td>
</tr>
<tr>
<td>NodeList</td>
<td>NodeList Objects</td>
<td>Interface for a sequence of nodes.</td>
</tr>
<tr>
<td>DocumentType</td>
<td>DocumentType Objects</td>
<td>Information about the declarations needed to process a document.</td>
</tr>
<tr>
<td>Document</td>
<td>Document Objects</td>
<td>Object which represents an entire document.</td>
</tr>
<tr>
<td>Element</td>
<td>Element Objects</td>
<td>Element nodes in the document hierarchy.</td>
</tr>
<tr>
<td>Attr</td>
<td>Attr Objects</td>
<td>Attribute value nodes on element nodes.</td>
</tr>
<tr>
<td>Comment</td>
<td>Comment Objects</td>
<td>Representation of comments in the source document.</td>
</tr>
<tr>
<td>Text</td>
<td>Text and CDATASection Objects</td>
<td>Nodes containing textual content from the document.</td>
</tr>
<tr>
<td>ProcessingInstruction</td>
<td>ProcessingInstruction Objects</td>
<td>Processing instruction representation.</td>
</tr>
</tbody>
</table>

An additional section describes the exceptions defined for working with the DOM in Python.

**DOMImplementation Objects**

The DOMImplementation interface provides a way for applications to determine the availability of particular features in the DOM they are using. DOM Level 2 added the ability to create new Document and DocumentType objects using the DOMImplementation as well.

```python
DOMImplementation.hasFeature(feature, version)
```

Return true if the feature identified by the pair of strings feature and version is implemented.

```python
DOMImplementation.createDocument(namespaceUri, qualifiedName, doctype)
```

Return a new Document object (the root of the DOM), with a child Element object having the given namespaceUri and qualifiedName. The doctype must be a DocumentType object created by createDocumentType(), or None. In the Python DOM API, the first two arguments can also be None in order to indicate that no Element child is to be created.
DOMImplementation.createDocumentType(qualifiedName, publicId, systemId)

Return a new DocumentType object that encapsulates the given qualifiedName, publicId, and systemId strings, representing the information contained in an XML document type declaration.

Node Objects

All of the components of an XML document are subclasses of Node.

Node.nodeType
An integer representing the node type. Symbolic constants for the types are on the Node object: ELEMENT_NODE, ATTRIBUTE_NODE, TEXT_NODE, CDATA_SECTION_NODE, ENTITY_NODE, PROCESSING_INSTRUCTION_NODE, COMMENT_NODE, DOCUMENT_NODE, DOCUMENT_TYPE_NODE, NOTATION_NODE. This is a read-only attribute.

Node.parentNode
The parent of the current node, or None for the document node. The value is always a Node object or None. For Element nodes, this will be the parent element, except for the root element, in which case it will be the Document object. For Attr nodes, this is always None. This is a read-only attribute.

Node.attributes
A NamedNodeMap of attribute objects. Only elements have actual values for this; others provide None for this attribute. This is a read-only attribute.

Node.previousSibling
The node that immediately precedes this one with the same parent. For instance the element with an end-tag that comes just before the self element’s start-tag. Of course, XML documents are made up of more than just elements so the previous sibling could be text, a comment, or something else. If this node is the first child of the parent, this attribute will be None. This is a read-only attribute.

Node.nextSibling
The node that immediately follows this one with the same parent. See also previousSibling. If this is the last child of the parent, this attribute will be None. This is a read-only attribute.

Node.childNodes
A list of nodes contained within this node. This is a read-only attribute.

Node.firstChild
The first child of the node, if there are any, or None. This is a read-only attribute.

Node.lastChild
The last child of the node, if there are any, or None. This is a read-only attribute.

Node.localName
The part of the tagName following the colon if there is one, else the entire tagName. The value is a string.

Node.prefix
The part of the tagName preceding the colon if there is one, else the empty string. The value is a string, or None

Node.namespaceURI
The namespace associated with the element name. This will be a string or None. This is a read-only attribute.

Node.nodeName
This has a different meaning for each node type; see the DOM specification for details. You can always get the information you would get here from another property such as the tagName property for elements or the name property for attributes. For all node types, the value of this attribute will be either a string or None. This is a read-only attribute.

Node.nodeValue
This has a different meaning for each node type; see the DOM specification for details. The situation is similar to that with nodeName. The value is a string or None.
Node.\texttt{hasAttributes()}\newline
Returns true if the node has any attributes.

Node.\texttt{hasChildNodes()}\newline
Returns true if the node has any child nodes.

Node.\texttt{isSameNode}(\texttt{other})\newline
Returns true if \texttt{other} refers to the same node as this node. This is especially useful for DOM implementations which use any sort of proxy architecture (because more than one object can refer to the same node).

\textbf{Note:}\hspace{1em}This is based on a proposed DOM Level 3 API which is still in the “working draft” stage, but this particular interface appears uncontroversial. Changes from the W3C will not necessarily affect this method in the Python DOM interface (though any new W3C API for this would also be supported).

Node.\texttt{appendChild}(\texttt{newChild})\newline
Add a new child node to this node at the end of the list of children, returning \texttt{newChild}. If the node was already in the tree, it is removed first.

Node.\texttt{insertBefore}(\texttt{newChild}, \texttt{refChild})\newline
Insert a new child node before an existing child. It must be the case that \texttt{refChild} is a child of this node; if not, \texttt{ValueError} is raised. \texttt{newChild} is returned. If \texttt{refChild} is \texttt{None}, it inserts \texttt{newChild} at the end of the children’s list.

Node.\texttt{removeChild}(\texttt{oldChild})\newline
Remove a child node. \texttt{oldChild} must be a child of this node; if not, \texttt{ValueError} is raised. \texttt{oldChild} is returned on success. If \texttt{oldChild} will not be used further, its \texttt{unlink()} method should be called.

Node.\texttt{replaceChild}(\texttt{newChild}, \texttt{oldChild})\newline
Replace an existing node with a new node. It must be the case that \texttt{oldChild} is a child of this node; if not, \texttt{ValueError} is raised.

Node.\texttt{normalize()}\newline
Join adjacent text nodes so that all stretches of text are stored as single \texttt{Text} instances. This simplifies processing text from a DOM tree for many applications.

Node.\texttt{cloneNode}(\texttt{deep})\newline
Clone this node. Setting \texttt{deep} means to clone all child nodes as well. This returns the clone.

\subsection*{NodeList Objects}

A \texttt{NodeList} represents a sequence of nodes. These objects are used in two ways in the DOM Core recommendation: the \texttt{Element} objects provides one as its list of child nodes, and the \texttt{getElementsByTagName()} and \texttt{getElementsByTagNameNS()} methods of \texttt{Node} return objects with this interface to represent query results.

The DOM Level 2 recommendation defines one method and one attribute for these objects:

\begin{itemize}
  \item \texttt{NodeList.\texttt{item}(i)}\newline
    Return the \texttt{i}’th item from the sequence, if there is one, or \texttt{None}. The index \texttt{i} is not allowed to be less then zero or greater than or equal to the length of the sequence.
  \item \texttt{NodeList.\texttt{length}}\newline
    The number of nodes in the sequence.
\end{itemize}

In addition, the Python DOM interface requires that some additional support is provided to allow \texttt{NodeList} objects to be used as Python sequences. All \texttt{NodeList} implementations must include support for \texttt{\_\_len\_()} and \texttt{\_\_getitem\_()}; this allows iteration over the \texttt{NodeList} in \texttt{for} statements and proper support for the \texttt{len()} built-in function.

If a DOM implementation supports modification of the document, the \texttt{NodeList} implementation must also support the \texttt{\_\_setitem\_()} and \texttt{\_\_delitem\_()} methods.
**DocumentType Objects**

Information about the notations and entities declared by a document (including the external subset if the parser uses it and can provide the information) is available from a `DocumentType` object. The `DocumentType` for a document is available from the `Document` object's `doctype` attribute; if there is no DOCTYPE declaration for the document, the document’s `doctype` attribute will be set to `None` instead of an instance of this interface.

`DocumentType` is a specialization of `Node`, and adds the following attributes:

- **DocumentType.publicId**  
  The public identifier for the external subset of the document type definition. This will be a string or `None`.

- **DocumentType.systemId**  
  The system identifier for the external subset of the document type definition. This will be a URI as a string, or `None`.

- **DocumentType.internalSubset**  
  A string giving the complete internal subset from the document. This does not include the brackets which enclose the subset. If the document has no internal subset, this should be `None`.

- **DocumentType.name**  
  The name of the root element as given in the DOCTYPE declaration, if present.

- **DocumentType.entities**  
  This is a `NamedNodeMap` giving the definitions of external entities. For entity names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be `None` if the information is not provided by the parser, or if no entities are defined.

- **DocumentType.notations**  
  This is a `NamedNodeMap` giving the definitions of notations. For notation names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be `None` if the information is not provided by the parser, or if no notations are defined.

**Document Objects**

A `Document` represents an entire XML document, including its constituent elements, attributes, processing instructions, comments etc. Remember that it inherits properties from `Node`.

- **Document.documentElement**  
  The one and only root element of the document.

- **Document.createElement**(tagName)  
  Create and return a new element node. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as `insertBefore()` or `appendChild()`.

- **Document.createElementNS**(namespaceURI, tagName)  
  Create and return a new element with a namespace. The `tagName` may have a prefix. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as `insertBefore()` or `appendChild()`.

- **Document.createTextNode**(data)  
  Create and return a text node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

- **Document.createComment**(data)  
  Create and return a comment node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

- **Document.createProcessingInstruction**(target, data)  
  Create and return a processing instruction node containing the `target` and `data` passed as parameters. As with the other creation methods, this one does not insert the node into the tree.

- **Document.createAttribute**(name)  
  Create and return an attribute node. This method does not associate the attribute node with any particular
element. You must use `setAttributeNode()` on the appropriate `Element` object to use the newly created attribute instance.

**Document.createElementNS**(namespaceURI, qualifiedName)
Create and return an attribute node with a namespace. The `tagName` may have a prefix. This method does not associate the attribute node with any particular element. You must use `setAttributeNode()` on the appropriate `Element` object to use the newly created attribute instance.

**Document.getElementsByTagName**(tagName)
Search for all descendants (direct children, children’s children, etc.) with a particular element type name.

**Document.getElementsByTagNameNS**(namespaceURI, localName)
Search for all descendants (direct children, children’s children, etc.) with a particular namespace URI and localname. The localname is the part of the namespace after the prefix.

**Element Objects**

Element is a subclass of Node, so inherits all the attributes of that class.

**Element.tagName**
The element type name. In a namespace-using document it may have colons in it. The value is a string.

**Element.getElementsByTagName**(tagName)
Same as equivalent method in the Document class.

**Element.getElementsByTagNameNS**(namespaceURI, localName)
Same as equivalent method in the Document class.

**Element.hasAttribute**(name)
Returns true if the element has an attribute named by name.

**Element.hasAttributeNS**(namespaceURI, localName)
Returns true if the element has an attribute named by namespaceURI and localName.

**Element.getAttribute**(name)
Return the value of the attribute named by name as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

**Element.getAttributeNode**(attrname)
Return the Attr node for the attribute named by attrname.

**Element.getAttributeNS**(namespaceURI, localName)
Return the value of the attribute named by namespaceURI and localName as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

**Element.getAttributeNodeNS**(namespaceURI, localName)
Return an attribute value as a node, given a namespaceURI and localName.

**Element.removeAttribute**(name)
Remove an attribute by name. If there is no matching attribute, a NotFoundErr is raised.

**Element.removeAttributeNode**(oldAttr)
Remove and return oldAttr from the attribute list, if present. If oldAttr is not present, NotFoundErr is raised.

**Element.removeAttributeNS**(namespaceURI, localName)
Remove an attribute by name. Note that it uses a localName, not a qname. No exception is raised if there is no matching attribute.

**Element.setAttribute**(name, value)
Set an attribute value from a string.

**Element.setAttributeNode**(newAttr)
Add a new attribute node to the element, replacing an existing attribute if necessary if the name attribute matches. If a replacement occurs, the old attribute node will be returned. If newAttr is already in use, InuseAttributeErr will be raised.
**Element.setAttributeNodeNS** *(newAttr)*

Add a new attribute node to the element, replacing an existing attribute if necessary if the namespaceURI and localName attributes match. If a replacement occurs, the old attribute node will be returned. If newAttr is already in use, InuseAttributeErr will be raised.

**Element.setAttributeNS** *(namespaceURI, qname, value)*

Set an attribute value from a string, given a namespaceURI and a qname. Note that a qname is the whole attribute name. This is different than above.

### Attr Objects

Attr inherits from Node, so inherits all its attributes.

**Attr.name**

The attribute name. In a namespace-using document it may include a colon.

**Attr.localName**

The part of the name following the colon if there is one, else the entire name. This is a read-only attribute.

**Attr.prefix**

The part of the name preceding the colon if there is one, else the empty string.

**Attr.value**

The text value of the attribute. This is a synonym for the nodeValue attribute.

### NamedNodeMap Objects

NamedNodeMap does not inherit from Node.

**NamedNodeMap.length**

The length of the attribute list.

**NamedNodeMap.item(index)**

Return an attribute with a particular index. The order you get the attributes in is arbitrary but will be consistent for the life of a DOM. Each item is an attribute node. Get its value with the value attribute.

There are also experimental methods that give this class more mapping behavior. You can use them or you can use the standardized getAttribute*() family of methods on the Element objects.

### Comment Objects

Comment represents a comment in the XML document. It is a subclass of Node, but cannot have child nodes.

**Comment.data**

The content of the comment as a string. The attribute contains all characters between the leading <!-- and trailing -->, but does not include them.

### Text and CDATASection Objects

The Text interface represents text in the XML document. If the parser and DOM implementation support the DOM’s XML extension, portions of the text enclosed in CDATA marked sections are stored in CDATASection objects. These two interfaces are identical, but provide different values for the nodeType attribute.

These interfaces extend the Node interface. They cannot have child nodes.

**Text.data**

The content of the text node as a string.

**Note:** The use of a CDATASection node does not indicate that the node represents a complete CDATA marked section, only that the content of the node was part of a CDATA section. A single CDATA section may be
represented by more than one node in the document tree. There is no way to determine whether two adjacent CDATASection nodes represent different CDATA marked sections.

**ProcessingInstruction Objects**

Represents a processing instruction in the XML document; this inherits from the Node interface and cannot have child nodes.

- **ProcessingInstruction.target**
  - The content of the processing instruction up to the first whitespace character. This is a read-only attribute.

- **ProcessingInstruction.data**
  - The content of the processing instruction following the first whitespace character.

**Exceptions**

The DOM Level 2 recommendation defines a single exception, DOMException, and a number of constants that allow applications to determine what sort of error occurred. DOMException instances carry a code attribute that provides the appropriate value for the specific exception.

The Python DOM interface provides the constants, but also expands the set of exceptions so that a specific exception exists for each of the exception codes defined by the DOM. The implementations must raise the appropriate specific exception, each of which carries the appropriate value for the code attribute.

- **exception xml.dom.DOMException**
  - Base exception class used for all specific DOM exceptions. This exception class cannot be directly instantiated.

- **exception xml.dom.DomStringSizeErr**
  - Raised when a specified range of text does not fit into a string. This is not known to be used in the Python DOM implementations, but may be received from DOM implementations not written in Python.

- **exception xml.dom.HierarchyRequestErr**
  - Raised when an attempt is made to insert a node where the node type is not allowed.

- **exception xml.dom.IndexSizeErr**
  - Raised when an index or size parameter to a method is negative or exceeds the allowed values.

- **exception xml.dom.InuseAttributeErr**
  - Raised when an attempt is made to insert an Attr node that is already present elsewhere in the document.

- **exception xml.dom.InvalidAccessErr**
  - Raised if a parameter or an operation is not supported on the underlying object.

- **exception xml.dom.InvalidCharacterErr**
  - This exception is raised when a string parameter contains a character that is not permitted in the context it’s being used in by the XML 1.0 recommendation. For example, attempting to create an Element node with a space in the element type name will cause this error to be raised.

- **exception xml.dom.InvalidModificationErr**
  - Raised when an attempt is made to modify the type of a node.

- **exception xml.dom.InvalidStateErr**
  - Raised when an attempt is made to use an object that is not defined or is no longer usable.

- **exception xml.dom.NamespaceErr**
  - If an attempt is made to change any object in a way that is not permitted with regard to the Namespaces in XML recommendation, this exception is raised.

- **exception xml.dom.NotFoundErr**
  - Exception when a node does not exist in the referenced context. For example, NamedNodeMap.removeNamedItem() will raise this if the node passed in does not exist in the map.
exception xml.dom.NotSupportedException
   Raised when the implementation does not support the requested type of object or operation.

exception xml.dom.NoDataAllowedErr
   This is raised if data is specified for a node which does not support data.

exception xml.dom.NoModificationAllowedErr
   Raised on attempts to modify an object where modifications are not allowed (such as for read-only nodes).

exception xml.dom.SyntaxErr
   Raised when an invalid or illegal string is specified.

exception xml.dom.WrongDocumentErr
   Raised when a node is inserted in a different document than it currently belongs to, and the implementation
does not support migrating the node from one document to the other.

The exception codes defined in the DOM recommendation map to the exceptions described above according to
this table:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMSTRING_SIZE_ERR</td>
<td>DomstringSizeErr</td>
</tr>
<tr>
<td>HIERARCHY_REQUEST_ERR</td>
<td>HierarchyRequestErr</td>
</tr>
<tr>
<td>INDEX_SIZE_ERR</td>
<td>IndexSizeErr</td>
</tr>
<tr>
<td>INUSE_ATTRIBUTE_ERR</td>
<td>InuseAttributeErr</td>
</tr>
<tr>
<td>INVALID_ACCESS_ERR</td>
<td>InvalidAccessErr</td>
</tr>
<tr>
<td>INVALID_CHARACTER_ERR</td>
<td>InvalidCharacterErr</td>
</tr>
<tr>
<td>INVALID_MODIFICATION_ERR</td>
<td>InvalidModificationErr</td>
</tr>
<tr>
<td>INVALID_STATE_ERR</td>
<td>InvalidStateErr</td>
</tr>
<tr>
<td>NAMESPACE_ERR</td>
<td>NamespaceErr</td>
</tr>
<tr>
<td>NOT_FOUND_ERR</td>
<td>NotFoundErr</td>
</tr>
<tr>
<td>NOT_SUPPORTED_ERR</td>
<td>NotSupportedErr</td>
</tr>
<tr>
<td>NO_DATA_ALLOWED_ERR</td>
<td>NoDataAllowedErr</td>
</tr>
<tr>
<td>NO_MODIFICATION_ALLOWED_ERR</td>
<td>NoModificationAllowedErr</td>
</tr>
<tr>
<td>SYNTAX_ERR</td>
<td>SyntaxErr</td>
</tr>
<tr>
<td>WRONG_DOCUMENT_ERR</td>
<td>WrongDocumentErr</td>
</tr>
</tbody>
</table>

20.7.3 Conformance

This section describes the conformance requirements and relationships between the Python DOM API, the W3C
DOM recommendations, and the OMG IDL mapping for Python.

Type Mapping

The IDL types used in the DOM specification are mapped to Python types according to the following table.

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Python Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>bool or int</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>long int</td>
<td>int</td>
</tr>
<tr>
<td>unsigned int</td>
<td>int</td>
</tr>
<tr>
<td>DOMString</td>
<td>str or bytes</td>
</tr>
<tr>
<td>null</td>
<td>None</td>
</tr>
</tbody>
</table>

Accessor Methods

The mapping from OMG IDL to Python defines accessor functions for IDL attribute declarations in much
the way the Java mapping does. Mapping the IDL declarations

```python
readonly attribute string someValue;
attribute string anotherValue;
```
yields three accessor functions: a “get” method for `someValue (_get_someValue())`, and “get” and “set” methods for `anotherValue (_get_anotherValue() and _set_anotherValue())`. The mapping, in particular, does not require that the IDL attributes are accessible as normal Python attributes: `object.someValue` is not required to work, and may raise an `AttributeError`.

The Python DOM API, however, does require that normal attribute access work. This means that the typical surrogates generated by Python IDL compilers are not likely to work, and wrapper objects may be needed on the client if the DOM objects are accessed via CORBA. While this does require some additional consideration for CORBA DOM clients, the implementers with experience using DOM over CORBA do not consider this a problem. Attributes that are declared `readonly` may not restrict write access in all DOM implementations.

In the Python DOM API, accessor functions are not required. If provided, they should take the form defined by the Python IDL mapping, but these methods are considered unnecessary since the attributes are accessible directly from Python. “Set” accessors should never be provided for `readonly` attributes.

The IDL definitions do not fully embody the requirements of the W3C DOM API, such as the notion of certain objects, such as the return value of `getElementsByTagName()`, being “live”. The Python DOM API does not require implementations to enforce such requirements.

## 20.8 xml.dom.minidom — Minimal DOM implementation

### Source code: Lib/xml/dom/minidom.py

xml.dom.minidom is a minimal implementation of the Document Object Model interface, with an API similar to that in other languages. It is intended to be simpler than the full DOM and also significantly smaller. Users who are not already proficient with the DOM should consider using the `xml.etree.ElementTree` module for their XML processing instead.

**Warning:** The `xml.dom.minidom` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see [XML vulnerabilities](#).

DOM applications typically start by parsing some XML into a DOM. With `xml.dom.minidom`, this is done through the parse functions:

```python
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\temp\mydata.xml')  # parse an XML file by name

datasource = open('c:\temp\mydata.xml')
dom2 = parse(datasource)  # parse an open file

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')
```

The `parse()` function can take either a filename or an open file object.

```python
xml.dom.minidom.parse (filename_or_file, parser=None, bufsize=None)
```

Return a Document from the given input. `filename_or_file` may be either a file name, or a file-like object. `parser`, if given, must be a SAX2 parser object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

If you have XML in a string, you can use the `parseString()` function instead:

```python
xml.dom.minidom.parseString (string, parser=None)
```

Return a Document that represents the string. This method creates a `io.StringIO` object for the string and passes that on to `parse()`.

Both functions return a `Document` object representing the content of the document.
What the `parse()` and `parseString()` functions do is connect an XML parser with a “DOM builder” that can accept parse events from any SAX parser and convert them into a DOM tree. The name of the functions are perhaps misleading, but are easy to grasp when learning the interfaces. The parsing of the document will be completed before these functions return; it’s simply that these functions do not provide a parser implementation themselves.

You can also create a `Document` by calling a method on a “DOM Implementation” object. You can get this object either by calling the `getDOMImplementation()` function in the `xml.dom` package or the `xml.dom.minidom` module. Once you have a `Document`, you can add child nodes to it to populate the DOM:

```python
from xml.dom.minidom import getDOMImplementation

impl = getDOMImplementation()
newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement

for text in 'Some textual content.':
    top_element.appendChild(text)
```

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification. The main property of the document object is the `documentElement` property. It gives you the main element in the XML document: the one that holds all others. Here is an example program:

```python
dom3 = parseString("<myxml>Some data</myxml>"
assert dom3.documentElement.tagName == "myxml"
```

When you are finished with a DOM tree, you may optionally call the `unlink()` method to encourage early cleanup of the now-unneeded objects. `unlink()` is a `xml.dom.minidom`-specific extension to the DOM API that renders the node and its descendants are essentially useless. Otherwise, Python’s garbage collector will eventually take care of the objects in the tree.

See Also:

- Document Object Model (DOM) Level 1 Specification  The W3C recommendation for the DOM supported by `xml.dom.minidom`.

## 20.8.1 DOM Objects

The definition of the DOM API for Python is given as part of the `xml.dom` module documentation. This section lists the differences between the API and `xml.dom.minidom`.

**Node. `unlink()`**

Break internal references within the DOM so that it will be garbage collected on versions of Python without cyclic GC. Even when cyclic GC is available, using this can make large amounts of memory available sooner, so calling this on DOM objects as soon as they are no longer needed is good practice. This only needs to be called on the `Document` object, but may be called on child nodes to discard children of that node.

You can avoid calling this method explicitly by using the `with` statement. The following code will automatically `unlink` `dom` when the `with` block is exited:

```python
with xml.dom.minidom.parse(datasource) as dom:
    ... # Work with dom.
```

**Node. `writexml()`**

Write XML to the writer object. The writer should have a `write()` method which matches that of the file object interface. The `indent` parameter is the indentation of the current node. The `addindent` parameter is the incremental indentation to use for subnodes of the current one. The `newl` parameter specifies the string to use to terminate newlines.
For the `Document` node, an additional keyword argument `encoding` can be used to specify the encoding field of the XML header.

`Node.toxml(encoding=None)`
Return a string or byte string containing the XML represented by the DOM node.

With an explicit `encoding` argument, the result is a byte string in the specified encoding. With no `encoding` argument, the result is a Unicode string, and the XML declaration in the resulting string does not specify an encoding. Encoding this string in an encoding other than UTF-8 is likely incorrect, since UTF-8 is the default encoding of XML.

`Node,toprettyxml(indent="", newl="", encoding="")`
Return a pretty-printed version of the document. `indent` specifies the indentation string and defaults to a tabulator; `newl` specifies the string emitted at the end of each line and defaults to 

The `encoding` argument behaves like the corresponding argument of `toxml()`.

### 20.8.2 DOM Example

This example program is a fairly realistic example of a simple program. In this particular case, we do not take much advantage of the flexibility of the DOM.

```python
import xml.dom.minidom

document = """"\n<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point>
</slide>

<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
"""

dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = []
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc.append(node.data)
    return ''.join(rc)

def handleSlideshow(slideshow):
    print("<html>")
    handleSlideshowTitle(slideshow.getElementsByTagName("title")[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print("</html>")

4 The encoding name included in the XML output should conform to the appropriate standards. For example, “UTF-8” is valid, but “UTF8” is not valid in an XML document’s declaration, even though Python accepts it as an encoding name. See [http://www.w3.org/TR/2006/REC-xml11-20060816#NT-EncodingDecl](http://www.w3.org/TR/2006/REC-xml11-20060816#NT-EncodingDecl) and [http://www.iana.org/assignments/character-sets](http://www.iana.org/assignments/character-sets).
def handleSlides(slides):
    for slide in slides:
        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print("<title>%s</title>" % getText(title.childNodes))

def handleSlideTitle(title):
    print("<h2>%s</h2>" % getText(title.childNodes))

def handlePoints(points):
    print("<ul>")
    for point in points:
        handlePoint(point)
    print("</ul>")

def handlePoint(point):
    print("<li>%s</li>" % getText(point.childNodes))

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print("<p>%s</p>" % getText(title.childNodes))

handleSlideshow(dom)

20.8.3 minidom and the DOM standard

The xml.dom.minidom module is essentially a DOM 1.0-compatible DOM with some DOM 2 features (primarily namespace features).

Usage of the DOM interface in Python is straight-forward. The following mapping rules apply:

- Interfaces are accessed through instance objects. Applications should not instantiate the classes themselves; they should use the creator functions available on the Document object. Derived interfaces support all operations (and attributes) from the base interfaces, plus any new operations.

- Operations are used as methods. Since the DOM uses only in parameters, the arguments are passed in normal order (from left to right). There are no optional arguments. void operations return None.

- IDL attributes map to instance attributes. For compatibility with the OMG IDL language mapping for Python, an attribute foo can also be accessed through accessor methods _get_foo() and _set_foo(). readonly attributes must not be changed; this is not enforced at runtime.

- The types short int, unsigned int, unsigned long long, and boolean all map to Python integer objects.

- The type DOMString maps to Python strings. xml.dom.minidom supports either bytes or strings, but will normally produce strings. Values of type DOMString may also be None where allowed to have the IDL null value by the DOM specification from the W3C.

- const declarations map to variables in their respective scope (e.g. xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE); they must not be changed.

- DOMException is currently not supported in xml.dom.minidom. Instead, xml.dom.minidom uses standard Python exceptions such as TypeError and AttributeError.
• NodeList objects are implemented using Python’s built-in list type. These objects provide the interface defined in the DOM specification, but with earlier versions of Python they do not support the official API. They are, however, much more “Pythonic” than the interface defined in the W3C recommendations.

The following interfaces have no implementation in xml.dom.minidom:

• DOMTimeStamp
• DocumentType
• DOMImplementation
• CharacterData
• CDATASection
• Notation
• Entity
• EntityReference
• DocumentFragment

Most of these reflect information in the XML document that is not of general utility to most DOM users.

20.9 xml.dom.pulldom — Support for building partial DOM trees

Source code: Lib/xml/dom/pulldom.py

The xml.dom.pulldom module provides a “pull parser” which can also be asked to produce DOM-accessible fragments of the document where necessary. The basic concept involves pulling “events” from a stream of incoming XML and processing them. In contrast to SAX which also employs an event-driven processing model together with callbacks, the user of a pull parser is responsible for explicitly pulling events from the stream, looping over those events until either processing is finished or an error condition occurs.

Warning: The xml.dom.pulldom module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

Example:

```python
from xml.dom import pulldom

doc = pulldom.parse('sales_items.xml')
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'item':
        if int(node.getAttribute('price')) > 50:
            doc.expandNode(node)
            print(node.toxml())
```

event is a constant and can be one of:

• START_ELEMENT
• END_ELEMENT
• COMMENT
• START_DOCUMENT
• END_DOCUMENT
• CHARACTERS
• PROCESSING_INSTRUCTION
node is an object of type `xml.dom.minidom.Document`, `xml.dom.minidom.Element` or `xml.dom.minidom.Text`.

Since the document is treated as a “flat” stream of events, the document “tree” is implicitly traversed and the desired elements are found regardless of their depth in the tree. In other words, one does not need to consider hierarchical issues such as recursive searching of the document nodes, although if the context of elements were important, one would either need to maintain some context-related state (i.e. remembering where one is in the document at any given point) or to make use of the `DOMEventStream.expandNode()` method and switch to DOM-related processing.

```python
class xml.dom.pulldom.PullDom(documentFactory=None):
    Subclass of `xml.sax.handler.ContentHandler`.

class xml.dom.pulldom.SAX2DOM(documentFactory=None):
    Subclass of `xml.sax.handler.ContentHandler`.

xml.dom.pulldom.parse(stream_or_string, parser=None, bufsize=None)
    Return a `DOMEventStream` from the given input. `stream_or_string` may be either a file name, or a file-like object. `parser`, if given, must be a `XMLReader` object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

If you have XML in a string, you can use the `parseString()` function instead:

```python
xml.dom.pulldom.parseString(string, parser=None)
    Return a `DOMEventStream` that represents the (Unicode) `string`.
```

```python
xml.dom.pulldom.default_bufsize
    Default value for the `bufsize` parameter to `parse()`.

    The value of this variable can be changed before calling `parse()` and the new value will take effect.
```

### 20.9.1 DOMEventStream Objects

```python
class xml.dom.pulldom.DOMEventStream(stream, parser, bufsize)
```

**getEvent()**

Return a tuple containing `event` and the current `node` as `xml.dom.minidom.Document` if `event` equals `START_DOCUMENT`, `xml.dom.minidom.Element` if `event` equals `START_ELEMENT` or `END_ELEMENT` or `xml.dom.minidom.Text` if `event` equals `CHARACTERS`. The current `node` does not contain informations about its children, unless `expandNode()` is called.

```python
expandNode(node)
    Expands all children of `node` into `node`. Example:

    ```python
    xml = '<html><title>Foo</title> <p>Some text <div>and more</div></p> </html>'
    doc = pulldom.parseString(xml)
    for event, node in doc:
        if event == pulldom.START_ELEMENT and node.tagName == 'p':
            # Following statement only prints '<p/>'
            print(node.toxml())
        doc.expandNode(node)
        # Following statement prints node with all its children '<p>Some text <div>and more</div></p>'
        print(node.toxml())
```

**reset()**
20.10 xml.sax — Support for SAX2 parsers

The xml.sax package provides a number of modules which implement the Simple API for XML (SAX) interface for Python. The package itself provides the SAX exceptions and the convenience functions which will be most used by users of the SAX API.

**Warning:** The xml.sax module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

The convenience functions are:

```python
xml.sax.make_parser(parser_list=[])
Create and return a SAX XMLReader object. The first parser found will be used. If parser_list is provided, it must be a sequence of strings which name modules that have a function named create_parser(). Modules listed in parser_list will be used before modules in the default list of parsers.
```

```python
xml.sax.parse(filename_or_stream, handler, error_handler=handler.ErrorHandler())
Create a SAX parser and use it to parse a document. The document, passed in as filename_or_stream, can be a filename or a file object. The handler parameter needs to be a SAX ContentHandler instance. If error_handler is given, it must be a SAX ErrorHandler instance; if omitted, SAXParseException will be raised on all errors. There is no return value; all work must be done by the handler passed in.
```

```python
xml.sax.parseString(string, handler, error_handler=handler.ErrorHandler())
Similar to parse(), but parses from a buffer string received as a parameter.
```

A typical SAX application uses three kinds of objects: readers, handlers and input sources. “Reader” in this context is another term for parser, i.e. some piece of code that reads the bytes or characters from the input source, and produces a sequence of events. The events then get distributed to the handler objects, i.e. the reader invokes a method on the handler. A SAX application must therefore obtain a reader object, create or open the input sources, create the handlers, and connect these objects all together. As the final step of preparation, the reader is called to parse the input. During parsing, methods on the handler objects are called based on structural and syntactic events from the input data.

For these objects, only the interfaces are relevant; they are normally not instantiated by the application itself. Since Python does not have an explicit notion of interface, they are formally introduced as classes, but applications may use implementations which do not inherit from the provided classes. The `InputSource`, `Locator`, `Attributes`, `AttributesNS`, and `XMLReader` interfaces are defined in the module `xml.sax.xmlreader`. The handler interfaces are defined in `xml.sax.handler`. For convenience, `InputSource` (which is often instantiated directly) and the handler classes are also available from `xml.sax`. These interfaces are described below.

In addition to these classes, xml.sax provides the following exception classes.

```python
exception xml.sax.SAXException(msg, exception=None)
Encapsulate an XML error or warning. This class can contain basic error or warning information from either the XML parser or the application: it can be subclassed to provide additional functionality or to add localization. Note that although the handlers defined in the ErrorHandler interface receive instances of this exception, it is not required to actually raise the exception — it is also useful as a container for information.

When instantiated, msg should be a human-readable description of the error. The optional exception parameter, if given, should be `None` or an exception that was caught by the parsing code and is being passed along as information.

This is the base class for the other SAX exception classes.
```

```python
exception xml.sax.SAXParseException(msg, exception, locator)
Subclass of SAXException raised on parse errors. Instances of this class are passed to the methods of the SAX ErrorHandler interface to provide information about the parse error. This class supports the SAX Locator interface as well as the SAXException interface.
```
exception `xml.sax.SAXNotRecognizedException` *(msg, exception=None)*

Subclass of `SAXException` raised when a SAX `XMLReader` is confronted with an unrecognized feature or property. SAX applications and extensions may use this class for similar purposes.

exception `xml.sax.SAXNotSupportedException` *(msg, exception=None)*

Subclass of `SAXException` raised when a SAX `XMLReader` is asked to enable a feature that is not supported, or to set a property to a value that the implementation does not support. SAX applications and extensions may use this class for similar purposes.

See Also:

SAX: The Simple API for XML. This site is the focal point for the definition of the SAX API. It provides a Java implementation and online documentation. Links to implementations and historical information are also available.

Module `xml.sax.handler` Definitions of the interfaces for application-provided objects.

Module `xml.sax.saxutils` Convenience functions for use in SAX applications.

Module `xml.sax.xmlreader` Definitions of the interfaces for parser-provided objects.

### 20.10.1 SAXException Objects

The `SAXException` exception class supports the following methods:

- `SAXException.getMessage()`
  - Return a human-readable message describing the error condition.

- `SAXException.getException()`
  - Return an encapsulated exception object, or `None`.

### 20.11 `xml.sax.handler` — Base classes for SAX handlers

The SAX API defines four kinds of handlers: content handlers, DTD handlers, error handlers, and entity resolvers. Applications normally only need to implement those interfaces whose events they are interested in; they can implement the interfaces in a single object or in multiple objects. Handler implementations should inherit from the base classes provided in the module `xml.sax.handler`, so that all methods get default implementations.

**class** `xml.sax.handler.ContentHandler`

This is the main callback interface in SAX, and the one most important to applications. The order of events in this interface mirrors the order of the information in the document.

**class** `xml.sax.handler.DTDHandler`

Handle DTD events.

This interface specifies only those DTD events required for basic parsing (unparsed entities and attributes).

**class** `xml.sax.handler.EntityResolver`

Basic interface for resolving entities. If you create an object implementing this interface, then register the object with your Parser, the parser will call the method in your object to resolve all external entities.

**class** `xml.sax.handler.ErrorHandler`

Interface used by the parser to present error and warning messages to the application. The methods of this object control whether errors are immediately converted to exceptions or are handled in some other way.

In addition to these classes, `xml.sax.handler` provides symbolic constants for the feature and property names.

`xml.sax.handler.feature_namespaces`

- **value**: "http://xml.org/sax/features/namespaces"
- **true**: Perform Namespace processing.
- **false**: Optionally do not perform Namespace processing (implies namespace-prefixes; default).
xml.sax.handler.feature_namespace_prefixes

value: "http://xml.org/sax/features/namespace-prefixes"
true: Report the original prefixed names and attributes used for Namespace declarations.
false: Do not report attributes used for Namespace declarations, and optionally do not report original prefixed names (default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.feature_string_interning

value: "http://xml.org/sax/features/string-interning"
true: All element names, prefixes, attribute names, Namespace URIs, and local names are interned using the built-in intern function.
false: Names are not necessarily interned, although they may be (default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.feature_validation

value: "http://xml.org/sax/features/validation"
true: Report all validation errors (implies external-general-entities and external-parameter-entities).
false: Do not report validation errors.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.feature_external_ges

value: "http://xml.org/sax/features/external-general-entities"
true: Include all external general (text) entities.
false: Do not include external general entities.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.feature_external_pes

value: "http://xml.org/sax/features/external-parameter-entities"
true: Include all external parameter entities, including the external DTD subset.
false: Do not include any external parameter entities, even the external DTD subset.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.all_features
List of all features.

xml.sax.handler.property_lexical_handler

value: "http://xml.org/sax/properties/lexical-handler"
data type: xml.sax.sax2lib.LexicalHandler (not supported in Python 2)
description: An optional extension handler for lexical events like comments.
access: read/write

xml.sax.handler.property_declaration_handler

value: "http://xml.org/sax/properties/declaration-handler"
data type: xml.sax.sax2lib.DeclHandler (not supported in Python 2)
description: An optional extension handler for DTD-related events other than notations and unparsed entities.
access: read/write

```
xml.sax.handler.property_dom_node
```

- **value**: "http://xml.org/sax/properties/dom-node"
- **data type**: org.w3c.dom.Node (not supported in Python 2)
- **description**: When parsing, the current DOM node being visited if this is a DOM iterator; when not parsing, the root DOM node for iteration.
- **access**: (parsing) read-only; (not parsing) read/write

```
xml.sax.handler.property_xml_string
```

- **value**: "http://xml.org/sax/properties/xml-string"
- **data type**: String
- **description**: The literal string of characters that was the source for the current event.
- **access**: read-only

```
xml.sax.handler.all_properties
```

List of all known property names.

### 20.11.1 ContentHandler Objects

Users are expected to subclass `ContentHandler` to support their application. The following methods are called by the parser on the appropriate events in the input document:

- **ContentHandler.setDocumentLocator** *(locator)*
  Called by the parser to give the application a locator for locating the origin of document events.

  SAX parsers are strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the DocumentHandler interface.

  The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application’s business rules). The information returned by the locator is probably not sufficient for use with a search engine.

  Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.

- **ContentHandler.startDocument** *
  Receive notification of the beginning of a document.

  The SAX parser will invoke this method only once, before any other methods in this interface or in DTD-Handler (except for `setDocumentLocator()`).

- **ContentHandler.endDocument** *
  Receive notification of the end of a document.

  The SAX parser will invoke this method only once, and it will be the last method invoked during the parse. The parser shall not invoke this method until it has either abandoned parsing (because of an unrecoverable error) or reached the end of input.

- **ContentHandler.startPrefixMapping** *(prefix, uri)*
  Begin the scope of a prefix-URI Namespace mapping.

  The information from this event is not necessary for normal Namespace processing: the SAX XML reader will automatically replace prefixes for element and attribute names when the `feature_namespaces` feature is enabled (the default).

  There are cases, however, when applications need to use prefixes in character data or in attribute values, where they cannot safely be expanded automatically; the `startPrefixMapping()` and
endPrefixMapping() events supply the information to the application to expand prefixes in those contexts itself, if necessary.

Note that startPrefixMapping() and endPrefixMapping() events are not guaranteed to be properly nested relative to each other: all startPrefixMapping() events will occur before the corresponding startElement() event, and all endPrefixMapping() events will occur after the corresponding endElement() event, but their order is not guaranteed.

ContentHandler.endPrefixMapping(prefix)
End the scope of a prefix-URI mapping.

See startPrefixMapping() for details. This event will always occur after the corresponding endElement() event, but the order of endPrefixMapping() events is not otherwise guaranteed.

ContentHandler.startElement(name, attrs)
Signals the start of an element in non-namespace mode.

The name parameter contains the raw XML 1.0 name of the element type as a string and the attrs parameter holds an object of the Attributes interface (see The Attributes Interface) containing the attributes of the element. The object passed as attrs may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the copy() method of the attrs object.

ContentHandler.endElement(name)
Signals the end of an element in non-namespace mode.

The name parameter contains the name of the element type, just as with the startElement() event.

ContentHandler.startElementNS(name, qname, attrs)
Signals the start of an element in namespace mode.

The name parameter contains the name of the element type as a (uri, localname) tuple, the qname parameter contains the raw XML 1.0 name used in the source document, and the attrs parameter holds an instance of the AttributesNS interface (see The AttributesNS Interface) containing the attributes of the element. If no namespace is associated with the element, the uri component of name will be None. The object passed as attrs may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the copy() method of the attrs object.

Parsers may set the qname parameter to None, unless the feature_namespace_prefixes feature is activated.

ContentHandler.endElementNS(name, qname)
Signals the end of an element in namespace mode.

The name parameter contains the name of the element type, just as with the startElementNS() method, likewise the qname parameter.

ContentHandler.characters(content)
Receive notification of character data.

The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity so that the Locator provides useful information. content may be a string or bytes instance; the expat reader module always produces strings.

Note: The earlier SAX 1 interface provided by the Python XML Special Interest Group used a more Java-like interface for this method. Since most parsers used from Python did not take advantage of the older interface, the simpler signature was chosen to replace it. To convert old code to the new interface, use content instead of slicing content with the old offset and length parameters.

ContentHandler.ignorableWhitespace(whitespace)
Receive notification of ignorable whitespace in element content.
Validating Parsers must use this method to report each chunk of ignorable whitespace (see the W3C XML 1.0 recommendation, section 2.10): non-validating parsers may also use this method if they are capable of parsing and using content models.

SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the Locator provides useful information.

```python
ContentHandler.processingInstruction(target, data)
Receive notification of a processing instruction.
```

The Parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.

A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.

```python
ContentHandler.skippedEntity(name)
Receive notification of a skipped entity.
```

The Parser will invoke this method once for each entity skipped. Non-validating processors may skip entities if they have not seen the declarations (because, for example, the entity was declared in an external DTD subset). All processors may skip external entities, depending on the values of the `feature_external_ges` and the `feature_external_pes` properties.

### 20.11.2 DTDHandler Objects

```python
DTDHandler.notationDecl(name, publicId, systemId)
Handle a notation declaration event.
```

```python
DTDHandler.unparsedEntityDecl(name, publicId, systemId, ndata)
Handle an unparsed entity declaration event.
```

### 20.11.3 EntityResolver Objects

```python
EntityResolver.resolveEntity(publicId, systemId)
Resolve the system identifier of an entity and return either the system identifier to read from as a string, or an InputSource to read from. The default implementation returns `systemId`.
```

### 20.11.4 ErrorHandler Objects

Objects with this interface are used to receive error and warning information from the `XMLReader`. If you create an object that implements this interface, then register the object with your `XMLReader`, the parser will call the methods in your object to report all warnings and errors. There are three levels of errors available: warnings, (possibly) recoverable errors, and unrecoverable errors. All methods take a `SAXParseException` as the only parameter. Errors and warnings may be converted to an exception by raising the passed-in exception object.

```python
ErrorHandler.error(exception)
Called when the parser encounters a recoverable error. If this method does not raise an exception, parsing may continue, but further document information should not be expected by the application. Allowing the parser to continue may allow additional errors to be discovered in the input document.
```

```python
ErrorHandler.fatalError(exception)
Called when the parser encounters an error it cannot recover from; parsing is expected to terminate when this method returns.
```
ErrorHandler.warning(exception)

Called when the parser presents minor warning information to the application. Parsing is expected to con-
tinue when this method returns, and document information will continue to be passed to the application.
Raising an exception in this method will cause parsing to end.

20.12 xml.sax.saxutils — SAX Utilities

The module xml.sax.saxutils contains a number of classes and functions that are commonly useful when
creating SAX applications, either in direct use, or as base classes.

xml.sax.saxutils.escape(data, entities={})

Escape ‘&’, ‘<’, and ‘>’ in a string of data.

You can escape other strings of data by passing a dictionary as the optional entities parameter. The keys and
values must all be strings; each key will be replaced with its corresponding value. The characters ‘&’, ‘<’
and ‘>’ are always escaped, even if entities is provided.

xml.sax.saxutils.unescape(data, entities={})

Unescape ‘&amp;’,’&lt;’, and ‘&gt;’ in a string of data.

You can unescape other strings of data by passing a dictionary as the optional entities parameter. The keys
and values must all be strings; each key will be replaced with its corresponding value. ‘&amp;’,’&lt;’,
and ‘&gt;’ are always unescaped, even if entities is provided.

xml.sax.saxutils.quoteattr(data, entities={})

Similar to escape(), but also prepares data to be used as an attribute value. The return value is a quoted
version of data with any additional required replacements. quoteattr() will select a quote character
based on the content of data, attempting to avoid encoding any quote characters in the string. If both single-
and double-quote characters are already in data, the double-quote characters will be encoded and data will
be wrapped in double-quotes. The resulting string can be used directly as an attribute value:

```python
>>> print("<element attr=%s>" % quoteattr("ab ' cd \" ef"))
<element attr="ab ' cd &quot; ef">
```

This function is useful when generating attribute values for HTML or any SGML using the reference con-
crete syntax.

class xml.sax.saxutils.XMLGenerator(out=None, encoding='iso-8859-1', short_empty_elements=False)

This class implements the ContentHandler interface by writing SAX events back into an XML docu-
ment. In other words, using an XMLGenerator as the content handler will reproduce the original docu-
ment being parsed. out should be a file-like object which will default to sys.stdout. encoding is the encoding of
the output stream which defaults to 'iso-8859-1'. short_empty_elements controls the formatting of
elements that contain no content: if False (the default) they are emitted as a pair of start/end tags, if set to
True they are emitted as a single self-closed tag. New in version 3.2: The short_empty_elements parameter.

class xml.sax.saxutils.XMLFilterBase(base)

This class is designed to sit between an XMLReader and the client application’s event handlers. By default,
it does nothing but pass requests up to the reader and events on to the handlers unmodified, but subclasses
can override specific methods to modify the event stream or the configuration requests as they pass through.

xml.sax.saxutils.prepare_input_source(source, base='')

This function takes an input source and an optional base URL and returns a fully resolved InputSource
object ready for reading. The input source can be given as a string, a file-like object, or an InputSource
object; parsers will use this function to implement the polymorphic source argument to their parse() method.
20.13 xml.sax.xmlreader — Interface for XML parsers

SAX parsers implement the XMLReader interface. They are implemented in a Python module, which must provide a function create_parser(). This function is invoked by xml.sax.make_parser() with no arguments to create a new parser object.

**class xml.sax.xmlreader.XMLReader**

Base class which can be inherited by SAX parsers.

**class xml.sax.xmlreader.IncrementalParser**

In some cases, it is desirable not to parse an input source at once, but to feed chunks of the document as they get available. Note that the reader will normally not read the entire file, but read it in chunks as well; still parse() won’t return until the entire document is processed. So these interfaces should be used if the blocking behaviour of parse() is not desirable.

When the parser is instantiated it is ready to begin accepting data from the feed method immediately. After parsing has been finished with a call to close the reset method must be called to make the parser ready to accept new data, either from feed or using the parse method.

Note that these methods must *not* be called during parsing, that is, after parse has been called and before it returns.

By default, the class also implements the parse method of the XMLReader interface using the feed, close and reset methods of the IncrementalParser interface as a convenience to SAX 2.0 driver writers.

**class xml.sax.xmlreader.Locator**

Interface for associating a SAX event with a document location. A locator object will return valid results only during calls to DocumentHandler methods; at any other time, the results are unpredictable. If information is not available, methods may return `None`.

**class xml.sax.xmlreader.InputSource**(system_id=None)

Encapsulation of the information needed by the XMLReader to read entities.

This class may include information about the public identifier, system identifier, byte stream (possibly with character encoding information) and/or the character stream of an entity.

Applications will create objects of this class for use in the XMLReader.parse() method and for returning from EntityResolver.resolveEntity.

An InputSource belongs to the application, the XMLReader is not allowed to modify InputSource objects passed to it from the application, although it may make copies and modify those.

**class xml.sax.xmlreader.AttributesImpl**(attrs)

This is an implementation of the Attributes interface (see section The Attributes Interface). This is a dictionary-like object which represents the element attributes in a startElement() call. In addition to the most useful dictionary operations, it supports a number of other methods as described by the interface. Objects of this class should be instantiated by readers; *attrs* must be a dictionary-like object containing a mapping from attribute names to attribute values.

**class xml.sax.xmlreader.AttributesNSImpl**(attrs, qnames)

Namespace-aware variant of AttributesImpl, which will be passed to startElementNS(). It is derived from AttributesImpl, but understands attribute names as two-tuples of namespaceURI and localname. In addition, it provides a number of methods expecting qualified names as they appear in the original document. This class implements the AttributesNS interface (see section The AttributesNS Interface).

20.13.1 XMLReader Objects

The XMLReader interface supports the following methods:

XMLReader.parse(source)

Process an input source, producing SAX events. The source object can be a system identifier (a string identifying the input source – typically a file name or an URL), a file-like object, or an InputSource
object. When `parse()` returns, the input is completely processed, and the parser object can be discarded or reset. As a limitation, the current implementation only accepts byte streams; processing of character streams is for further study.

```python
XMLReader.getContentHandler()
Return the current ContentHandler.
```

```python
XMLReader.setContentHandler(handler)
Set the current ContentHandler. If no ContentHandler is set, content events will be discarded.
```

```python
XMLReader.getDTDHandler()
Return the current DTDHandler.
```

```python
XMLReader.setDTDHandler(handler)
Set the current DTDHandler. If no DTDHandler is set, DTD events will be discarded.
```

```python
XMLReader.getEntityResolver()
Return the current EntityResolver.
```

```python
XMLReader.setEntityResolver(handler)
Set the current EntityResolver. If no EntityResolver is set, attempts to resolve an external entity will result in opening the system identifier for the entity, and fail if it is not available.
```

```python
XMLReader.getErrorHandler()
Return the current ErrorHandler.
```

```python
XMLReader.setErrorHandler(handler)
Set the current error handler. If no ErrorHandler is set, errors will be raised as exceptions, and warnings will be printed.
```

```python
XMLReader.setLocale(locale)
Allow an application to set the locale for errors and warnings.
SAX parsers are not required to provide localization for errors and warnings; if they cannot support the requested locale, however, they must raise a SAX exception. Applications may request a locale change in the middle of a parse.
```

```python
XMLReader.getFeature(featurename)
Return the current setting for feature `featurename`. If the feature is not recognized, `SAXNotRecognizedException` is raised. The well-known featurenames are listed in the module `xml.sax.handler`.
```

```python
XMLReader.setFeature(featurename, value)
Set the `featurename` to `value`. If the feature is not recognized, `SAXNotRecognizedException` is raised. If the feature or its setting is not supported by the parser, `SAXNotSupportedException` is raised.
```

```python
XMLReader.getProperty(propertyname)
Return the current setting for property `propertyname`. If the property is not recognized, a `SAXNotRecognizedException` is raised. The well-known propertynames are listed in the module `xml.sax.handler`.
```

```python
XMLReader.setProperty(propertyname, value)
Set the `propertyname` to `value`. If the property is not recognized, `SAXNotRecognizedException` is raised. If the property or its setting is not supported by the parser, `SAXNotSupportedException` is raised.
```

### 20.13.2 IncrementalParser Objects

Instances of `IncrementalParser` offer the following additional methods:

```python
IncrementalParser.feed(data)
Process a chunk of `data`.
```

```python
IncrementalParser.close()
Assume the end of the document. That will check well-formedness conditions that can be checked only at the end, invoke handlers, and may clean up resources allocated during parsing.
```
IncrementalParser.\texttt{reset}()

This method is called after close has been called to reset the parser so that it is ready to parse new documents. The results of calling parse or feed after close without calling \texttt{reset} are undefined.

### 20.13.3 Locator Objects

Instances of \texttt{Locator} provide these methods:

\begin{verbatim}
Locator.\texttt{getColumnNumber}()
    Return the column number where the current event ends.

Locator.\texttt{getLineNumber}()
    Return the line number where the current event ends.

Locator.\texttt{getPublicId}()
    Return the public identifier for the current event.

Locator.\texttt{getSystemId}()
    Return the system identifier for the current event.
\end{verbatim}

### 20.13.4 InputSource Objects

\begin{verbatim}
InputSource.\texttt{setPublicId}(id)
    Sets the public identifier of this \texttt{InputSource}.

InputSource.\texttt{getPublicId}()
    Returns the public identifier of this \texttt{InputSource}.

InputSource.\texttt{setSystemId}(id)
    Sets the system identifier of this \texttt{InputSource}.

InputSource.\texttt{getSystemId}()
    Returns the system identifier of this \texttt{InputSource}.

InputSource.\texttt{setEncoding}(encoding)
    Sets the character encoding of this \texttt{InputSource}.
    The encoding must be a string acceptable for an XML encoding declaration (see section 4.3.3 of the XML recommendation).
    The encoding attribute of the \texttt{InputSource} is ignored if the \texttt{InputSource} also contains a character stream.

InputSource.\texttt{getEncoding}()
    Get the character encoding of this InputSource.

InputSource.\texttt{setByteStream}(bytefile)
    Set the byte stream (a Python file-like object which does not perform byte-to-character conversion) for this input source.
    The SAX parser will ignore this if there is also a character stream specified, but it will use a byte stream in preference to opening a URI connection itself.
    If the application knows the character encoding of the byte stream, it should set it with the setEncoding method.

InputSource.\texttt{getByteStream}()
    Get the byte stream for this input source.
    The getEncoding method will return the character encoding for this byte stream, or None if unknown.

InputSource.\texttt{setCharacterStream}(charfile)
    Set the character stream for this input source. (The stream must be a Python 1.6 Unicode-wrapped file-like that performs conversion to strings.)
\end{verbatim}
If there is a character stream specified, the SAX parser will ignore any byte stream and will not attempt to open a URI connection to the system identifier.

```python
InputSource.getCharacterStream()
```
Get the character stream for this input source.

### 20.13.5 The Attributes Interface

Attributes objects implement a portion of the mapping protocol, including the methods `copy()`, `get()`, `__contains__()`, `items()`, `keys()`, and `values()`. The following methods are also provided:

```python
Attributes.getLength()
```
Return the number of attributes.

```python
Attributes.getNames()
```
Return the names of the attributes.

```python
Attributes.getType(name)
```
Return the type of the attribute `name`, which is normally `'CDATA'`.

```python
Attributes.getValue(name)
```
Return the value of attribute `name`.

### 20.13.6 The AttributesNS Interface

This interface is a subtype of the Attributes interface (see section The Attributes Interface). All methods supported by that interface are also available on AttributesNS objects.

The following methods are also available:

```python
AttributesNS.getValueByQName(name)
```
Return the value for a qualified name.

```python
AttributesNS.getNameByQName(name)
```
Return the `(namespace, localname)` pair for a qualified name.

```python
AttributesNS.getQNameByName(name)
```
Return the qualified name for a `(namespace, localname)` pair.

```python
AttributesNS.getQNames()
```
Return the qualified names of all attributes.

### 20.14 xml.parsers.expat — Fast XML parsing using Expat

**Warning:** The `pyexpat` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

The `xml.parsers.expat` module is a Python interface to the Expat non-validating XML parser. The module provides a single extension type, `xmlparser`, that represents the current state of an XML parser. After an `xmlparser` object has been created, various attributes of the object can be set to handler functions. When an XML document is then fed to the parser, the handler functions are called for the character data and markup in the XML document.

This module uses the `pyexpat` module to provide access to the Expat parser. Direct use of the `pyexpat` module is deprecated.

This module provides one exception and one type object:

**exception** `xml.parsers.expat.ExpatError`

The exception raised when Expat reports an error. See section ExpatError Exceptions for more information on interpreting Expat errors.
**exception** `xml.parsers.expat.error`

Alias for `ExpatError`.

`xml.parsers.expat.XMLParserType`

The type of the return values from the `ParserCreate()` function.

The `xml.parsers.expat` module contains two functions:

`xml.parsers.expat.ErrorString(errno)`

Returns an explanatory string for a given error number `errno`.

`xml.parsers.expat.ParserCreate(encoding=None, namespace_separator=None)`

Creates and returns a new XML parser object. `encoding`, if specified, must be a string naming the encoding used by the XML data. Expat doesn’t support as many encodings as Python does, and its repertoire of encodings can’t be extended; it supports UTF-8, UTF-16, ISO-8859-1 (Latin1), and ASCII. If `encoding` is given it will override the implicit or explicit encoding of the document.

Expat can optionally do XML namespace processing for you, enabled by providing a value for `namespace_separator`. The value must be a one-character string; a `ValueError` will be raised if the string has an illegal length (None is considered the same as omission). When namespace processing is enabled, element type names and attribute names that belong to a namespace will be expanded. The element name passed to the element handlers `StartElementHandler` and `EndElementHandler` will be the concatenation of the namespace URI, the namespace separator character, and the local part of the name. If the namespace separator is a zero byte (chr(0)) then the namespace URI and the local part will be concatenated without any separator.

For example, if `namespace_separator` is set to a space character (‘ ‘) and the following document is parsed:

```xml
<?xml version="1.0"?>
<root xmlns:py="http://www.python.org/ns/">
    <py:elem1 />
    <elem2 xmlns="" />
</root>
```

`StartElementHandler` will receive the following strings for each element:

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

See Also:

**The Expat XML Parser** Home page of the Expat project.

### 20.14.1 XMLParser Objects

`xmlparser` objects have the following methods:

`xmlparser.Parse(data[, isfinal])`

Parses the contents of the string `data`, calling the appropriate handler functions to process the parsed data. `isfinal` must be true on the final call to this method. `data` can be the empty string at any time.

`xmlparser.ParseFile(file)`

Parse XML data reading from the object `file`. `file` only needs to provide the `read(nbytes)` method, returning the empty string when there’s no more data.

---

5 The encoding string included in XML output should conform to the appropriate standards. For example, “UTF-8” is valid, but “UTF8” is not. See [http://www.w3.org/TR/2006/REC-xml11-20060816/#NT- EncodingDecl](http://www.w3.org/TR/2006/REC-xml11-20060816/#NT- EncodingDecl) and [http://www.iana.org/assignments/character-sets](http://www.iana.org/assignments/character-sets).
xmlparser.SetBase (base)
   Sets the base to be used for resolving relative URIs in system identifiers in declarations. Resolving relative identifiers is left to the application: this value will be passed through as the base argument to the ExternalEntityRefHandler(), NotationDeclHandler(), and UnparsedEntityDeclHandler() functions.

xmlparser.GetBase ()
   Returns a string containing the base set by a previous call to SetBase(), or None if SetBase() hasn’t been called.

xmlparser.GetInputContext ()
   Returns the input data that generated the current event as a string. The data is in the encoding of the entity which contains the text. When called while an event handler is not active, the return value is None.

xmlparser.ExternalEntityParserCreate (context[, encoding])
   Create a “child” parser which can be used to parse an external parsed entity referred to by content parsed by the parent parser. The context parameter should be the string passed to the ExternalEntityRefHandler() handler function, described below. The child parser is created with the ordered_attributes and specified_attributes set to the values of this parser.

xmlparser.SetParamEntityParsing (flag)
   Control parsing of parameter entities (including the external DTD subset). Possible flag values are XML_PARAM_ENTITY_PARSING_NEVER, XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE and XML_PARAM_ENTITY_PARSING_ALWAYS. Return true if setting the flag was successful.

xmlparser.UseForeignDTD ([flag])
   Calling this with a true value for flag (the default) will cause Expat to call the ExternalEntityRefHandler with None for all arguments to allow an alternate DTD to be loaded. If the document does not contain a document type declaration, the ExternalEntityRefHandler will still be called, but the StartDoctypeDeclHandler and EndDoctypeDeclHandler will not be called.

   Passing a false value for flag will cancel a previous call that passed a true value, but otherwise has no effect.

   This method can only be called before the Parse() or ParseFile() methods are called; calling it after either of those have been called causes ExpatError to be raised with the code attribute set to errors.codes[errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING].

xmlparser objects have the following attributes:

xmlparser.buffer_size
   The size of the buffer used when buffer_text is true. A new buffer size can be set by assigning a new integer value to this attribute. When the size is changed, the buffer will be flushed.

xmlparser.buffer_text
   Setting this to true causes the xmlparser object to buffer textual content returned by Expat to avoid multiple calls to the CharacterDataHandler() callback whenever possible. This can improve performance substantially since Expat normally breaks character data into chunks at every line ending. This attribute is false by default, and may be changed at any time.

xmlparser.buffer_used
   If buffer_text is enabled, the number of bytes stored in the buffer. These bytes represent UTF-8 encoded text. This attribute has no meaningful interpretation when buffer_text is false.

xmlparser.ordered_attributes
   Setting this attribute to a non-zero integer causes the attributes to be reported as a list rather than a dictionary. The attributes are presented in the order found in the document text. For each attribute, two list entries are presented: the attribute name and the attribute value. (Older versions of this module also used this format.) By default, this attribute is false; it may be changed at any time.

xmlparser.specified_attributes
   If set to a non-zero integer, the parser will report only those attributes which were specified in the document instance and not those which were derived from attribute declarations. Applications which set this need to be especially careful to use what additional information is available from the declarations as needed to
comply with the standards for the behavior of XML processors. By default, this attribute is false; it may be changed at any time.

The following attributes contain values relating to the most recent error encountered by an xmlparser object, and will only have correct values once a call to Parse() or ParseFile() has raised a xml.parsers.expat.ExpatError exception.

xmlparser.ErrorByteIndex
Byte index at which an error occurred.

xmlparser.ErrorCode
Numeric code specifying the problem. This value can be passed to the ErrorString() function, or compared to one of the constants defined in the errors object.

xmlparser.ErrorLineNumber
Line number at which an error occurred.

xmlparser.ErrorColumnNumber
Column number at which an error occurred.

The following attributes contain values relating to the current parse location in an xmlparser object. During a callback reporting a parse event they indicate the location of the first of the sequence of characters that generated the event. When called outside of a callback, the position indicated will be just past the last parse event (regardless of whether there was an associated callback).

xmlparser.CurrentByteIndex
Current byte index in the parser input.

xmlparser.CurrentColumnNumber
Current column number in the parser input.

xmlparser.CurrentLineNumber
Current line number in the parser input.

Here is the list of handlers that can be set. To set a handler on an xmlparser object o, use o.handlername = func. handlername must be taken from the following list, and func must be a callable object accepting the correct number of arguments. The arguments are all strings, unless otherwise stated.

xmlparser.XmlDeclHandler (version, encoding, standalone)
Called when the XML declaration is parsed. The XML declaration is the (optional) declaration of the applicable version of the XML recommendation, the encoding of the document text, and an optional “standalone” declaration. version and encoding will be strings, and standalone will be 1 if the document is declared standalone, 0 if it is declared not to be standalone, or -1 if the standalone clause was omitted. This is only available with Expat version 1.95.0 or newer.

xmlparser.StartDoctypeDeclHandler (doctypeName, systemId, publicId, has_internal_subset)
Called when Expat begins parsing the document type declaration (<!DOCTYPE ...). The doctypeName is provided exactly as presented. The systemId and publicId parameters give the system and public identifiers if specified, or None if omitted. has_internal_subset will be true if the document contains and internal document declaration subset. This requires Expat version 1.2 or newer.

xmlparser.EndDoctypeDeclHandler ()
Called when Expat is done parsing the document type declaration. This requires Expat version 1.2 or newer.

xmlparser.ElementDeclHandler (name, model)
Called once for each element type declaration. name is the name of the element type, and model is a representation of the content model.

xmlparser.AttlistDeclHandler (elname, attname, type, default, required)
Called for each declared attribute for an element type. If an attribute list declaration declares three attributes, this handler is called three times, once for each attribute. elname is the name of the element type which the declaration applies and attname is the name of the attribute declared. The attribute type is a string passed as type; the possible values are ‘CDATA’, ‘ID’, ‘IDREF’, ... default gives the default value for the attribute used when the attribute is not specified by the document instance, or None if there is no default value (#IMPLIED values). If the attribute is required to be given in the document instance, required will be true. This requires Expat version 1.95.0 or newer.
xmlparser.StartElementHandler(name, attributes)
Called for the start of every element. name is a string containing the element name, and attributes is the
element attributes. If ordered_attributes is true, this is a list (see ordered_attributes for a
full description). Otherwise it’s a dictionary mapping names to values.

xmlparser.EndElementHandler(name)
Called for the end of every element.

xmlparser.ProcessingInstructionHandler(target, data)
Called for every processing instruction.

xmlparser.CharacterDataHandler(data)
Called for character data. This will be called for normal character data, CDATA marked con-
tent, and ignorable whitespace. Applications which must distinguish these cases can use the
StartCdataSectionHandler, EndCdataSectionHandler, and ElementDeclHandler
callbacks to collect the required information.

xmlparser.UnparsedEntityDeclHandler(entityName, base, systemId, publicId, notation-
Name)
Called for unparsed (NDATA) entity declarations. This is only present for version 1.2 of the Expat library;
for more recent versions, use EntityDeclHandler instead. (The underlying function in the Expat
library has been declared obsolete.)

xmlparser.EntityDeclHandler(entityName, is_parameter_entity, value, base, systemId, publicId,
notationName)
Called for all entity declarations. For parameter and internal entities, value will be a string giving the
declared contents of the entity; this will be None for external entities. The notationName parameter will
be None for parsed entities, and the name of the notation for unparsed entities. is_parameter_entity
will be true if the entity is a parameter entity or false for general entities (most applications only need to be
concerned with general entities). This is only available starting with version 1.95.0 of the Expat library.

xmlparser.NotationDeclHandler(notationName, base, systemId, publicId)
Called for notation declarations. notationName, base, and systemId, and publicId are strings if given. If the
public identifier is omitted, publicId will be None.

xmlparser.StartNamespaceDeclHandler(prefix, uri)
Called when an element contains a namespace declaration. Namespace declarations are processed before
the StartElementHandler is called for the element on which declarations are placed.

xmlparser.EndNamespaceDeclHandler(prefix)
Called when the closing tag is reached for an element that contained a namespace declaration. This is
called once for each namespace declaration on the element in the reverse of the order for which the
StartNamespaceDeclHandler was called to indicate the start of each namespace declaration’s scope.
Calls to this handler are made after the corresponding EndElementHandler for the end of the element.

xmlparser.CommentHandler(data)
Called for comments. data is the text of the comment, excluding the leading ‘<!--’ and trailing ‘--’.

xmlparser.StartCdataSectionHandler()
Called at the start of a CDATA section. This and EndCdataSectionHandler are needed to be able to
identify the syntactical start and end for CDATA sections.

xmlparser.EndCdataSectionHandler()
Called at the end of a CDATA section.

xmlparser.DefaultHandler(data)
Called for any characters in the XML document for which no applicable handler has been specified. This
means characters that are part of a construct which could be reported, but for which no handler has been
supplied.

xmlparser.DefaultHandlerExpand(data)
This is the same as the DefaultHandler(), but doesn’t inhibit expansion of internal entities. The entity
reference will not be passed to the default handler.

xmlparser.NotStandaloneHandler()
Called if the XML document hasn’t been declared as being a standalone document. This happens when
there is an external subset or a reference to a parameter entity, but the XML declaration does not set standalone to yes in an XML declaration. If this handler returns 0, then the parser will raise an XML_ERROR_NOT_STANDALONE error. If this handler is not set, no exception is raised by the parser for this condition.

xmlparser.ExternalEntityRefHandler (context, base, systemId, publicId)
Called for references to external entities. base is the current base, as set by a previous call to SetBase(). The public and system identifiers, systemId and publicId, are strings if given; if the public identifier is not given, publicId will be None. The context value is opaque and should only be used as described below.

For external entities to be parsed, this handler must be implemented. It is responsible for creating the sub-parser using ExternalEntityParserCreate(context), initializing it with the appropriate callbacks, and parsing the entity. This handler should return an integer; if it returns 0, the parser will raise an XML_ERROR_EXTERNAL_ENTITY_HANDLING error, otherwise parsing will continue.

If this handler is not provided, external entities are reported by the DefaultHandler callback, if provided.

20.14.2 ExpatError Exceptions

ExpatError exceptions have a number of interesting attributes:

ExpatError.code
Expat’s internal error number for the specific error. The errors.messages dictionary maps these error numbers to Expat’s error messages. For example:

```python
from xml.parsers.expat import ParserCreate, ExpatError, errors

p = ParserCreate()
try:
    p.Parse(some_xml_document)
except ExpatError as err:
    print("Error:", errors.messages[err.code])
```

The errors module also provides error message constants and a dictionary codes mapping these messages back to the error codes, see below.

ExpatError.lineno
Line number on which the error was detected. The first line is numbered 1.

ExpatError.offset
Character offset into the line where the error occurred. The first column is numbered 0.

20.14.3 Example

The following program defines three handlers that just print out their arguments.

```python
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print(’Start element:’, name, attrs)
def end_element(name):
    print(’End element:’, name)
def char_data(data):
    print(’Character data:’, repr(data))

p = xml.parsers.expat.ParserCreate()
p.StartElementHandler = start_element
```

p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""<?xml version="1.0"?>
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""", 1)
The output from this program is:
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent

20.14.4 Content Model Descriptions

Content modules are described using nested tuples. Each tuple contains four values: the type, the quantifier, the name, and a tuple of children. Children are simply additional content module descriptions.

The values of the first two fields are constants defined in the xml.parsers.expat.model module. These constants can be collected in two groups: the model type group and the quantifier group.

The constants in the model type group are:

xml.parsers.expat.model.XML_CTYPE_ANY
   The element named by the model name was declared to have a content model of ANY.

xml.parsers.expat.model.XML_CTYPE_CHOICE
   The named element allows a choice from a number of options; this is used for content models such as (A | B | C).

xml.parsers.expat.model.XML_CTYPE_EMPTY
   Elements which are declared to be EMPTY have this model type.

xml.parsers.expat.model.XML_CTYPE_MIXED

xml.parsers.expat.model.XML_CTYPE_NAME

xml.parsers.expat.model.XML_CTYPE_SEQ
   Models which represent a series of models which follow one after the other are indicated with this model type. This is used for models such as (A, B, C).

The constants in the quantifier group are:

xml.parsers.expat.model.XML_CQUANT_NONE
   No modifier is given, so it can appear exactly once, as for A.

xml.parsers.expat.model.XML_CQUANT_OPT
   The model is optional: it can appear once or not at all, as for A?.

xml.parsers.expat.model.XML_CQUANT_PLUS
   The model must occur one or more times (like A+).

xml.parsers.expat.model.XML_CQUANT_REP
   The model must occur zero or more times, as for A*.
20.14.5 Expat error constants

The following constants are provided in the `xml.parsers.expat.errors` module. These constants are useful in interpreting some of the attributes of the `ExpatError` exception objects raised when an error has occurred. Since for backwards compatibility reasons, the constants’ value is the error message and not the numeric error code, you do this by comparing its code attribute with `errors.codes[errors.XML_ERROR_CONSTANT_NAME]`.

The `errors` module has the following attributes:

- `xml.parsers.expat.errors.codes`: A dictionary mapping numeric error codes to their string descriptions. New in version 3.2.
- `xml.parsers.expat.errors.messages`: A dictionary mapping string descriptions to their error codes. New in version 3.2.
- `xml.parsers.expat.errors.XML_ERROR_ASYNC_ENTITY`: An entity reference in an attribute value referred to an external entity instead of an internal entity.
- `xml.parsers.expat.errors.XML_ERRORATTRIBUTE_EXTERNAL_ENTITY_REF`: An entity reference referred to an external entity instead of an internal entity.
- `xml.parsers.expat.errors.XML_ERROR_BAD_CHAR_REF`: A character reference referred to a character which is illegal in XML (for example, character 0, or `&#0;`).
- `xml.parsers.expat.errors.XML_ERROR_BINARY_ENTITY_REF`: An entity reference referred to an entity which was declared with a notation, so cannot be parsed.
- `xml.parsers.expat.errors.XML_ERROR_DUPLICATE_ATTRIBUTE`: An attribute was used more than once in a start tag.
- `xml.parsers.expat.errors.XML_ERROR_INCORRECT_ENCODING`: Raised when an input byte could not properly be assigned to a character; for example, a NUL byte (value 0) in a UTF-8 input stream.
- `xml.parsers.expat.errors.XML_ERROR_JUNK_AFTER_DOC_ELEMENT`: Something other than whitespace occurred after the document element.
- `xml.parsers.expat.errors.XML_ERROR_MISPLACED_XML_PI`: An XML declaration was found somewhere other than the start of the input data.
- `xml.parsers.expat.errors.XML_ERROR_NO_ELEMENTS`: The document contains no elements (XML requires all documents to contain exactly one top-level element).
- `xml.parsers.expat.errors.XML_ERROR_NO_MEMORY`: Expat was not able to allocate memory internally.
- `xml.parsers.expat.errors.XML_ERROR_PARAM_ENTITY_REF`: A parameter entity reference was found where it was not allowed.
- `xml.parsers.expat.errors.XML_ERROR_PARTIAL_CHAR`: An incomplete character was found in the input.
- `xml.parsers.expat.errors.XML_ERROR_RECURSIVE_ENTITY_REF`: An entity reference contained another reference to the same entity; possibly via a different name, and possibly indirectly.
- `xml.parsers.expat.errors.XML_ERROR_SYNTAX`: Some unspecified syntax error was encountered.
- `xml.parsers.expat.errors.XML_ERROR_TAG_MISMATCH`: An end tag did not match the innermost open start tag.
- `xml.parsers.expat.errors.XML_ERROR_UNCLOSED_TOKEN`: Some token (such as a start tag) was not closed before the end of the stream or the next token was encountered.
xml.parsers.expat.errors.XML_ERROR_UNDEFINED_ENTITY
A reference was made to an entity which was not defined.

xml.parsers.expat.errors.XML_ERROR_UNKNOWN_ENCODING
The document encoding is not supported by Expat.

xml.parsers.expat.errors.XML_ERROR_UNCLOSED_CDATA_SECTION
A CDATA marked section was not closed.

xml.parsers.expat.errors.XML_ERROR_EXTERNAL_ENTITY_HANDLING

xml.parsers.expat.errors.XML_ERROR_NOT_STANDALONE
The parser determined that the document was not “standalone” though it declared itself to be in the XML declaration, and the NotStandaloneHandler was set and returned 0.

xml.parsers.expat.errors.XML_ERROR_UNEXPECTED_STATE

xml.parsers.expat.errors.XML_ERROR_ENTITY_DECLARED_IN_PE
An undeclared prefix was found when namespace processing was enabled.

xml.parsers.expat.errors.XML_ERROR_UNDECLARING_PREFIX
The document attempted to remove the namespace declaration associated with a prefix.

xml.parsers.expat.errors.XML_ERROR_INCOMPLETE_PE
A parameter entity contained incomplete markup.

xml.parsers.expat.errors.XML_ERROR_XML_DECL
The document contained no document element at all.

xml.parsers.expat.errors.XML_ERROR_TEXT_DECL
There was an error parsing a text declaration in an external entity.

xml.parsers.expat.errors.XML_ERROR_PUBLICID
Characters were found in the public id that are not allowed.

xml.parsers.expat.errors.XML_ERROR_SUSPENDED
The requested operation was made on a suspended parser, but isn’t allowed. This includes attempts to provide additional input or to stop the parser.

xml.parsers.expat.errors.XML_ERROR_NOT_SUSPENDED
An attempt to resume the parser was made when the parser had not been suspended.

xml.parsers.expat.errors.XML_ERROR_ABORTED
This should not be reported to Python applications.

xml.parsers.expat.errors.XML_ERROR_FINISHED
The requested operation was made on a parser which was finished parsing input, but isn’t allowed. This includes attempts to provide additional input or to stop the parser.
The modules described in this chapter implement Internet protocols and support for related technology. They are all implemented in Python. Most of these modules require the presence of the system-dependent module `socket`, which is currently supported on most popular platforms. Here is an overview:

### 21.1 `webbrowser` — Convenient Web-browser controller

**Source code:** `Lib/webbrowser.py`

The `webbrowser` module provides a high-level interface to allow displaying Web-based documents to users. Under most circumstances, simply calling the `open()` function from this module will do the right thing.

Under Unix, graphical browsers are preferred under X11, but text-mode browsers will be used if graphical browsers are not available or an X11 display isn’t available. If text-mode browsers are used, the calling process will block until the user exits the browser.

If the environment variable `BROWSER` exists, it is interpreted to override the platform default list of browsers, as a `os.pathsep`-separated list of browsers to try in order. When the value of a list part contains the string `%s`, then it is interpreted as a literal browser command line to be used with the argument URL substituted for `%s`; if the part does not contain `%s`, it is simply interpreted as the name of the browser to launch. 1

For non-Unix platforms, or when a remote browser is available on Unix, the controlling process will not wait for the user to finish with the browser, but allow the remote browser to maintain its own windows on the display. If remote browsers are not available on Unix, the controlling process will launch a new browser and wait.

The script `webbrowser` can be used as a command-line interface for the module. It accepts an URL as the argument. It accepts the following optional parameters: `-n` opens the URL in a new browser window, if possible; `-t` opens the URL in a new browser page (“tab”). The options are, naturally, mutually exclusive. Usage example:

```
python -m webbrowser -t "http://www.python.org"
```

The following exception is defined:

**exception** `webbrowser.Error`

Exception raised when a browser control error occurs.

The following functions are defined:

```
webbrowser.open (url, new=0, autoraise=True)
```

Display `url` using the default browser. If `new` is 0, the `url` is opened in the same browser window if possible. If `new` is 1, a new browser window is opened if possible. If `new` is 2, a new browser page (“tab”) is opened if possible. If `autoraise` is `True`, the window is raised if possible (note that under many window managers this will occur regardless of the setting of this variable).

---

1 Executables named here without a full path will be searched in the directories given in the `PATH` environment variable.
Note that on some platforms, trying to open a filename using this function, may work and start the operating system’s associated program. However, this is neither supported nor portable.

```
webbrowser.open_new(url)
```

Open `url` in a new window of the default browser, if possible, otherwise, open `url` in the only browser window.

```
webbrowser.open_new_tab(url)
```

Open `url` in a new page (“tab”) of the default browser, if possible, otherwise equivalent to `open_new()`.

```
webbrowser.get(using=None)
```

Return a controller object for the browser type `using`. If `using` is `None`, return a controller for a default browser appropriate to the caller’s environment.

```
webbrowser.register(name, constructor, instance=None)
```

Register the browser type `name`. Once a browser type is registered, the `get()` function can return a controller for that browser type. If `instance` is not provided, or is `None`, `constructor` will be called without parameters to create an instance when needed. If `instance` is provided, `constructor` will never be called, and may be `None`.

This entry point is only useful if you plan to either set the `BROWSER` variable or call `get()` with a nonempty argument matching the name of a handler you declare.

A number of browser types are predefined. This table gives the type names that may be passed to the `get()` function and the corresponding instantiations for the controller classes, all defined in this module.

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Class Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>'mozilla'</td>
<td>Mozilla('mozilla')</td>
<td></td>
</tr>
<tr>
<td>'firefox'</td>
<td>Mozilla('mozilla')</td>
<td></td>
</tr>
<tr>
<td>'netscape'</td>
<td>Mozilla('netscape')</td>
<td></td>
</tr>
<tr>
<td>'galeon'</td>
<td>Galeon('galeon')</td>
<td></td>
</tr>
<tr>
<td>'epiphany'</td>
<td>Galeon('epiphany')</td>
<td></td>
</tr>
<tr>
<td>'skipstone'</td>
<td>BackgroundBrowser('skipstone')</td>
<td></td>
</tr>
<tr>
<td>'kfmclient'</td>
<td>Konqueror()</td>
<td>(1)</td>
</tr>
<tr>
<td>'konqueror'</td>
<td>Konqueror()</td>
<td>(1)</td>
</tr>
<tr>
<td>'kfm'</td>
<td>Konqueror()</td>
<td>(1)</td>
</tr>
<tr>
<td>'mosaic'</td>
<td>BackgroundBrowser('mosaic')</td>
<td>(1)</td>
</tr>
<tr>
<td>'opera'</td>
<td>Opera()</td>
<td></td>
</tr>
<tr>
<td>'grail'</td>
<td>Grail()</td>
<td></td>
</tr>
<tr>
<td>'links'</td>
<td>GenericBrowser('links')</td>
<td></td>
</tr>
<tr>
<td>'elinks'</td>
<td>Elinks('elinks')</td>
<td></td>
</tr>
<tr>
<td>'lynx'</td>
<td>GenericBrowser('lynx')</td>
<td></td>
</tr>
<tr>
<td>'w3m'</td>
<td>GenericBrowser('w3m')</td>
<td></td>
</tr>
<tr>
<td>'windows-default'</td>
<td>WindowsDefault</td>
<td>(2)</td>
</tr>
<tr>
<td>'macosx'</td>
<td>MacOSX('default')</td>
<td>(3)</td>
</tr>
<tr>
<td>'safari'</td>
<td>MacOSX('safari')</td>
<td>(3)</td>
</tr>
<tr>
<td>'google-chrome'</td>
<td>Chrome('google-chrome')</td>
<td></td>
</tr>
<tr>
<td>'chrome'</td>
<td>Chrome('chrome')</td>
<td></td>
</tr>
<tr>
<td>'chromium'</td>
<td>Chromium('chromium')</td>
<td></td>
</tr>
<tr>
<td>'chromium-browser'</td>
<td>Chromium('chromium-browser')</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. “Konqueror” is the file manager for the KDE desktop environment for Unix, and only makes sense to use if KDE is running. Some way of reliably detecting KDE would be nice; the `KDEDIR` variable is not sufficient. Note also that the name “kfm” is used even when using the `konqueror` command with KDE 2 — the implementation selects the best strategy for running Konqueror.

2. Only on Windows platforms.

3. Only on Mac OS X platform.

New in version 3.3: Support for Chrome/Chromium has been added. Here are some simple examples:
21.1.1 Browser Controller Objects

Browser controllers provide these methods which parallel three of the module-level convenience functions:

controller.**open**(url, new=0, autoraise=True)

Display url using the browser handled by this controller. If new is 1, a new browser window is opened if possible. If new is 2, a new browser page ("tab") is opened if possible.

controller.**open_new**(url)

Open url in a new window of the browser handled by this controller, if possible, otherwise, open url in the only browser window. Alias open_new().

controller.**open_new_tab**(url)

Open url in a new page ("tab") of the browser handled by this controller, if possible, otherwise equivalent to open_new().

21.2 **cgi** — Common Gateway Interface support

*Source code:* Lib/cgi.py

Support module for Common Gateway Interface (CGI) scripts.

This module defines a number of utilities for use by CGI scripts written in Python.

21.2.1 Introduction

A CGI script is invoked by an HTTP server, usually to process user input submitted through an HTML `<FORM>` or `<ISINDEX>` element.

Most often, CGI scripts live in the server’s special `cgi-bin` directory. The HTTP server places all sorts of information about the request (such as the client’s hostname, the requested URL, the query string, and lots of other goodies) in the script’s shell environment, executes the script, and sends the script’s output back to the client.

The script’s input is connected to the client too, and sometimes the form data is read this way; at other times the form data is passed via the “query string” part of the URL. This module is intended to take care of the different cases and provide a simpler interface to the Python script. It also provides a number of utilities that help in debugging scripts, and the latest addition is support for file uploads from a form (if your browser supports it).

The output of a CGI script should consist of two sections, separated by a blank line. The first section contains a number of headers, telling the client what kind of data is following. Python code to generate a minimal header section looks like this:

```python
print("Content-Type: text/html")  # HTML is following
print()                          # blank line, end of headers
```

The second section is usually HTML, which allows the client software to display nicely formatted text with header, in-line images, etc. Here’s Python code that prints a simple piece of HTML:
print("<TITLE>CGI script output</TITLE>")
print("<H1>This is my first CGI script</H1>")
print("Hello, world!")

21.2.2 Using the cgi module

Begin by writing import cgi.

When you write a new script, consider adding these lines:

```python
import cgi
cgitb.enable()
```

This activates a special exception handler that will display detailed reports in the Web browser if any errors occur. If you’d rather not show the guts of your program to users of your script, you can have the reports saved to files instead, with code like this:

```python
import cgi
cgitb.enable(display=0, logdir="/path/to/logdir")
```

It’s very helpful to use this feature during script development. The reports produced by `cgitb` provide information that can save you a lot of time in tracking down bugs. You can always remove the `cgitb` line later when you have tested your script and are confident that it works correctly.

To get at submitted form data, use the `FieldStorage` class. If the form contains non-ASCII characters, use the `encoding` keyword parameter set to the value of the encoding defined for the document. It is usually contained in the META tag in the HEAD section of the HTML document or by the `Content-Type` header. This reads the form contents from the standard input or the environment (depending on the value of various environment variables set according to the CGI standard). Since it may consume standard input, it should be instantiated only once.

The `FieldStorage` instance can be indexed like a Python dictionary. It allows membership testing with the `in` operator, and also supports the standard dictionary method `keys()` and the built-in function `len()`. Form fields containing empty strings are ignored and do not appear in the dictionary; to keep such values, provide a true value for the optional `keep_blank_values` keyword parameter when creating the `FieldStorage` instance.

For instance, the following code (which assumes that the `Content-Type` header and blank line have already been printed) checks that the fields `name` and `addr` are both set to a non-empty string:

```python
form = cgi.FieldStorage()
if "name" not in form or "addr" not in form:
    print("<H1>Error</H1>")
    print("Please fill in the name and addr fields.")
    return
print("<p>name:", form["name"].value)
print("<p>addr:", form["addr"].value)
...further form processing here...
```

Here the fields, accessed through `form[key]`, are themselves instances of `FieldStorage` (or `MiniFieldStorage`, depending on the form encoding). The `value` attribute of the instance yields the string value of the field. The `getvalue()` method returns this string value directly; it also accepts an optional second argument as a default to return if the requested key is not present.

If the submitted form data contains more than one field with the same name, the object retrieved by `form[key]` is not a `FieldStorage` or `MiniFieldStorage` instance but a list of such instances. Similarly, in this situation, `form.getvalue(key)` would return a list of strings. If you expect this possibility (when your HTML form contains multiple fields with the same name), use the `getlist()` method, which always returns a list of values (so that you do not need to special-case the single item case). For example, this code concatenates any number of username fields, separated by commas:

```python
value = form.getlist("username")
usernames = ",".join(value)
```
If a field represents an uploaded file, accessing the value via the value attribute or the getvalue() method reads the entire file in memory as bytes. This may not be what you want. You can test for an uploaded file by testing either the filename attribute or the file attribute. You can then read the data at leisure from the file attribute (the read() and readline() methods will return bytes):

```python
fileitem = form["userfile"]
if fileitem.file:
    # It’s an uploaded file; count lines
    linecount = 0
    while True:
        line = fileitem.file.readline()
        if not line: break
        linecount = linecount + 1
```

If an error is encountered when obtaining the contents of an uploaded file (for example, when the user interrupts the form submission by clicking on a Back or Cancel button) the done attribute of the object for the field will be set to the value -1.

The file upload draft standard entertains the possibility of uploading multiple files from one field (using a recursive multipart/* encoding). When this occurs, the item will be a dictionary-like FieldStorage item. This can be determined by testing its type attribute, which should be multipart/form-data (or perhaps another MIME type matching multipart/*). In this case, it can be iterated over recursively just like the top-level form object.

When a form is submitted in the “old” format (as the query string or as a single data part of type application/x-www-form-urlencoded), the items will actually be instances of the class MiniFieldStorage. In this case, the list, file, and filename attributes are always None.

A form submitted via POST that also has a query string will contain both FieldStorage and MiniFieldStorage items.

### 21.2.3 Higher Level Interface

The previous section explains how to read CGI form data using the FieldStorage class. This section describes a higher level interface which was added to this class to allow one to do it in a more readable and intuitive way. The interface doesn’t make the techniques described in previous sections obsolete — they are still useful to process file uploads efficiently, for example.

The interface consists of two simple methods. Using the methods you can process form data in a generic way, without the need to worry whether only one or more values were posted under one name.

In the previous section, you learned to write following code anytime you expected a user to post more than one value under one name:

```python
item = form.getvalue("item")
if isinstance(item, list):
    # The user is requesting more than one item.
else:
    # The user is requesting only one item.
```

This situation is common for example when a form contains a group of multiple checkboxes with the same name:

```html
<input type="checkbox" name="item" value="1" />
<input type="checkbox" name="item" value="2" />
```

In most situations, however, there’s only one form control with a particular name in a form and then you expect and need only one value associated with this name. So you write a script containing for example this code:

```python
user = form.getvalue("user").upper()
```

The problem with the code is that you should never expect that a client will provide valid input to your scripts. For example, if a curious user appends another user=foo pair to the query string, then the script would
crash, because in this situation the `getvalue("user")` method call returns a list instead of a string. Calling the `upper()` method on a list is not valid (since lists do not have a method of this name) and results in an `AttributeError` exception.

Therefore, the appropriate way to read form data values was to always use the code which checks whether the obtained value is a single value or a list of values. That’s annoying and leads to less readable scripts.

A more convenient approach is to use the methods `getfirst()` and `getlist()` provided by this higher level interface.

**FieldStorage.getfirst**(name, default=None)

This method always returns only one value associated with form field name. The method returns only the first value in case that more values were posted under such name. Please note that the order in which the values are received may vary from browser to browser and should not be counted on. If no such form field or value exists then the method returns the value specified by the optional parameter default. This parameter defaults to None if not specified.

**FieldStorage.getlist**(name)

This method always returns a list of values associated with form field name. The method returns an empty list if no such form field or value exists for name. It returns a list consisting of one item if only one such value exists.

Using these methods you can write nice compact code:

```python
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", ").upper() # This way it's safe.
for item in form.getlist("item"):
    do_something(item)
```

### 21.2.4 Functions

These are useful if you want more control, or if you want to employ some of the algorithms implemented in this module in other circumstances.

**cgi.parse**(fp=None, environ=os.environ, keep_blank_values=False, strict_parsing=False)

Parse a query in the environment or from a file (the file defaults to `sys.stdin`). The `keep_blank_values` and `strict_parsing` parameters are passed to `urllib.parse.parse_qs()` unchanged.

**cgi.parse_qs**(qs, keep_blank_values=False, strict_parsing=False)

This function is deprecated in this module. Use `urllib.parse.parse_qs()` instead. It is maintained here only for backward compatibility.

**cgi.parse_qsl**(qs, keep_blank_values=False, strict_parsing=False)

This function is deprecated in this module. Use `urllib.parse.parse_qs()` instead. It is maintained here only for backward compatibility.

**cgi.parse_multipart**(fp, pdict)

Parse input of type `multipart/form-data` (for file uploads). Arguments are `fp` for the input file and `pdict` for a dictionary containing other parameters in the `Content-Type` header.

Returns a dictionary just like `urllib.parse.parse_qs()` keys are the field names, each value is a list of values for that field. This is easy to use but not much good if you are expecting megabytes to be uploaded — in that case, use the `FieldStorage` class instead which is much more flexible.

Note that this does not parse nested multipart parts — use `FieldStorage` for that.

**cgi.parse_header**(string)

Parse a MIME header (such as `Content-Type`) into a main value and a dictionary of parameters.

---

2 Note that some recent versions of the HTML specification do state what order the field values should be supplied in, but knowing whether a request was received from a conforming browser, or even from a browser at all, is tedious and error-prone.
The Python Library Reference, Release 3.3.3

21.2.5 Caring about security

There’s one important rule: if you invoke an external program (via the os.system() or os.popen() functions, or others with similar functionality), make very sure you don’t pass arbitrary strings received from the client to the shell. This is a well-known security hole whereby clever hackers anywhere on the Web can exploit a gullible CGI script to invoke arbitrary shell commands. Even parts of the URL or field names cannot be trusted, since the request doesn’t have to come from your form!

To be on the safe side, if you must pass a string gotten from a form to a shell command, you should make sure the string contains only alphanumeric characters, dashes, underscores, and periods.

21.2.6 Installing your CGI script on a Unix system

Read the documentation for your HTTP server and check with your local system administrator to find the directory where CGI scripts should be installed; usually this is in a directory cgi-bin in the server tree.

Make sure that your script is readable and executable by “others”; the Unix file mode should be 0655 octal (use chmod 0755 filename). Make sure that the first line of the script contains #! starting in column 1 followed by the pathname of the Python interpreter, for instance:

```
#!/usr/local/bin/python
```

Make sure the Python interpreter exists and is executable by “others”.

Make sure that any files your script needs to read or write are readable or writable, respectively, by “others” — their mode should be 0644 for readable and 0666 for writable. This is because, for security reasons, the HTTP server executes your script as user “nobody”, without any special privileges. It can only read (write, execute) files that everybody can read (write, execute). The current directory at execution time is also different (it is usually the server’s cgi-bin directory) and the set of environment variables is also different from what you get when you log in. In particular, don’t count on the shell’s search path for executables (PATH) or the Python module search path (PYTHONPATH) to be set to anything interesting.

If you need to load modules from a directory which is not on Python’s default module search path, you can change the path in your script, before importing other modules. For example:

```
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
sys.path.insert(0, "/usr/local/lib/python")
```
Instructions for non-Unix systems will vary; check your HTTP server’s documentation (it will usually have a section on CGI scripts).

21.2.7 Testing your CGI script

Unfortunately, a CGI script will generally not run when you try it from the command line, and a script that works perfectly from the command line may fail mysteriously when run from the server. There’s one reason why you should still test your script from the command line: if it contains a syntax error, the Python interpreter won’t execute it at all, and the HTTP server will most likely send a cryptic error to the client.

Assuming your script has no syntax errors, yet it does not work, you have no choice but to read the next section.

21.2.8 Debugging CGI scripts

First of all, check for trivial installation errors — reading the section above on installing your CGI script carefully can save you a lot of time. If you wonder whether you have understood the installation procedure correctly, try installing a copy of this module file (cgi.py) as a CGI script. When invoked as a script, the file will dump its environment and the contents of the form in HTML form. Give it the right mode etc. and send it a request. If it’s installed in the standard cgi-bin directory, it should be possible to send it a request by entering a URL into your browser of the form:

http://yourhostname/cgi-bin/cgi.py?name=Joe+Blow&addr=At+Home

If this gives an error of type 404, the server cannot find the script – perhaps you need to install it in a different directory. If it gives another error, there’s an installation problem that you should fix before trying to go any further. If you get a nicely formatted listing of the environment and form content (in this example, the fields should be listed as “addr” with value “At Home” and “name” with value “Joe Blow”), the cgi.py script has been installed correctly. If you follow the same procedure for your own script, you should now be able to debug it.

The next step could be to call the cgi module’s test() function from your script: replace its main code with the single statement

cgi.test()

This should produce the same results as those gotten from installing the cgi.py file itself.

When an ordinary Python script raises an unhandled exception (for whatever reason: of a typo in a module name, a file that can’t be opened, etc.), the Python interpreter prints a nice traceback and exits. While the Python interpreter will still do this when your CGI script raises an exception, most likely the traceback will end up in one of the HTTP server’s log files, or be discarded altogether.

Fortunately, once you have managed to get your script to execute some code, you can easily send tracebacks to the Web browser using the cgitb module. If you haven’t done so already, just add the lines:

```python
import cgitb
cgitb.enable()
```

to the top of your script. Then try running it again; when a problem occurs, you should see a detailed report that will likely make apparent the cause of the crash.

If you suspect that there may be a problem in importing the cgitb module, you can use an even more robust approach (which only uses built-in modules):

```python
import sys
sys.stderr = sys.stdout
print("Content-Type: text/plain")
print()
...your code here...
```

This relies on the Python interpreter to print the traceback. The content type of the output is set to plain text, which disables all HTML processing. If your script works, the raw HTML will be displayed by your client. If it
raises an exception, most likely after the first two lines have been printed, a traceback will be displayed. Because no HTML interpretation is going on, the traceback will be readable.

21.2.9 Common problems and solutions

• Most HTTP servers buffer the output from CGI scripts until the script is completed. This means that it is not possible to display a progress report on the client’s display while the script is running.
• Check the installation instructions above.
• Check the HTTP server’s log files. (tail -f logfile in a separate window may be useful!)
• Always check a script for syntax errors first, by doing something like python script.py.
• If your script does not have any syntax errors, try adding import cgitb; cgitb.enable() to the top of the script.
• When invoking external programs, make sure they can be found. Usually, this means using absolute path names — PATH is usually not set to a very useful value in a CGI script.
• When reading or writing external files, make sure they can be read or written by the userid under which your CGI script will be running: this is typically the userid under which the web server is running, or some explicitly specified userid for a web server’s suexec feature.
• Don’t try to give a CGI script a set-uid mode. This doesn’t work on most systems, and is a security liability as well.

21.3 cgitb — Traceback manager for CGI scripts

The cgitb module provides a special exception handler for Python scripts. (Its name is a bit misleading. It was originally designed to display extensive traceback information in HTML for CGI scripts. It was later generalized to also display this information in plain text.) After this module is activated, if an uncaught exception occurs, a detailed, formatted report will be displayed. The report includes a traceback showing excerpts of the source code for each level, as well as the values of the arguments and local variables to currently running functions, to help you debug the problem. Optionally, you can save this information to a file instead of sending it to the browser.

To enable this feature, simply add this to the top of your CGI script:

import cgitb

cgitb.enable()

The options to the enable() function control whether the report is displayed in the browser and whether the report is logged to a file for later analysis.

cgitb.enable(display=1, logdir=None, context=5, format=’html’)

This function causes the cgitb module to take over the interpreter’s default handling for exceptions by setting the value of sys.excepthook.

The optional argument display defaults to 1 and can be set to 0 to suppress sending the traceback to the browser. If the argument logdir is present, the traceback reports are written to files. The value of logdir should be a directory where these files will be placed. The optional argument context is the number of lines of context to display around the current line of source code in the traceback; this defaults to 5. If the optional argument format is "html", the output is formatted as HTML. Any other value forces plain text output. The default value is "html".

cgitb.handler(info=None)

This function handles an exception using the default settings (that is, show a report in the browser, but don’t log to a file). This can be used when you’ve caught an exception and want to report it using cgitb. The optional info argument should be a 3-tuple containing an exception type, exception value, and traceback object, exactly like the tuple returned by sys.exc_info(). If the info argument is not supplied, the current exception is obtained from sys.exc_info().

21.3. cgitb — Traceback manager for CGI scripts 853
21.4 wsgiref — WSGI Utilities and Reference Implementation

The Web Server Gateway Interface (WSGI) is a standard interface between web server software and web applications written in Python. Having a standard interface makes it easy to use an application that supports WSGI with a number of different web servers.

Only authors of web servers and programming frameworks need to know every detail and corner case of the WSGI design. You don’t need to understand every detail of WSGI just to install a WSGI application or to write a web application using an existing framework.

wsgiref is a reference implementation of the WSGI specification that can be used to add WSGI support to a web server or framework. It provides utilities for manipulating WSGI environment variables and response headers, base classes for implementing WSGI servers, a demo HTTP server that serves WSGI applications, and a validation tool that checks WSGI servers and applications for conformance to the WSGI specification (PEP 3333).

See http://www.wsgi.org for more information about WSGI, and links to tutorials and other resources.

21.4.1 wsgiref.util — WSGI environment utilities

This module provides a variety of utility functions for working with WSGI environments. A WSGI environment is a dictionary containing HTTP request variables as described in PEP 3333. All of the functions taking an environ parameter expect a WSGI-compliant dictionary to be supplied; please see PEP 3333 for a detailed specification.

wsgiref.util.guess_scheme(environ)

Return a guess for whether wsgi.url_scheme should be “http” or “https”, by checking for a HTTPS environment variable in the environ dictionary. The return value is a string.

This function is useful when creating a gateway that wraps CGI or a CGI-like protocol such as FastCGI. Typically, servers providing such protocols will include a HTTPS variable with a value of “1” “yes”, or “on” when a request is received via SSL. So, this function returns “https” if such a value is found, and “http” otherwise.

wsgiref.util.request_uri(environ, include_query=True)

Return the full request URI, optionally including the query string, using the algorithm found in the “URL Reconstruction” section of PEP 3333. If include_query is false, the query string is not included in the resulting URI.

wsgiref.util.application_uri(environ)

Similar to request_uri(), except that the PATH_INFO and QUERY_STRING variables are ignored. The result is the base URI of the application object addressed by the request.

wsgiref.util.shift_path_info(environ)

Shift a single name from PATH_INFO to SCRIPT_NAME and return the name. The environ dictionary is modified in-place; use a copy if you need to keep the original PATH_INFO or SCRIPT_NAME intact.

If there are no remaining path segments in PATH_INFO, None is returned.

Typically, this routine is used to process each portion of a request URI path, for example to treat the path as a series of dictionary keys. This routine modifies the passed-in environment to make it suitable for invoking another WSGI application that is located at the target URI. For example, if there is a WSGI application at /foo, and the request URI path is /foo/bar/baz, and the WSGI application at /foo calls shift_path_info(), it will receive the string “bar”, and the environment will be updated to be suitable for passing to a WSGI application at /foo/bar. That is, SCRIPT_NAME will change from /foo to /foo/bar, and PATH_INFO will change from /bar/baz to /baz.

When PATH_INFO is just a “/”, this routine returns an empty string and appends a trailing slash to SCRIPT_NAME, even though empty path segments are normally ignored, and SCRIPT_NAME doesn’t normally end in a slash. This is intentional behavior, to ensure that an application can tell the difference between URIs ending in /x from ones ending in /x/ when using this routine to do object traversal.
wsgiref.util.setup_testing_defaults(environ)

Update environ with trivial defaults for testing purposes.

This routine adds various parameters required for WSGI, including HTTP_HOST, SERVER_NAME, SERVER_PORT, REQUEST_METHOD, SCRIPT_NAME, PATH_INFO, and all of the PEP 3333-defined wsgi.* variables. It only supplies default values, and does not replace any existing settings for these variables.

This routine is intended to make it easier for unit tests of WSGI servers and applications to set up dummy environments. It should NOT be used by actual WSGI servers or applications, since the data is fake!

Example usage:

```python
from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# A relatively simple WSGI application. It’s going to print out the
# environment dictionary after being updated by setup_testing_defaults
def simple_app(environ, start_response):
    setup_testing_defaults(environ)
    status = '200 OK'
    headers = [('Content-type', 'text/plain; charset=utf-8')]
    start_response(status, headers)
    ret = [("%s: %s\n" % (key, value)).encode("utf-8")
          for key, value in environ.items()]
    return ret

httpd = make_server('', 8000, simple_app)
print("Serving on port 8000...")
httpd.serve_forever()
```

In addition to the environment functions above, the wsgiref.util module also provides these miscellaneous utilities:

wsgiref.util.is_hop_by_hop(header_name)

Return true if ‘header_name’ is an HTTP/1.1 “Hop-by-Hop” header, as defined by RFC 2616.

class wsgiref.util.FileWrapper(filelike, blksize=8192)

A wrapper to convert a file-like object to an iterator. The resulting objects support both __getitem__() and __iter__() iteration styles, for compatibility with Python 2.1 and Jython. As the object is iterated over, the optional blksize parameter will be repeatedly passed to the filelike object’s read() method to obtain bytestrings to yield. When read() returns an empty bytestring, iteration is ended and is not resumable.

If filelike has a close() method, the returned object will also have a close() method, and it will invoke the filelike object’s close() method when called.

Example usage:

```python
from io import StringIO
from wsgiref.util import FileWrapper

# We're using a StringIO-buffer for as the file-like object
filelike = StringIO("This is an example file-like object"*10)
wrapper = FileWrapper(filelike, blksize=5)
```
for chunk in wrapper:
    print(chunk)

21.4.2 wsgiref.headers – WSGI response header tools

This module provides a single class, Headers, for convenient manipulation of WSGI response headers using a mapping-like interface.

class wsgiref.headers.Headers(headers)

Create a mapping-like object wrapping headers, which must be a list of header name/value tuples as described in PEP 3333.

Headers objects support typical mapping operations including __getitem__(), get(), __setitem__(), setdefault(), __delitem__() and __contains__() and delimiters ( ). For each of these methods, the key is the header name (treated case-insensitively), and the value is the first value associated with that header name. Setting a header deletes any existing values for that header, then adds a new value at the end of the wrapped header list. Headers’ existing order is generally maintained, with new headers added to the end of the wrapped list.

Unlike a dictionary, Headers objects do not raise an error when you try to get or delete a key that isn’t in the wrapped header list. Getting a nonexistent header just returns None, and deleting a nonexistent header does nothing.

Headers objects also support keys(), values(), and items() methods. The lists returned by keys() and items() can include the same key more than once if there is a multi-valued header. The len() of a Headers object is the same as the length of its items(), which is the same as the length of the wrapped header list. In fact, the items() method just returns a copy of the wrapped header list.

Calling bytes() on a Headers object returns a formatted bytestring suitable for transmission as HTTP response headers. Each header is placed on a line with its value, separated by a colon and a space. Each line is terminated by a carriage return and line feed, and the bytestring is terminated with a blank line.

In addition to their mapping interface and formatting features, Headers objects also have the following methods for querying and adding multi-valued headers, and for adding headers with MIME parameters:

get_all(name)

Return a list of all the values for the named header.

The returned list will be sorted in the order they appeared in the original header list or were added to this instance, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list. If no fields exist with the given name, returns an empty list.

add_header(name, value, **_params)

Add a (possibly multi-valued) header, with optional MIME parameters specified via keyword arguments.

name is the header field to add. Keyword arguments can be used to set MIME parameters for the header field. Each parameter must be a string or None. Underscores in parameter names are converted to dashes, since dashes are illegal in Python identifiers, but many MIME parameter names include dashes. If the parameter value is a string, it is added to the header value parameters in the form name="value". If it is None, only the parameter name is added. (This is used for MIME parameters without a value.) Example usage:

h.add_header('content-disposition', 'attachment', filename='bud.gif')

The above will add a header that looks like this:

Content-Disposition: attachment; filename="bud.gif"
21.4.3 wsgiref.simple_server — a simple WSGI HTTP server

This module implements a simple HTTP server (based on http.server) that serves WSGI applications. Each server instance serves a single WSGI application on a given host and port. If you want to serve multiple applications on a single host and port, you should create a WSGI application that parses PATH_INFO to select which application to invoke for each request. (E.g., using the shift_path_info() function from wsgiref.util.)

```python
def make_server(host, port, app, server_class=WSGIServer, handler_class=WSGIRequestHandler):
    Create a new WSGI server listening on host and port, accepting connections for app. The return value is an instance of the supplied server_class, and will process requests using the specified handler_class. app must be a WSGI application object, as defined by PEP 3333.
```

Example usage:

```python
from wsgiref.simple_server import make_server, demo_app

httpd = make_server('', 8000, demo_app)
print("Serving HTTP on port 8000...")

# Respond to requests until process is killed
httpd.serve_forever()

# Alternative: serve one request, then exit
httpd.handle_request()
```

```python
def demo_app(environ, start_response):
    This function is a small but complete WSGI application that returns a text page containing the message “Hello world!” and a list of the key/value pairs provided in the environ parameter. It’s useful for verifying that a WSGI server (such as wsgiref.simple_server) is able to run a simple WSGI application correctly.
```

```python
class WSGIServer(server_address, RequestHandlerClass):
    Create a WSGIServer instance. server_address should be a (host, port) tuple, and RequestHandlerClass should be the subclass of http.server.BaseHTTPRequestHandler that will be used to process requests.

    You do not normally need to call this constructor, as the make_server() function can handle all the details for you.

    WSGIServer is a subclass of http.server.HTTPServer, so all of its methods (such as serve_forever() and handle_request()) are available. WSGIServer also provides these WSGI-specific methods:

    set_app(application)
        Sets the callable application as the WSGI application that will receive requests.

    get_app()
        Returns the currently-set application callable.

        Normally, however, you do not need to use these additional methods, as set_app() is normally called by make_server(), and the get_app() exists mainly for the benefit of request handler instances.
```

```python
class WSGIRequestHandler(request, client_address, server)
    Create an HTTP handler for the given request (i.e. a socket), client_address (a (host, port) tuple), and server (WSGIServer instance).

    You do not need to create instances of this class directly; they are automatically created as needed by WSGIServer objects. You can, however, subclass this class and supply it as a handler_class to the make_server() function. Some possibly relevant methods for overriding in subclasses:

    get_environ()
        Returns a dictionary containing the WSGI environment for a request. The default implementation copies the contents of the WSGIServer object’s base_environ dictionary attribute and then adds
various headers derived from the HTTP request. Each call to this method should return a new dictionary containing all of the relevant CGI environment variables as specified in PEP 3333.

**get_stderr()**
Return the object that should be used as the wsgi.errors stream. The default implementation just returns sys.stderr.

**handle()**
Process the HTTP request. The default implementation creates a handler instance using a wsgiref.handlers class to implement the actual WSGI application interface.

### 21.4.4 wsgiref.validate — WSGI conformance checker

When creating new WSGI application objects, frameworks, servers, or middleware, it can be useful to validate the new code’s conformance using wsgiref.validate. This module provides a function that creates WSGI application objects that validate communications between a WSGI server or gateway and a WSGI application object, to check both sides for protocol conformance.

Note that this utility does not guarantee complete PEP 3333 compliance; an absence of errors from this module does not necessarily mean that errors do not exist. However, if this module does produce an error, then it is virtually certain that either the server or application is not 100% compliant.

This module is based on the paste.lint module from Ian Bicking’s “Python Paste” library.

```
from wsgiref.validate import validator

validator_app = validator(simple_app)

httpd = make_server('', 8000, validator_app)
```

Example usage:

```python
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# Our callable object which is intentionally not compliant to the standard, so the validator is going to break

def simple_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)

    # This is going to break because we need to return a list, and the validator is going to inform us
    return [b"Hello World"]

# This is the application wrapped in a validator
validator_app = validator(simple_app)

httpd = make_server('', 8000, validator_app)
```
print("Listening on port 8000....")
httpd.serve_forever()

21.4.5 wsgiref.handlers – server/gateway base classes

This module provides base handler classes for implementing WSGI servers and gateways. These base classes handle most of the work of communicating with a WSGI application, as long as they are given a CGI-like environment, along with input, output, and error streams.

class wsgiref.handlers.CGIHandler

CGI-based invocation via `sys.stdin`, `sys.stdout`, `sys.stderr` and `os.environ`. This is useful when you have a WSGI application and want to run it as a CGI script. Simply invoke `CGIHandler().run(app)`, where `app` is the WSGI application object you wish to invoke.

This class is a subclass of `BaseCGIHandler` that sets `wsgi.run_once` to `true`, `wsgi.multithread` to `false`, and `wsgi.multiprocess` to `true`, and always uses `sys` and `os` to obtain the necessary CGI streams and environment.

class wsgiref.handlers.IISCGIHandler

A specialized alternative to `CGIHandler`, for use when deploying on Microsoft’s IIS web server, without having set the config allowPathInfo option (IIS>=7) or metabase allowPathInfoForScriptMappings (IIS<7).

By default, IIS gives a `PATH_INFO` that duplicates the `SCRIPT_NAME` at the front, causing problems for WSGI applications that wish to implement routing. This handler strips any such duplicated path.

IIS can be configured to pass the correct `PATH_INFO`, but this causes another bug where `PATH_TRANSLATED` is wrong. Luckily this variable is rarely used and is not guaranteed by WSGI. On IIS<7, though, the setting can only be made on a vhost level, affecting all other script mappings, many of which break when exposed to the `PATH_TRANSLATED` bug. For this reason IIS<7 is almost never deployed with the fix. (Even IIS7 rarely uses it because there is still no UI for it.)

There is no way for CGI code to tell whether the option was set, so a separate handler class is provided. It is used in the same way as `CGIHandler`, i.e., by calling `IISCGIHandler().run(app)`, where `app` is the WSGI application object you wish to invoke. New in version 3.2.

class wsgiref.handlers.BaseCGIHandler (stdin, stdout, stderr, environ, multithread=True, multiprocess=False)

Similar to `CGIHandler`, but instead of using the `sys` and `os` modules, the CGI environment and I/O streams are specified explicitly. The `multithread` and `multiprocess` values are used to set the `wsgi.multithread` and `wsgi.multiprocess` flags for any applications run by the handler instance.

This class is a subclass of `SimpleHandler` intended for use with software other than HTTP “origin servers”. If you are writing a gateway protocol implementation (such as CGI, FastCGI, SCGI, etc.) that uses a `Status:` header to send an HTTP status, you probably want to subclass this instead of `SimpleHandler`.

class wsgiref.handlers.SimpleHandler (stdin, stdout, stderr, environ, multithread=True, multiprocess=False)

Similar to `BaseCGIHandler`, but designed for use with HTTP origin servers. If you are writing an HTTP server implementation, you will probably want to subclass this instead of `BaseCGIHandler`.

This class is a subclass of `BaseHandler`. It overrides the `__init__()`, `get_stdin()`, `get_stderr()`, `add_cgi_vars()`, `__write()`, and `__flush()` methods to support explicitly setting the environment and streams via the constructor. The supplied environment and streams are stored in the `stdin`, `stdout`, `stderr`, and `environ` attributes.

class wsgiref.handlers.BaseHandler

This is an abstract base class for running WSGI applications. Each instance will handle a single HTTP request, although in principle you could create a subclass that was reusable for multiple requests.

`BaseHandler` instances have only one method intended for external use:

```python
run(app)
```
Run the specified WSGI application, `app`. 859
All of the other BaseHandler methods are invoked by this method in the process of running the application, and thus exist primarily to allow customizing the process.

The following methods MUST be overridden in a subclass:

**_write (data)**
Buffer the bytes data for transmission to the client. It’s okay if this method actually transmits the data; BaseHandler just separates write and flush operations for greater efficiency when the underlying system actually has such a distinction.

**_flush ()**
Force buffered data to be transmitted to the client. It’s okay if this method is a no-op (i.e., if _write() actually sends the data).

**get_stdin ()**
Return an input stream object suitable for use as the wsgi.input of the request currently being processed.

**get_stderr ()**
Return an output stream object suitable for use as the wsgi.errors of the request currently being processed.

**add_cgi_vars ()**
Insert CGI variables for the current request into the environ attribute.

Here are some other methods and attributes you may wish to override. This list is only a summary, however, and does not include every method that can be overridden. You should consult the docstrings and source code for additional information before attempting to create a customized BaseHandler subclass.

Attributes and methods for customizing the WSGI environment:

**wsgi_multithread**
The value to be used for the wsgi.multithread environment variable. It defaults to true in BaseHandler, but may have a different default (or be set by the constructor) in the other subclasses.

**wsgi_multiprocess**
The value to be used for the wsgi.multiprocess environment variable. It defaults to true in BaseHandler, but may have a different default (or be set by the constructor) in the other subclasses.

**wsgi_run_once**
The value to be used for the wsgi.run_once environment variable. It defaults to false in BaseHandler, but CGIHandler sets it to true by default.

**os_environ**
The default environment variables to be included in every request’s WSGI environment. By default, this is a copy of os.environ at the time that wsgiref.handlers was imported, but subclasses can either create their own at the class or instance level. Note that the dictionary should be considered read-only, since the default value is shared between multiple classes and instances.

**server_software**
If the origin_server attribute is set, this attribute’s value is used to set the default SERVER_SOFTWARE WSGI environment variable, and also to set a default Server: header in HTTP responses. It is ignored for handlers (such as BaseCGIHandler and CGIHandler) that are not HTTP origin servers. Changed in version 3.3: The term “Python” is replaced with implementation specific term like “CPython”, “Jython” etc.

**get_scheme ()**
Return the URL scheme being used for the current request. The default implementation uses the guess_scheme() function from wsgiref.util to guess whether the scheme should be “http” or “https”, based on the current request’s environ variables.

**setup_environ ()**
Set the environ attribute to a fully-populated WSGI environment. The default implementation uses all of the above methods and attributes, plus the get_stdin(), get.stderr(), and add_cgi_vars() methods and the wsgi_file_wrapper attribute. It also inserts a
SERVERSOFTWARE key if not present, as long as the origin_server attribute is a true value and the server_software attribute is set.

Methods and attributes for customizing exception handling:

**log_exception** (exc_info)
Log the exc_info tuple in the server log. exc_info is a (type, value, traceback) tuple. The default implementation simply writes the traceback to the request’s wsgi.errors stream and flushes it. Subclasses can override this method to change the format or retarget the output, mail the traceback to an administrator, or whatever other action may be deemed suitable.

**traceback_limit**
The maximum number of frames to include in tracebacks output by the default log_exception() method. If None, all frames are included.

**error_output** (environ, start_response)
This method is a WSGI application to generate an error page for the user. It is only invoked if an error occurs before headers are sent to the client.

This method can access the current error information using sys.exc_info(), and should pass that information to start_response when calling it (as described in the “Error Handling” section of PEP 3333).

The default implementation just uses the error_status, error_headers, and error_body attributes to generate an output page. Subclasses can override this to produce more dynamic error output.

Note, however, that it’s not recommended from a security perspective to spit out diagnostics to any old user; ideally, you should have to do something special to enable diagnostic output, which is why the default implementation doesn’t include any.

**error_status**
The HTTP status used for error responses. This should be a status string as defined in PEP 3333; it defaults to a 500 code and message.

**error_headers**
The HTTP headers used for error responses. This should be a list of WSGI response headers ((name, value) tuples), as described in PEP 3333. The default list just sets the content type to text/plain.

**error_body**
The error response body. This should be an HTTP response body bytestring. It defaults to the plain text, “A server error occurred. Please contact the administrator.”

Methods and attributes for PEP 3333’s “Optional Platform-Specific File Handling” feature:

**wsgi_file_wrapper**
A wsgi.file_wrapper factory, or None. The default value of this attribute is the wsgiref.util.FileWrapper class.

**sendfile**()
Override to implement platform-specific file transmission. This method is called only if the application’s return value is an instance of the class specified by the wsgi_file_wrapper attribute. It should return a true value if it was able to successfully transmit the file, so that the default transmission code will not be executed. The default implementation of this method just returns a false value.

Miscellaneous methods and attributes:

**origin_server**
This attribute should be set to a true value if the handler’s _write() and _flush() are being used to communicate directly to the client, rather than via a CGI-like gateway protocol that wants the HTTP status in a special Status: header.

This attribute’s default value is true in BaseHandler, but false in BaseCGIHandler and CGIHandler.
http_version

If `origin_server` is true, this string attribute is used to set the HTTP version of the response set to the client. It defaults to "1.0".

`wsgiref.handlers.read_environ()`

Transcode CGI variables from `os.environ` to PEP 3333 “bytes in unicode” strings, returning a new dictionary. This function is used by CGIHandler and IISCGIHandler in place of directly using `os.environ`, which is not necessarily WSGI-compliant on all platforms and web servers using Python 3 – specifically, ones where the OS’s actual environment is Unicode (i.e. Windows), or ones where the environment is bytes, but the system encoding used by Python to decode it is anything other than ISO-8859-1 (e.g. Unix systems using UTF-8).

If you are implementing a CGI-based handler of your own, you probably want to use this routine instead of just copying values out of `os.environ` directly. New in version 3.2.

21.4.6 Examples

This is a working “Hello World” WSGI application:

```python
from wsgiref.simple_server import make_server

# Every WSGI application must have an application object - a callable object that accepts two arguments. For that purpose, we’re going to use a function (note that you’re not limited to a function, you can use a class for example). The first argument passed to the function is a dictionary containing CGI-style environment variables and the second variable is the callable object (see PEP 333).

def hello_world_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain; charset=utf-8')]  # HTTP Headers
    start_response(status, headers)

    # The returned object is going to be printed
    return [b"Hello World"]

httpd = make_server('', 8000, hello_world_app)
print("Serving on port 8000...")

# Serve until process is killed
httpd.serve_forever()
```

21.5 `urllib` — URL handling modules

`urllib` is a package that collects several modules for working with URLs:

- `urllib.request` for opening and reading URLs
- `urllib.error` containing the exceptions raised by `urllib.request`
- `urllib.parse` for parsing URLs
- `urllib.robotparser` for parsing robots.txt files

21.6 `urllib.request` — Extensible library for opening URLs

The `urllib.request` module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections, cookies and more.
The `urllib.request` module defines the following functions:

```python
urllib.request.urlopen(url, data=None[, timeout], *, cafile=None, capath=None, cadatale=False)
```

Open the URL `url`, which can be either a string or a `Request` object.

data must be a bytes object specifying additional data to be sent to the server, or `None` if no such data is needed. data may also be an iterable object and in that case Content-Length value must be specified in the headers. Currently HTTP requests are the only ones that use data; the HTTP request will be a POST instead of a GET when the data parameter is provided.

data should be a buffer in the standard application/x-www-form-urlencoded format. The `urllib.parse.urlencode()` function takes a mapping or sequence of 2-tuples and returns a string in this format. It should be encoded to bytes before being used as the data parameter. The charset parameter in Content-Type header may be used to specify the encoding. If charset parameter is not sent with the Content-Type header, the server following the HTTP 1.1 recommendation may assume that the data is encoded in ISO-8859-1 encoding. It is advisable to use charset parameter with encoding used in Content-Type header with the Request.

urllib.request module uses HTTP/1.1 and includes Connection:close header in its HTTP requests.

The optional `timeout` parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). This actually only works for HTTP, HTTPS and FTP connections.

The optional `cafile` and `capath` parameters specify a set of trusted CA certificates for HTTPS requests. `cafile` should point to a single file containing a bundle of CA certificates, whereas `capath` should point to a directory of hashed certificate files. More information can be found in `ssl.SSLContext.load_verify_locations()`.

The `cadatale` parameter specifies whether to fall back to loading a default certificate store defined by the underlying OpenSSL library if the `cafile` and `capath` parameters are omitted. This will only work on some non-Windows platforms.

**Warning:** If neither `cafile` nor `capath` is specified, and `cadatale` is False, an HTTPS request will not do any verification of the server’s certificate.

For http and https urls, this function returns a `http.client.HTTPResponse` object which has the following `HTTPResponse` Objects methods.

For ftp, file, and data urls and requests explicitly handled by legacy `URLopener` and `FancyURLopener` classes, this function returns a `urllib.response.addinfourl` object which can work as `context manager` and has methods such as

* `geturl()` — return the URL of the resource retrieved, commonly used to determine if a redirect was followed

* `info()` — return the meta-information of the page, such as headers, in the form of an `email.message_from_string()` instance (see Quick Reference to HTTP Headers)

* `getcode()` — return the HTTP status code of the response.

Raises `URLError` on errors.

Note that `None` may be returned if no handler handles the request (though the default installed global `OpenerDirector` uses `UnknownHandler` to ensure this never happens).

In addition, if proxy settings are detected (for example, when a `*_proxy` environment variable like `http_proxy` is set), `ProxyHandler` is default installed and makes sure the requests are handled through the proxy.

The legacy `urllib.urlopen` function from Python 2.6 and earlier has been discontinued; `urllib.request.urlopen()` corresponds to the old `urllib2.urlopen`. Proxy handling, which was done by passing a dictionary parameter to `urllib.urlopen`, can be obtained by using `ProxyHandler` objects. Changed in version 3.2: `cafile` and `capath` were added.Changed in version 3.2:
HTTPS virtual hosts are now supported if possible (that is, if `ssl.HAS_SNI` is true). New in version 3.2: `data` can be an iterable object. Changed in version 3.3: `cadefault` was added.

```
urllib.request.install_opener(opener)
```

Install an OpenerDirector instance as the default global opener. Installing an opener is only necessary if you want urlopen to use that opener; otherwise, simply call `OpenerDirector.open()` instead of `urlopen()`. The code does not check for a real OpenerDirector, and any class with the appropriate interface will work.

```
urllib.request.build_opener([handler, ...])
```

Return an OpenerDirector instance, which chains the handlers in the order given. `handlers` can be either instances of `BaseHandler`, or subclasses of `BaseHandler` (in which case it must be possible to call the constructor without any parameters). Instances of the following classes will be in front of the `handlers`, unless the `handlers` contain them, instances of them or subclasses of them: `ProxyHandler` (if proxy settings are detected), `UnknownHandler`, `HTTPHandler`, `HTTPDefaultErrorHandler`, `HTTPRedirectHandler`, `FTPHandler`, `FileHandler`, `HTTPErrorProcessor`.

If the Python installation has SSL support (i.e., if the `ssl` module can be imported), `HTTPSHandler` will also be added.

A `BaseHandler` subclass may also change its `handler_order` attribute to modify its position in the handlers list.

```
urllib.request.pathname2url(path)
```

Convert the pathname `path` from the local syntax for a path to the form used in the path component of a URL. This does not produce a complete URL. The return value will already be quoted using the `quote()` function.

```
urllib.request.url2pathname(path)
```

Convert the path component `path` from a percent-encoded URL to the local syntax for a path. This does not accept a complete URL. This function uses `unquote()` to decode `path`.

```
urllib.request.getproxies()
```

This helper function returns a dictionary of scheme to proxy server URL mappings. It scans the environment for variables named `<scheme>_proxy`, in a case insensitive approach, for all operating systems first, and when it cannot find it, looks for proxy information from Mac OSX System Configuration for Mac OS X and Windows Systems Registry for Windows.

The following classes are provided:

```
class urllib.request.Request(url, data=None, headers={}, origin_req_host=None, unverifiable=False, method=None)
```

This class is an abstraction of a URL request.

`url` should be a string containing a valid URL.

`data` must be a bytes object specifying additional data to send to the server, or `None` if no such data is needed. Currently HTTP requests are the only ones that use `data`; the HTTP request will be a POST instead of a GET when the `data` parameter is provided. `data` should be a buffer in the standard `application/x-www-form-urlencoded` format.

The `urllib.parse.urlencode()` function takes a mapping or sequence of 2-tuples and returns a string in this format. It should be encoded to bytes before being used as the `data` parameter. The charset parameter in Content-Type header may be used to specify the encoding. If charset parameter is not sent with the Content-Type header, the server following the HTTP 1.1 recommendation may assume that the data is encoded in ISO-8859-1 encoding. It is advisable to use charset parameter with encoding used in Content-Type header with the `Request`.

`headers` should be a dictionary, and will be treated as if `add_header()` was called with each key and value as arguments. This is often used to “spooof” the User-Agent header, which is used by a browser to identify itself – some HTTP servers only allow requests coming from common browsers as opposed to scripts. For example, Mozilla Firefox may identify itself as "Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11", while urllib's default user agent string is "Python-urllib/2.6" (on Python 2.6).
An example of using Content-Type header with data argument would be sending a dictionary like
{
"Content-Type":" application/x-www-form-urlencoded;charset=utf-8"
}

The final two arguments are only of interest for correct handling of third-party HTTP cookies:

- origin_req_host should be the request-host of the origin transaction, as defined by RFC 2965. It defaults to http.cookiejar.request_host(self). This is the host name or IP address of the original request that was initiated by the user. For example, if the request is for an image in an HTML document, this should be the request-host of the request for the page containing the image.

- unverifiable should indicate whether the request is unverifiable, as defined by RFC 2965. It defaults to False. An unverifiable request is one whose URL the user did not have the option to approve. For example, if the request is for an image in an HTML document, and the user had no option to approve the automatic fetching of the image, this should be true.

- method should be a string that indicates the HTTP request method that will be used (e.g. ‘HEAD’). Its value is stored in the method attribute and is used by get_method(). Changed in version 3.3: Request.method argument is added to the Request class.

```
class urllib.request.OpenerDirector
    The OpenerDirector class opens URLs via BaseHandlers chained together. It manages the chaining of handlers, and recovery from errors.

class urllib.request.BaseHandler
    This is the base class for all registered handlers — and handles only the simple mechanics of registration.

class urllib.request.HTTPDefaultErrorHandler
    A class which defines a default handler for HTTP error responses; all responses are turned into HTTPError exceptions.

class urllib.request.HTTPRedirectHandler
    A class to handle redirections.

class urllib.request.HTTPCookieProcessor(cookiejar=None)
    A class to handle HTTP Cookies.

class urllib.request.ProxyHandler(proxies=None)
    Cause requests to go through a proxy. If proxies is given, it must be a dictionary mapping protocol names to URLs of proxies. The default is to read the list of proxies from the environment variables

    <protocol>_proxy. If no proxy environment variables are set, then in a Windows environment proxy settings are obtained from the registry’s Internet Settings section, and in a Mac OS X environment proxy information is retrieved from the OS X System Configuration Framework.

    To disable autodetected proxy pass an empty dictionary.

class urllib.request.HTTPPasswordMgr
    Keep a database of (realm, uri) -> (user, password) mappings.

class urllib.request.HTTPPasswordMgrWithDefaultRealm
    Keep a database of (realm, uri) -> (user, password) mappings. A realm of None is considered a catch-all realm, which is searched if no other realm fits.

class urllib.request(AbstractBasicAuthHandler(password_mgr=None)
    This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported.

class urllib.request.HTTPBasicAuthHandler(password_mgr=None)
    Handle authentication with the remote host. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported. HTTPBasicAuthHandler will raise a ValueError when presented with a wrong Authentication scheme.

class urllib.request.ProxyBasicAuthHandler(password_mgr=None)
    Handle authentication with the proxy. password_mgr, if given, should be something that is compatible with
HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported.

class urllib.request.AbstractDigestAuthHandler (password_mgr=None)
This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported.

class urllib.request.HTTPDigestAuthHandler (password_mgr=None)
Handle authentication with the remote host. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported. When both Digest Authentication Handler and Basic Authentication Handler are both added, Digest Authentication is always tried first. If the Digest Authentication returns a 40x response again, it is sent to Basic Authentication handler to Handle. This Handler method will raise a ValueError when presented with an authentication scheme other than Digest or Basic. Changed in version 3.3: Raise ValueError on unsupported Authentication Scheme.

class urllib.request.ProxyDigestAuthHandler (password_mgr=None)
Handle authentication with the proxy. password_mgr, if given, should be something that is compatible with HTTPPasswordMgr; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported.

class urllib.request.HTTPHandler
A class to handle opening of HTTP URLs.

class urllib.request.HTTPSHandler (debuglevel=0, context=None, check_hostname=None)
A class to handle opening of HTTPS URLs. context and check_hostname have the same meaning as in http.client.HTTPSConnection. Changed in version 3.2: context and check_hostname were added.

class urllib.request.FileHandler
Open local files.

class urllib.request.FTPHandler
Open FTP URLs.

class urllib.request.CacheFTPHandler
Open FTP URLs, keeping a cache of open FTP connections to minimize delays.

class urllib.request.UnknownHandler
A catch-all class to handle unknown URLs.

class urllib.request.HTTPErrorProcessor
Process HTTP error responses.

21.6.1 Request Objects

The following methods describe Request’s public interface, and so all may be overridden in subclasses. It also defines several public attributes that can be used by clients to inspect the parsed request.

Request.full_url
The original URL passed to the constructor.

Request.type
The URI scheme.

Request.host
The URI authority, typically a host, but may also contain a port separated by a colon.

Request.origin_req_host
The original host for the request, without port.

Request.selector
The URI path. If the Request uses a proxy, then selector will be the full url that is passed to the proxy.
Request.data
  The entity body for the request, or None if not specified.

Request.unverifiable
  boolean, indicates whether the request is unverifiable as defined by RFC 2965.

Request.method
  The HTTP request method to use. This value is used by get_method() to override the computed HTTP request method that would otherwise be returned. This attribute is initialized with the value of the method argument passed to the constructor. New in version 3.3.

Request.get_method()
  Return a string indicating the HTTP request method. If Request.method is not None, return its value, otherwise return 'GET' if Request.data is None, or 'POST' if it's not. This is only meaningful for HTTP requests. Changed in version 3.3: get_method now looks at the value of Request.method.

Request.add_header(key, val)
  Add another header to the request. Headers are currently ignored by all handlers except HTTP handlers, where they are added to the list of headers sent to the server. Note that there cannot be more than one header with the same name, and later calls will overwrite previous calls in case the key collides. Currently, this is no loss of HTTP functionality, since all headers which have meaning when used more than once have a (header-specific) way of gaining the same functionality using only one header.

Request.add_unredirected_header(key, header)
  Add a header that will not be added to a redirected request.

Request.has_header(header)
  Return whether the instance has the named header (checks both regular and unredirected).

Request.get_full_url()
  Return the URL given in the constructor.

Request.set_proxy(host, type)
  Prepare the request by connecting to a proxy server. The host and type will replace those of the instance, and the instance’s selector will be the original URL given in the constructor.

Request.add_data(data)
  Set the Request data to data. This is ignored by all handlers except HTTP handlers — and there it should be a byte string, and will change the request to be POST rather than GET. Deprecated in 3.3, use Request.data. Deprecated since version 3.3, will be removed in version 3.4.

Request.has_data()
  Return whether the instance has a non-None data. Deprecated in 3.3, use Request.data. Deprecated since version 3.3, will be removed in version 3.4.

Request.get_data()
  Return the instance’s data. Deprecated in 3.3, use Request.data. Deprecated since version 3.3, will be removed in version 3.4.

Request.get_type()
  Return the type of the URL — also known as the scheme. Deprecated in 3.3, use Request.type. Deprecated since version 3.3, will be removed in version 3.4.

Request.get_host()
  Return the host to which a connection will be made. Deprecated in 3.3, use Request.host. Deprecated since version 3.3, will be removed in version 3.4.

Request.get_selector()
  Return the selector — the part of the URL that is sent to the server. Deprecated in 3.3, use Request.selector. Deprecated since version 3.3, will be removed in version 3.4.

Request.get_header(header_name, default=None)
  Return the value of the given header. If the header is not present, return the default value.

Request.header_items()
  Return a list of tuples (header_name, header_value) of the Request headers.
The Python Library Reference, Release 3.3.3

Project.set_proxy (host, type)

Request.get_origin_req_host ()

Return the request-host of the origin transaction, as defined by

RFC 2965. See the documentation for the Request constructor. Deprecated in 3.3, use Request.origin_req_host. Deprecated since version 3.3, will be removed in version 3.4.

Request.is_unverifiable ()

Return whether the request is unverifiable, as defined by RFC 2965. See the documentation for the Request constructor. Deprecated in 3.3, use Request.unverifiable. Deprecated since version 3.3, will be removed in version 3.4.

21.6.2 OpenerDirector Objects

OpenerDirector instances have the following methods:

OpenerDirector.add_handler (handler)

handler should be an instance of BaseHandler. The following methods are searched, and added to the possible chains (note that HTTP errors are a special case).

• protocol_open () — signal that the handler knows how to open protocol URLs.

• http_error_type () — signal that the handler knows how to handle HTTP errors with HTTP error code type.

• protocol_error () — signal that the handler knows how to handle errors from (non-http) protocol.

• protocol_request () — signal that the handler knows how to pre-process protocol requests.

• protocol_response () — signal that the handler knows how to post-process protocol responses.

OpenerDirector.open (url, data=None, timeout)

Open the given url (which can be a request object or a string), optionally passing the given data. Arguments, return values and exceptions raised are the same as those of urlopen () (which simply calls the open () method on the currently installed global OpenerDirector). The optional timeout parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). The timeout feature actually works only for HTTP, HTTPS and FTP connections.

OpenerDirector.error (proto, *args)

Handle an error of the given protocol. This will call the registered error handlers for the given protocol with the given arguments (which are protocol specific). The HTTP protocol is a special case which uses the HTTP response code to determine the specific error handler; refer to the http_error_* () methods of the handler classes.

Return values and exceptions raised are the same as those of urlopen().

OpenerDirector objects open URLs in three stages:

The order in which these methods are called within each stage is determined by sorting the handler instances.

1. Every handler with a method named like protocol_request () has that method called to pre-process the request.

2. Handlers with a method named like protocol_open () are called to handle the request. This stage ends when a handler either returns a non-None value (i.e. a response), or raises an exception (usually URLError). Exceptions are allowed to propagate.

In fact, the above algorithm is first tried for methods named default_open (). If all such methods return None , the algorithm is repeated for methods named like protocol_open (). If all such methods return None , the algorithm is repeated for methods named unknown_open ()

Note that the implementation of these methods may involve calls of the parent OpenerDirector instance’s open () and error () methods.
3. Every handler with a method named like `protocol_response()` has that method called to post-process the response.

### 21.6.3 BaseHandler Objects

`BaseHandler` objects provide a couple of methods that are directly useful, and others that are meant to be used by derived classes. These are intended for direct use:

- **BaseHandler.add_parent**(director)
  
  Add a director as parent.

- **BaseHandler.close()
  
  Remove any parents.

The following attribute and methods should only be used by classes derived from `BaseHandler`.

---

**Note:** The convention has been adopted that subclasses defining `protocol_request()` or `protocol_response()` methods are named *Processor; all others are named *Handler.

- **BaseHandler.parent
  
  A valid `OpenerDirector`, which can be used to open using a different protocol, or handle errors.

- **BaseHandler.default_open**(req)
  
  This method is not defined in `BaseHandler`, but subclasses should define it if they want to catch all URLs.

  This method, if implemented, will be called by the parent `OpenerDirector`. It should return a file-like object as described in the return value of the `open()` of `OpenerDirector`, or None. It should raise `URLError`, unless a truly exceptional thing happens (for example, `MemoryError` should not be mapped to `URLError`).

  This method will be called before any protocol-specific open method.

- **BaseHandler.protocol_open**(req)
  
  This method is not defined in `BaseHandler`, but subclasses should define it if they want to handle URLs with the given protocol.

  This method, if defined, will be called by the parent `OpenerDirector`. Return values should be the same as for `default_open()`.

- **BaseHandler.unknown_open**(req)
  
  This method is not defined in `BaseHandler`, but subclasses should define it if they want to catch all URLs with no specific registered handler to open it.

  This method, if implemented, will be called by the parent `OpenerDirector`. Return values should be the same as for `default_open()`.

- **BaseHandler.http_error_default**(req, fp, code, msg, hdrs)
  
  This method is not defined in `BaseHandler`, but subclasses should override it if they intend to provide a catch-all for otherwise unhandled HTTP errors. It will be called automatically by the `OpenerDirector` getting the error, and should not normally be called in other circumstances.

  `req` will be a `Request` object, `fp` will be a file-like object with the HTTP error body, `code` will be the three-digit code of the error, `msg` will be the user-visible explanation of the code and `hdrs` will be a mapping object with the headers of the error.

  Return values and exceptions raised should be the same as those of `urlopen()`.

- **BaseHandler.http_error_nnn**(req, fp, code, msg, hdrs)
  
  `nnn` should be a three-digit HTTP error code. This method is also not defined in `BaseHandler`, but will be called, if it exists, on an instance of a subclass, when an HTTP error with code `nnn` occurs.

  Subclasses should override this method to handle specific HTTP errors.

  Arguments, return values and exceptions raised should be the same as for `http_error_default()`.
BaseHandler.prototype_request\( (req) \)
This method is not defined in BaseHandler, but subclasses should define it if they want to pre-process requests of the given protocol.

This method, if defined, will be called by the parent OpenerDirector. \( req \) will be a Request object.
The return value should be a Request object.

BaseHandler.prototype_response\( (req, response) \)
This method is not defined in BaseHandler, but subclasses should define it if they want to post-process responses of the given protocol.

This method, if defined, will be called by the parent OpenerDirector. \( req \) will be a Request object.
response will be an object implementing the same interface as the return value of urlopen(). The return value should implement the same interface as the return value of urlopen().

### 21.6.4 HTTPRedirectHandler Objects

**Note:** Some HTTP redirections require action from this module’s client code. If this is the case, HTTPError is raised. See RFC 2616 for details of the precise meanings of the various redirection codes.

An HTTPError exception raised as a security consideration if the HTTPRedirectHandler is presented with a redirected url which is not an HTTP, HTTPS or FTP url.

HTTPRedirectHandler.prototype.redirect_request\( (req, fp, code, msg, hdrs, newurl) \)
Return a Request or None in response to a redirect. This is called by the default implementations of the http-error-30\*() methods when a redirection is received from the server. If a redirection should take place, return a new Request to allow http-error-30\*() to perform the redirect to newurl. Otherwise, raise HTTPError if no other handler should try to handle this URL, or return None if you can’t but another handler might.

**Note:** The default implementation of this method does not strictly follow RFC 2616, which says that 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and the default implementation reproduces this behavior.

HTTPRedirectHandler.prototype.http_error_301\( (req, fp, code, msg, hdrs) \)
Redirect to the Location: or URI: URL. This method is called by the parent OpenerDirector when getting an HTTP 'moved permanently' response.

HTTPRedirectHandler.prototype.http_error_302\( (req, fp, code, msg, hdrs) \)
The same as http-error-301(), but called for the 'found' response.

HTTPRedirectHandler.prototype.http_error_303\( (req, fp, code, msg, hdrs) \)
The same as http-error-301(), but called for the 'see other' response.

HTTPRedirectHandler.prototype.http_error_307\( (req, fp, code, msg, hdrs) \)
The same as http-error-301(), but called for the ‘temporary redirect’ response.

### 21.6.5 HTTPCookieProcessor Objects

HTTPCookieProcessor instances have one attribute:

HTTPCookieProcessor.prototype.cookiejar
The http.cookiejar.CookieJar in which cookies are stored.
21.6.6 ProxyHandler Objects

ProxyHandler.\texttt{protocol\_open}(\texttt{request})

The \texttt{ProxyHandler} will have a method \texttt{protocol\_open()} for every \texttt{protocol} which has a proxy in the \texttt{proxies} dictionary given in the constructor. The method will modify requests to go through the proxy, by calling \texttt{request.set\_proxy()}, and call the next handler in the chain to actually execute the protocol.

21.6.7 HTTPPasswordMgr Objects

These methods are available on \texttt{HTTPPasswordMgr} and \texttt{HTTPPasswordMgrWithDefaultRealm} objects.

\texttt{HTTPPasswordMgr.add\_password}(\texttt{realm, uri, user, passwd})

\texttt{uri} can be either a single URI, or a sequence of URIs. \texttt{realm, user} and \texttt{passwd} must be strings. This causes \texttt{(user, passwd)} to be used as authentication tokens when authentication for \texttt{realm} and a super-URI of any of the given URIs is given.

\texttt{HTTPPasswordMgr.find\_user\_password}(\texttt{realm, authuri})

Get user/password for given \texttt{realm} and \texttt{URI}, if any. This method will return \texttt{(None, None)} if there is no matching user/password.

For \texttt{HTTPPasswordMgrWithDefaultRealm} objects, the \texttt{realm None} will be searched if the given \texttt{realm} has no matching user/password.

21.6.8 AbstractBasicAuthHandler Objects

AbstractBasicAuthHandler.\texttt{http\_error\_auth\_reqed}(\texttt{authreq, host, req, headers})

Handle an authentication request by getting a user/password pair, and re-trying the request. \texttt{authreq} should be the name of the header where the information about the realm is included in the request, \texttt{host} specifies the URL and path to authenticate for, \texttt{req} should be the (failed) \texttt{Request} object, and \texttt{headers} should be the error headers.

\texttt{host} is either an authority (e.g. "python.org") or a URL containing an authority component (e.g. "http://python.org/"). In either case, the authority must not contain a userinfo component (so, "python.org" and "python.org:80" are fine, "joe:password@python.org" is not).

21.6.9 HTTPBasicAuthHandler Objects

HTTPBasicAuthHandler.\texttt{http\_error\_401}(\texttt{req, fp, code, msg, hdrs})

Retry the request with authentication information, if available.

21.6.10 ProxyBasicAuthHandler Objects

ProxyBasicAuthHandler.\texttt{http\_error\_407}(\texttt{req, fp, code, msg, hdrs})

Retry the request with authentication information, if available.

21.6.11 AbstractDigestAuthHandler Objects

AbstractDigestAuthHandler.\texttt{http\_error\_auth\_reqed}(\texttt{authreq, host, req, headers})

\texttt{authreq} should be the name of the header where the information about the realm is included in the request, \texttt{host} should be the host to authenticate to, \texttt{req} should be the (failed) \texttt{Request} object, and \texttt{headers} should be the error headers.
21.6.12 HTTPDigestAuthHandler Objects

HTTPDigestAuthHandler.http_error_401(req, fp, code, msg, hdrs)
Retry the request with authentication information, if available.

21.6.13 ProxyDigestAuthHandler Objects

ProxyDigestAuthHandler.http_error_407(req, fp, code, msg, hdrs)
Retry the request with authentication information, if available.

21.6.14 HTTPHandler Objects

HTTPHandler.http_open(req)
Send an HTTP request, which can be either GET or POST, depending on req.has_data().

21.6.15 HTTPSHandler Objects

HTTPSHandler.https_open(req)
Send an HTTPS request, which can be either GET or POST, depending on req.has_data().

21.6.16 FileHandler Objects

FileHandler.file_open(req)
Open the file locally, if there is no host name, or the host name is ‘localhost’. Changed in version 3.2: This method is applicable only for local hostnames. When a remote hostname is given, an URLError is raised.

21.6.17 FTPHandler Objects

FTPHandler.ftp_open(req)
Open the FTP file indicated by req. The login is always done with empty username and password.

21.6.18 CacheFTPHandler Objects

CacheFTPHandler objects are FTPHandler objects with the following additional methods:
CacheFTPHandler.set_timeout(t)
Set timeout of connections to t seconds.
CacheFTPHandler.set_max_conns(m)
Set maximum number of cached connections to m.

21.6.19 UnknownHandler Objects

UnknownHandler.unknown_open()
Raise a URLError exception.
The Python Library Reference, Release 3.3.3

21.6.20 HTTPErrorProcessor Objects
HTTPErrorProcessor.http_response()
Process HTTP error responses.
For 200 error codes, the response object is returned immediately.
For non-200 error codes, this simply passes the job on to the protocol_error_code() handler
methods, via OpenerDirector.error(). Eventually, HTTPDefaultErrorHandler will raise
an HTTPError if no other handler handles the error.
HTTPErrorProcessor.https_response()
Process HTTPS error responses.
The behavior is same as http_response().

21.6.21 Examples
This example gets the python.org main page and displays the first 300 bytes of it.
>>> import urllib.request
>>> f = urllib.request.urlopen(’http://www.python.org/’)
>>> print(f.read(300))
b’<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">\n\n\n<html
xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">\n\n<head>\n
<meta http-equiv="content-type" content="text/html; charset=utf-8" />\n
<title>Python Programming ’
Note that urlopen returns a bytes object. This is because there is no way for urlopen to automatically determine
the encoding of the byte stream it receives from the http server. In general, a program will decode the returned
bytes object to string once it determines or guesses the appropriate encoding.
The following W3C document, http://www.w3.org/International/O-charset , lists the various ways in which a
(X)HTML or a XML document could have specified its encoding information.
As the python.org website uses utf-8 encoding as specified in it’s meta tag, we will use the same for decoding the
bytes object.
>>> with urllib.request.urlopen(’http://www.python.org/’) as f:
...
print(f.read(100).decode(’utf-8’))
...
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
It is also possible to achieve the same result without using the context manager approach.
>>> import urllib.request
>>> f = urllib.request.urlopen(’http://www.python.org/’)
>>> print(f.read(100).decode(’utf-8’))
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
In the following example, we are sending a data-stream to the stdin of a CGI and reading the data it returns to us.
Note that this example will only work when the Python installation supports SSL.
>>>
>>>
...
>>>
>>>
Got

import urllib.request
req = urllib.request.Request(url=’https://localhost/cgi-bin/test.cgi’,
data=b’This data is passed to stdin of the CGI’)
f = urllib.request.urlopen(req)
print(f.read().decode(’utf-8’))
Data: "This data is passed to stdin of the CGI"

The code for the sample CGI used in the above example is:

21.6. urllib.request — Extensible library for opening URLs

873


#!/usr/bin/env python
import sys
data = sys.stdin.read()
print('Content-type: text/plain

Got Data: "%s"% data)

Here is an example of doing a PUT request using Request:

import urllib.request
DATA=b'some data'
req = urllib.request.Request(url='http://localhost:8080', data=DATA, method='PUT')
f = urllib.request.urlopen(req)
print(f.status)
print(f.reason)

Use of Basic HTTP Authentication:

import urllib.request
# Create an OpenerDirector with support for Basic HTTP Authentication...
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
uri='https://mahler:8092/site-updates.py',
user='klem',
passwd='kadidd!ehopper')

opener = urllib.request.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib.request.install_opener(opener)
urllib.request.urlopen('http://www.example.com/login.html')

build_opener() provides many handlers by default, including a ProxyHandler. By default, ProxyHandler uses the environment variables named <scheme>_proxy, where <scheme> is the URL scheme involved. For example, the http_proxy environment variable is read to obtain the HTTP proxy’s URL.

This example replaces the default ProxyHandler with one that uses programmatically-supplied proxy URLs, and adds proxy authorization support with ProxyBasicAuthHandler.

proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib.request.build_opener(proxy_handler, proxy_auth_handler)
# This time, rather than install the OpenerDirector, we use it directly:
opener.open('http://www.example.com/login.html')

Adding HTTP headers:

Use the headers argument to the Request constructor, or:

import urllib.request
req = urllib.request.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
r = urllib.request.urlopen(req)

OpenerDirector automatically adds a User-Agent header to every Request. To change this:

import urllib.request
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/')

Also, remember that a few standard headers (Content-Length, Content-Type without charset parameter and Host) are added when the Request is passed to urlopen() (or OpenerDirector.open()). Here is an example session that uses the GET method to retrieve a URL containing parameters:

>>> import urllib.request
>>> import urllib.parse
```python
>>> params = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> f = urllib.request.urlopen("http://www.musical.com/cgi-bin/query?\%s" % params)
>>> print(f.read().decode('utf-8'))

The following example uses the POST method instead. Note that params output from urlencode is encoded to bytes before it is sent to urlopen as data:

```python
>>> import urllib.request
>>> import urllib.parse

```python
>>> data = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> data = data.encode('utf-8')
>>> request = urllib.request.Request("http://requestb.in/xrbl82xr")
>>> # adding charset parameter to the Content-Type header.
>>> request.add_header("Content-Type","application/x-www-form-urlencoded;charset=utf-8")
>>> f = urllib.request.urlopen(request, data)
>>> print(f.read().decode('utf-8'))

The following example uses an explicitly specified HTTP proxy, overriding environment settings:

```python
>>> import urllib.request

```python
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.request.FancyURLopener(proxies)
>>> f = opener.open("http://www.python.org")
>>> f.read().decode('utf-8')

The following example uses no proxies at all, overriding environment settings:

```python
>>> import urllib.request

```python
>>> opener = urllib.request.FancyURLopener({})
>>> f = opener.open("http://www.python.org")
>>> f.read().decode('utf-8')

### 21.6.22 Legacy interface

The following functions and classes are ported from the Python 2 module `urllib` (as opposed to `urllib2`). They might become deprecated at some point in the future.

`urllib.request.urlretrieve(url, filename=None, reporthook=None, data=None)`

Copy a network object denoted by a URL to a local file. If the URL points to a local file, the object will not be copied unless `filename` is supplied. Return a tuple (file name, headers) where `filename` is the local file name under which the object can be found, and `headers` is whatever the `info()` method of the object returned by `urlopen()` returned (for a remote object). Exceptions are the same as for `urlopen()`.

The second argument, if present, specifies the file location to copy to (if absent, the location will be a temporary file with a generated name). The third argument, if present, is a hook function that will be called once on establishment of the network connection and once after each block read thereafter. The hook will be passed three arguments; a count of blocks transferred so far, a block size in bytes, and the total size of the file. The third argument may be `-1` on older FTP servers which do not return a file size in response to a retrieval request.

The following example illustrates the most common usage scenario:

```python
>>> import urllib.request

```python
>>> local_filename, headers = urllib.request.urlretrieve('http://python.org/')
>>> html = open(local_filename)
>>> html.close()

If the `url` uses the `http:` scheme identifier, the optional `data` argument may be given to specify a POST request (normally the request type is GET). The `data` argument must be a bytes object in standard `application/x-www-form-urlencoded` format; see the `urllib.parse.urlencode()` function.
urlretrieve() will raise `ContentTooShortError` when it detects that the amount of data available was less than the expected amount (which is the size reported by a `Content-Length` header). This can occur, for example, when the download is interrupted.

The `Content-Length` is treated as a lower bound: if there’s more data to read, urlretrieve reads more data, but if less data is available, it raises the exception.

You can still retrieve the downloaded data in this case, it is stored in the `content` attribute of the exception instance.

If no `Content-Length` header was supplied, urlretrieve can not check the size of the data it has downloaded, and just returns it. In this case you just have to assume that the download was successful.

```
urlib.request.urlcleanup()
```

Cleans up temporary files that may have been left behind by previous calls to `urlretrieve()`.

```
class urllib.request.URLopener (proxies=None, **x509)
```

Deprecated since version 3.3. Base class for opening and reading URLs. Unless you need to support opening objects using schemes other than `http:`, `ftp:`, or `file:`, you probably want to use `FancyURLopener`.

By default, the `URLopener` class sends a `User-Agent` header of `urllib/VVV`, where `VVV` is the `urllib` version number. Applications can define their own `User-Agent` header by subclassing `URLopener` or `FancyURLopener` and setting the class attribute `version` to an appropriate string value in the subclass definition.

The optional `proxies` parameter should be a dictionary mapping scheme names to proxy URLs, where an empty dictionary turns proxies off completely. Its default value is `None`, in which case environmental proxy settings will be used if present, as discussed in the definition of `urlopen()`, above.

Additional keyword parameters, collected in `x509`, may be used for authentication of the client when using the `https:` scheme. The keywords `key_file` and `cert_file` are supported to provide an SSL key and certificate; both are needed to support client authentication.

`URLopener` objects will raise an `OSError` exception if the server returns an error code.

```
open (fullurl, data=None)
```

Open `fullurl` using the appropriate protocol. This method sets up cache and proxy information, then calls the appropriate open method with its input arguments. If the scheme is not recognized, `open_unknown()` is called. The `data` argument has the same meaning as the `data` argument of `urlopen()`.

```
open_unknown (fullurl, data=None)
```

Overridable interface to open unknown URL types.

```
retrieve (url, filename=None, reporthook=None, data=None)
```

Retrieves the contents of `url` and places it in `filename`. The return value is a tuple consisting of a local filename and either a `email.message.Message` object containing the response headers (for remote URLs) or `None` (for local URLs). The caller must then open and read the contents of `filename`. If `filename` is not given and the URL refers to a local file, the input filename is returned. If the URL is non-local and `filename` is not given, the filename is the output of `tempfile.mktemp()` with a suffix that matches the suffix of the last path component of the input URL. If `reporthook` is given, it must be a function accepting three numeric parameters: A chunk number, the maximum size chunks are read in and the total size of the download (-1 if unknown). It will be called once at the start and after each chunk of data is read from the network. `reporthook` is ignored for local URLs.

If the `url` uses the `http:` scheme identifier, the optional `data` argument may be given to specify a `POST` request (normally the request type is `GET`). The `data` argument must in standard `application/x-www-form-urlencoded` format; see the `urllib.parse.urlencode()` function.

```
version
```

Variable that specifies the user agent of the opener object. To get `urllib` to tell servers that it is a particular user agent, set this in a subclass as a class variable or in the constructor before calling the base constructor.
class urllib.request.FancyURLopener(...)

Deprecated since version 3.3. FancyURLopener subclasses URLopener providing default handling for the following HTTP response codes: 301, 302, 303, 307 and 401. For the 30x response codes listed above, the Location header is used to fetch the actual URL. For 401 response codes (authentication required), basic HTTP authentication is performed. For the 30x response codes, recursion is bounded by the value of the maxtries attribute, which defaults to 10.

For all other response codes, the method http_error_default() is called which you can override in subclasses to handle the error appropriately.

Note: According to the letter of RFC 2616, 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and urllib reproduces this behaviour.

The parameters to the constructor are the same as those for URLopener.

Note: When performing basic authentication, a FancyURLopener instance calls its prompt_user_passwd() method. The default implementation asks the users for the required information on the controlling terminal. A subclass may override this method to support more appropriate behavior if needed.

The FancyURLopener class offers one additional method that should be overloaded to provide the appropriate behavior:

prompt_user_passwd(host, realm)

Return information needed to authenticate the user at the given host in the specified security realm. The return value should be a tuple, (user, password), which can be used for basic authentication.

The implementation prompts for this information on the terminal; an application should override this method to use an appropriate interaction model in the local environment.

21.6.23 urllib.request Restrictions

- Currently, only the following protocols are supported: HTTP (versions 0.9 and 1.0), FTP, and local files.
- The caching feature of urlretrieve() has been disabled until someone finds the time to hack proper processing of Expiration time headers.
- There should be a function to query whether a particular URL is in the cache.
- For backward compatibility, if a URL appears to point to a local file but the file can’t be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.
- The urlopen() and urlretrieve() functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive Web client using these functions without using threads.
- The data returned by urlopen() or urlretrieve() is the raw data returned by the server. This may be binary data (such as an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the Content-Type header. If the returned data is HTML, you can use the module html.parser to parse it.
- The code handling the FTP protocol cannot differentiate between a file and a directory. This can lead to unexpected behavior when attempting to read a URL that points to a file that is not accessible. If the URL ends in a /, it is assumed to refer to a directory and will be handled accordingly. But if an attempt to read a file leads to a 550 error (meaning the URL cannot be found or is not accessible, often for permission reasons), then the path is treated as a directory in order to handle the case when a directory is specified by a URL but the trailing / has been left off. This can cause misleading results when you try to fetch a file whose read permissions make it inaccessible; the FTP code will try to read it, fail with a 550 error, and
then perform a directory listing for the unreadable file. If fine-grained control is needed, consider using the
\texttt{ftplib} module, subclassing \texttt{FancyURLopener}, or changing \texttt{urloper}
to meet your needs.

### 21.7 \texttt{urllib.response} — Response classes used by \texttt{urllib}

The \texttt{urllib.response} module defines functions and classes which define a minimal file like interface, including \texttt{read()} and \texttt{readline()}. The typical response object is an \texttt{addinfourl} instance, which defines an \texttt{info()} method and that returns headers and a \texttt{geturl()} method that returns the url. Functions defined by this module are used internally by the \texttt{urllib.request} module.

### 21.8 \texttt{urllib.parse} — Parse URLs into components

Source code: Lib/urllib/parse.py

This module defines a standard interface to break Uniform Resource Locator (URL) strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a “relative URL” to an absolute URL given a “base URL.”

The module has been designed to match the Internet RFC on Relative Uniform Resource Locators. It supports the following URL schemes: file, ftp, gopher, hdl, http, https, imap, mailto, mms, news, nntp, prospero, rsync, rtsp, rtsps, sftp, shttp, sip, sips, snews, svn, svn+ssh, telnet, wais.

The \texttt{urllib.parse} module defines functions that fall into two broad categories: URL parsing and URL quoting. These are covered in detail in the following sections.

#### 21.8.1 URL Parsing

The URL parsing functions focus on splitting a URL string into its components, or on combining URL components into a URL string.

\texttt{urllib.parse.urlencoded}(*urlstring*, *scheme=’’, *allow_fragments=True*)

Parse a URL into six components, returning a 6-tuple. This corresponds to the general structure of a URL: \texttt{scheme://netloc/path;parameters?query#fragment}. Each tuple item is a string, possibly empty. The components are not broken up in smaller parts (for example, the network location is a single string), and \% escapes are not expanded. The delimiters as shown above are not part of the result, except for a leading slash in the path component, which is retained if present. For example:

```python
>>> from urllib.parse import urlparse
>>> o = urlparse(’http://www.cwi.nl:80/%7Eguido/Python.html’)
>>> o
ParseResult(scheme=’http’, netloc=’www.cwi.nl:80’, path=’/%7Eguido/Python.html’,
 params=’’, query=’’, fragment=’’)
>>> o.scheme
’http’
>>> o.port
80
>>> o.geturl()
’http://www.cwi.nl:80/%7Eguido/Python.html’
```

Following the syntax specifications in RFC 1808, \texttt{urloparse} recognizes a netloc only if it is properly introduced by ‘//’. Otherwise the input is presumed to be a relative URL and thus to start with a path component.

```python
>>> from urllib.parse import urlparse
>>> urlparse(’//www.cwi.nl:80/%7Eguido/Python.html’)
```
If the `scheme` argument is specified, it gives the default addressing scheme, to be used only if the URL does not specify one. The default value for this argument is the empty string.

If the `allow_fragments` argument is false, fragment identifiers are not allowed. The default value for this argument is `True`.

The return value is actually an instance of a subclass of `tuple`. This class has the following additional read-only convenience attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Index</th>
<th>Value</th>
<th>Value if not present</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheme</td>
<td>0</td>
<td>URL scheme specifier</td>
<td>empty string</td>
</tr>
<tr>
<td>netloc</td>
<td>1</td>
<td>Network location part</td>
<td>empty string</td>
</tr>
<tr>
<td>path</td>
<td>2</td>
<td>Hierarchical path</td>
<td>empty string</td>
</tr>
<tr>
<td>params</td>
<td>3</td>
<td>Parameters for last path element</td>
<td>empty string</td>
</tr>
<tr>
<td>query</td>
<td>4</td>
<td>Query component</td>
<td>empty string</td>
</tr>
<tr>
<td>fragment</td>
<td>5</td>
<td>Fragment identifier</td>
<td>empty string</td>
</tr>
<tr>
<td>username</td>
<td></td>
<td>User name</td>
<td>None</td>
</tr>
<tr>
<td>password</td>
<td></td>
<td>Password</td>
<td>None</td>
</tr>
<tr>
<td>hostname</td>
<td></td>
<td>Host name (lower case)</td>
<td>None</td>
</tr>
<tr>
<td>port</td>
<td></td>
<td>Port number as integer, if present</td>
<td>None</td>
</tr>
</tbody>
</table>

See section `Structured Parse Results` for more information on the result object. Changed in version 3.2: Added IPv6 URL parsing capabilities. Changed in version 3.3: The fragment is now parsed for all URL schemes (unless `allow_fragment` is false), in accordance with RFC 3986. Previously, a whitelist of schemes that support fragments existed.

```python
urllib.parse.parse_qs(qs, keep_blank_values=False, strict_parsing=False, encoding='utf-8', errors='replace')
```

Parse a query string given as a string argument (data of type `application/x-www-form-urlencoded`). Data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

The optional argument `keep_blank_values` is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument `strict_parsing` is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a `ValueError` exception.

The optional `encoding` and `errors` parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the `bytes.decode()` method.

Use the `urllib.parse.urlencode()` function (with the `doseq` parameter set to `True`) to convert such dictionaries into query strings. Changed in version 3.2: Add `encoding` and `errors` parameters.

```python
urllib.parse.parse_qsl(qs, keep_blank_values=False, strict_parsing=False, encoding='utf-8', errors='replace')
```

Parse a query string given as a string argument (data of type `application/x-www-form-urlencoded`). Data are returned as a list of name, value pairs.

The optional argument `keep_blank_values` is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.
The optional argument `strict_parsing` is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a `ValueError` exception.

The optional `encoding` and `errors` parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the `bytes.decode()` method.

Use the `urllib.parse.urlencode()` function to convert such lists of pairs into query strings. Changed in version 3.2: Add `encoding` and `errors` parameters.

```python
urllib.parse.urlparse(parts)
```

Construct a URL from a tuple as returned by `urlparse()`. The `parts` argument can be any six-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a ? with an empty query; the RFC states that these are equivalent).

```python
urllib.parse.urlsplit(urlstring, scheme='', allow_fragments=True)
```

This is similar to `urlparse()` but does not split the params from the URL. This should generally be used instead of `urlparse()` if the more recent URL syntax allowing parameters to be applied to each segment of the path portion of the URL (see RFC 2396) is wanted. A separate function is needed to separate the path segments and parameters. This function returns a 5-tuple: (addressing scheme, network location, path, query, fragment identifier).

The return value is actually an instance of a subclass of `tuple`. This class has the following additional read-only convenience attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Index</th>
<th>Value</th>
<th>Value if not present</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheme</td>
<td>0</td>
<td>URL scheme specifier</td>
<td>empty string</td>
</tr>
<tr>
<td>netloc</td>
<td>1</td>
<td>Network location part</td>
<td>empty string</td>
</tr>
<tr>
<td>path</td>
<td>2</td>
<td>Hierarchical path</td>
<td>empty string</td>
</tr>
<tr>
<td>query</td>
<td>3</td>
<td>Query component</td>
<td>empty string</td>
</tr>
<tr>
<td>fragment</td>
<td>4</td>
<td>Fragment identifier</td>
<td>empty string</td>
</tr>
<tr>
<td>username</td>
<td></td>
<td>User name</td>
<td>None</td>
</tr>
<tr>
<td>password</td>
<td></td>
<td>Password</td>
<td>None</td>
</tr>
<tr>
<td>hostname</td>
<td></td>
<td>Host name (lower case)</td>
<td>None</td>
</tr>
<tr>
<td>port</td>
<td></td>
<td>Port number as integer, if present</td>
<td>None</td>
</tr>
</tbody>
</table>

See section *Structured Parse Results* for more information on the result object.

```python
urllib.parse.urlunsplit(parts)
```

Combine the elements of a tuple as returned by `urlsplit()` into a complete URL as a string. The `parts` argument can be any five-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a ? with an empty query; the RFC states that these are equivalent).

```python
urllib.parse.urljoin(base, url, allow_fragments=True)
```

Construct a full (“absolute”) URL by combining a “base URL” (`base`) with another URL (`url`). Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL. For example:

```python
>>> from urllib.parse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

The `allow_fragments` argument has the same meaning and default as for `urlparse()`.

**Note:** If `url` is an absolute URL (that is, starting with // or `scheme://`), the `url`’s host name and/or scheme will be present in the result. For example:

```python
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
          ...   'http://www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```
If you do not want that behavior, preprocess the `url` with `urlsplit()` and `urlunsplit()`, removing possible `scheme` and `netloc` parts.

```
urllib.parse.urldefrag(url)
```

If `url` contains a fragment identifier, return a modified version of `url` with no fragment identifier, and the fragment identifier as a separate string. If there is no fragment identifier in `url`, return `url` unmodified and an empty string.

The return value is actually an instance of a subclass of `tuple`. This class has the following additional read-only convenience attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Index</th>
<th>Value</th>
<th>Value if not present</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>0</td>
<td>URL with no fragment</td>
<td>empty string</td>
</tr>
<tr>
<td>fragment</td>
<td>1</td>
<td>Fragment identifier</td>
<td>empty string</td>
</tr>
</tbody>
</table>

See section [Structured Parse Results](#structured-parse-results) for more information on the result object. Changed in version 3.2: Result is a structured object rather than a simple 2-tuple.

## 21.8.2 Parsing ASCII Encoded Bytes

The URL parsing functions were originally designed to operate on character strings only. In practice, it is useful to be able to manipulate properly quoted and encoded URLs as sequences of ASCII bytes. Accordingly, the URL parsing functions in this module all operate on `bytes` and `bytearray` objects in addition to `str` objects.

If `str` data is passed in, the result will also contain only `str` data. If `bytes` or `bytearray` data is passed in, the result will contain only `bytes` data.

Attempting to mix `str` data with `bytes` or `bytearray` in a single function call will result in a `TypeError` being raised, while attempting to pass in non-ASCII byte values will trigger a `UnicodeDecodeError`.

To support easier conversion of result objects between `str` and `bytes`, all return values from URL parsing functions provide either an `encode()` method (when the result contains `str` data) or a `decode()` method (when the result contains `bytes` data). The signatures of these methods match those of the corresponding `str` and `bytes` methods (except that the default encoding is ‘ascii’ rather than ‘utf-8’). Each produces a value of a corresponding type that contains either `bytes` data (for `encode()` methods) or `str` data (for `decode()` methods).

Applications that need to operate on potentially improperly quoted URLs that may contain non-ASCII data will need to do their own decoding from bytes to characters before invoking the URL parsing methods.

The behaviour described in this section applies only to the URL parsing functions. The URL quoting functions use their own rules when producing or consuming byte sequences as detailed in the documentation of the individual URL quoting functions. Changed in version 3.2: URL parsing functions now accept ASCII encoded byte sequences.

## 21.8.3 Structured Parse Results

The result objects from the `urlparse()`, `urlsplit()` and `urldefrag()` functions are subclasses of the `tuple` type. These subclasses add the attributes listed in the documentation for those functions, the encoding and decoding support described in the previous section, as well as an additional method:

```
urllib.parse.SplitResult.geturl()
```

Return the re-combined version of the original URL as a string. This may differ from the original URL in that the scheme may be normalized to lower case and empty components may be dropped. Specifically, empty parameters, queries, and fragment identifiers will be removed.

For `urldefrag()` results, only empty fragment identifiers will be removed. For `urlsplit()` and `urlparse()` results, all noted changes will be made to the URL returned by this method.

The result of this method remains unchanged if passed back through the original parsing function.
>>> from urllib.parse import urlsplit
>>> url = 'HTTP://www.Python.org/doc/#'
>>> r1 = urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
>>> r2 = urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'

The following classes provide the implementations of the structured parse results when operating on `str` objects:

```python
class urllib.parse.DefragResult(url, fragment)
    Concrete class for urldefrag() results containing str data. The encode() method returns a
    DefragResultBytes instance. New in version 3.2.

class urllib.parse.ParseResult(scheme, netloc, path, params, query, fragment)
    Concrete class for urlparse() results containing str data. The encode() method returns a
    ParseResultBytes instance.

class urllib.parse.SplitResult(scheme, netloc, path, query, fragment)
    Concrete class for urlsplit() results containing str data. The encode() method returns a
    SplitResultBytes instance. New in version 3.2.
```

The following classes provide the implementations of the parse results when operating on `bytes` or `bytearray` objects:

```python
class urllib.parse.DefragResultBytes(url, fragment)
    Concrete class for urldefrag() results containing bytes data. The decode() method returns a
    DefragResult instance. New in version 3.2.

class urllib.parse.ParseResultBytes(scheme, netloc, path, params, query, fragment)
    Concrete class for urlparse() results containing bytes data. The decode() method returns a
    ParseResult instance. New in version 3.2.

class urllib.parse.SplitResultBytes(scheme, netloc, path, query, fragment)
    Concrete class for urlsplit() results containing bytes data. The decode() method returns a
    SplitResult instance. New in version 3.2.
```

### 21.8.4 URL Quoting

The URL quoting functions focus on taking program data and making it safe for use as URL components by quoting special characters and appropriately encoding non-ASCII text. They also support reversing these operations to recreate the original data from the contents of a URL component if that task isn’t already covered by the URL parsing functions above.

```python
urllib.parse.quote(string, safe='/', encoding=None, errors=None)
```

Replace special characters in `string` using the `%xx` escape. Letters, digits, and the characters ‘-’ are never quoted. By default, this function is intended for quoting the path section of URL. The optional `safe` parameter specifies additional ASCII characters that should not be quoted — its default value is ‘/’.

- `string` may be either a `str` or a `bytes`.
- The optional `encoding` and `errors` parameters specify how to deal with non-ASCII characters, as accepted by the `str.encode()` method. `encoding` defaults to ‘utf-8’. `errors` defaults to ‘strict’, meaning unsupported characters raise a `UnicodeEncodeError`. `encoding` and `errors` must not be supplied if `string` is a `bytes`, or a `TypeError` is raised.

Note that `quote(string, safe, encoding, errors)` is equivalent to `quote_from_bytes(string.encode(encoding, errors), safe)`.

Example: `quote('/El Niño/')` yields `'/%E1%20Ni%C3%B1o/'`.

```python
urllib.parse.quote_plus(string, safe=' ', encoding=None, errors=None)
```

Like `quote()`, but also replace spaces by plus signs, as required for quoting HTML form values when
building up a query string to go into a URL. Plus signs in the original string are escaped unless they are included in safe. It also does not have safe default to ’/’.

Example: `quote_plus('/El Niño/')` yields ‘%2FE+Ni%C3%B1o%2F’.

`urllib.parse.quote_from_bytes(bytes, safe='/')`  
Like `quote()`, but accepts a bytes object rather than a str, and does not perform string-to-bytes encoding.

Example: `quote_from_bytes(b’a&\xef’)` yields ‘a%26\xf’.

`urllib.parse.unquote(string, encoding='utf-8', errors='replace')`  
Replace %xx escapes by their single-character equivalent. The optional encoding and errors parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the bytes.decode() method.

string must be a str.
encoding defaults to ‘utf-8’. errors defaults to ‘replace’, meaning invalid sequences are replaced by a placeholder character.

Example: `unquote('/El%20Ni%C3%B1o/')` yields ‘/El Niño/’.

`urllib.parse.unquote_plus(string, encoding='utf-8', errors='replace')`  
Like `unquote()`, but also replace plus signs by spaces, as required for unquoting HTML form values.

string must be a str.

Example: `unquote_plus('/El+Ni%C3%B1o/')` yields ‘/El Niño/’.

`urllib.parse.unquote_to_bytes(string)`  
Replace %xx escapes by their single-octet equivalent, and return a bytes object.

string may be either a str or a bytes.

If it is a str, unescaped non-ASCII characters in string are encoded into UTF-8 bytes.

Example: `unquote_to_bytes('a%26%EF')` yields b’a&\xef’.

`urllib.parse.urlencode(query, doseq=False, safe='', encoding=None, errors=None)`  
Convert a mapping object or a sequence of two-element tuples, which may either be a str or a bytes, to a “percent-encoded” string. If the resultant string is to be used as a data for POST operation with urlopen() function, then it should be properly encoded to bytes, otherwise it would result in a TypeError.

The resulting string is a series of key=value pairs separated by ‘&’ characters, where both key and value are quoted using `quote_plus()` above. When a sequence of two-element tuples is used as the query argument, the first element of each tuple is a key and the second is a value. The value element in itself can be a sequence and in that case, if the optional parameter doseq is evaluates to True, individual key=value pairs separated by ‘&’ are generated for each element of the value sequence for the key. The order of parameters in the encoded string will match the order of parameter tuples in the sequence.

When query parameter is a str, the safe, encoding and error parameters are passed down to `quote_plus()` for encoding.

To reverse this encoding process, `parse_qsl()` and `parse_qsl()` are provided in this module to parse query strings into Python data structures.

Refer to urllib examples to find out how urlencode method can be used for generating query string for a URL or data for POST. Changed in version 3.2: Query parameter supports bytes and string objects.

See Also:

RFC 3986 - Uniform Resource Identifiers This is the current standard (STD66). Any changes to urllib.parse module should conform to this. Certain deviations could be observed, which are mostly for backward compatibility purposes and for certain de-facto parsing requirements as commonly observed in major browsers.

RFC 2732 - Format for Literal IPv6 Addresses in URL’s. This specifies the parsing requirements of IPv6 URLs.
The Python Library Reference, Release 3.3.3

RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax  Document describing the generic syntactic requirements for both Uniform Resource Names (URNs) and Uniform Resource Locators (URLs).

RFC 2368 - The mailto URL scheme. Parsing requirements for mailto url schemes.

RFC 1808 - Relative Uniform Resource Locators  This Request For Comments includes the rules for joining an absolute and a relative URL, including a fair number of “Abnormal Examples” which govern the treatment of border cases.

RFC 1738 - Uniform Resource Locators (URL)  This specifies the formal syntax and semantics of absolute URLs.

21.9 urllib.error — Exception classes raised by urlib.request

The urllib.error module defines the exception classes for exceptions raised by urlib.request. The base exception class is URLError.

The following exceptions are raised by urllib.error as appropriate:

exception urllib.error.URLError

The handlers raise this exception (or derived exceptions) when they run into a problem. It is a subclass of OSError.

reason

The reason for this error. It can be a message string or another exception instance.

Changed in version 3.3: URLError has been made a subclass of OSError instead of IOError.

exception urllib.error.HTTPError

Though being an exception (a subclass ofURLError), an HTTPError can also function as a non-exceptional file-like return value (the same thing that urlopen() returns). This is useful when handling exotic HTTP errors, such as requests for authentication.

code

An HTTP status code as defined in RFC 2616. This numeric value corresponds to a value found in the dictionary of codes as found in http.server.BaseHTTPRequestHandler.responses.

reason

This is usually a string explaining the reason for this error.

exception urllib.error.ContentTooShortError (msg, content)

This exception is raised when the urlretrieve() function detects that the amount of the downloaded data is less than the expected amount (given by the Content-Length header). The content attribute stores the downloaded (and supposedly truncated) data.

21.10 urllib.robotparser — Parser for robots.txt

This module provides a single class, RobotFileParser, which answers questions about whether or not a particular user agent can fetch a URL on the Web site that published the robots.txt file. For more details on the structure of robots.txt files, see http://www.robotstxt.org/orig.html.

class urllib.robotparser.RobotFileParser (url=’’)

This class provides methods to read, parse and answer questions about the robots.txt file at url.

set_url (url)

Sets the URL referring to a robots.txt file.

read()

Reads the robots.txt URL and feeds it to the parser.

parse (lines)

Parses the lines argument.
can_fetch (useragent, url)

Returns True if the useragent is allowed to fetch the url according to the rules contained in the parsed robots.txt file.

mtime()

Returns the time the robots.txt file was last fetched. This is useful for long-running web spiders that need to check for new robots.txt files periodically.

modified()

Sets the time the robots.txt file was last fetched to the current time.

The following example demonstrates basic use of the RobotFileParser class.

```python
>>> import urllib.robotparser
>>> rp = urllib.robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```

### 21.11 http — HTTP modules

http is a package that collects several modules for working with the HyperText Transfer Protocol:

- **http.client** is a low-level HTTP protocol client; for high-level URL opening use `urllib.request`
- **http.server** contains basic HTTP server classes based on `socketserver`
- **http.cookies** has utilities for implementing state management with cookies
- **http.cookiejar** provides persistence of cookies

### 21.12 http.client — HTTP protocol client

**Source code:** Lib/http/client.py

This module defines classes which implement the client side of the HTTP and HTTPS protocols. It is normally not used directly — the module `urllib.request` uses it to handle URLs that use HTTP and HTTPS.

**Note:** HTTPS support is only available if Python was compiled with SSL support (through the `ssl` module).

The module provides the following classes:

```python
class http.client.HTTPConnection (host, port=None[, strict][, timeout], source_address=None)
```

An HTTPConnection instance represents one transaction with an HTTP server. It should be instantiated passing in a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form host:port, else the default HTTP port (80) is used. If the optional `timeout` parameter is given, blocking operations (like connection attempts) will timeout after that many seconds (if it is not given, the global default timeout setting is used). The optional `source_address` parameter may be a tuple of a (host, port) to use as the source address the HTTP connection is made from.

For example, the following calls all create instances that connect to the server at the same host and port:

```python
>>> h1 = http.client.HTTPConnection('www.cwi.nl')
>>> h2 = http.client.HTTPConnection('www.cwi.nl:80')
```
>>> h3 = http.client.HTTPConnection('www.cwi.nl', 80)
>>> h3 = http.client.HTTPConnection('www.cwi.nl', 80, timeout=10)

Changed in version 3.2: source_address was added. Deprecated since version 3.2, will be removed in version 3.4: The strict parameter is deprecated. HTTP 0.9-style “Simple Responses” are not supported anymore.

class http.client.HTTPConnection (host, port=None, key_file=None, cert_file=None[, strict][, source_address=None, *, context=None, check_hostname=None])
A subclass of HTTPConnection that uses SSL for communication with secure servers. Default port is 443. If context is specified, it must be a ssl.SSLContext instance describing the various SSL options. If context is specified and has a verify_mode of either CERT_OPTIONAL or CERT_REQUIRED, then by default host is matched against the host name(s) allowed by the server’s certificate. If you want to change that behaviour, you can explicitly set check_hostname to False.

key_file and cert_file are deprecated, please use ssl.SSLContext.load_cert_chain() instead.

If you access arbitrary hosts on the Internet, it is recommended to require certificate checking and feed the context with a set of trusted CA certificates:

c = ssl.SSLContext(ssl.PROTOCOL_TLSv1)
c.verify_mode = ssl.CERT_REQUIRED
c.load_verify_locations('/etc/pki/tls/certs/ca-bundle.crt')
h = client.HTTPSConnection('svn.python.org', 443, context=ctx)

Changed in version 3.2: source_address, context and check_hostname were added. Changed in version 3.2: This class now supports HTTPS virtual hosts if possible (that is, if ssl.HAS_SNI is true). Deprecated since version 3.2, will be removed in version 3.4: The strict parameter is deprecated. HTTP 0.9-style “Simple Responses” are not supported anymore.

class http.client.HTTPResponse (sock, debuglevel=0[, strict], method=None, url=None)
Class whose instances are returned upon successful connection. Not instantiated directly by user. Deprecated since version 3.2, will be removed in version 3.4: The strict parameter is deprecated. HTTP 0.9-style “Simple Responses” are not supported anymore.

The following exceptions are raised as appropriate:

exception http.client.HTTPException
The base class of the other exceptions in this module. It is a subclass of Exception.

exception http.client.NotConnected
A subclass of HTTPException.

exception http.client.InvalidURL
A subclass of HTTPException, raised if a port is given and is either non-numeric or empty.

exception http.client.UnknownProtocol
A subclass of HTTPException.

exception http.client.UnknownTransferEncoding
A subclass of HTTPException.

exception http.client.UnimplementedFileMode
A subclass of HTTPException.

exception http.clientIncompleteRead
A subclass of HTTPException.

exception http.client.ImproperConnectionState
A subclass of HTTPException.

exception http.client.CannotSendRequest
A subclass of ImproperConnectionState.

exception http.client.CannotSendHeader
A subclass of ImproperConnectionState.
exception http.client.ResponseNotReady
   A subclass of ImproperConnectionState.

exception http.client.BadStatusLine
   A subclass of HTTPException. Raised if a server responds with a HTTP status code that we don’t understand.

The constants defined in this module are:

http.client.HTTP_PORT
   The default port for the HTTP protocol (always 80).

http.clientHTTPS_PORT
   The default port for the HTTPS protocol (always 443).

and also the following constants for integer status codes:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTINUE</td>
<td>100</td>
<td>HTTP/1.1, RFC 2616, Section 10.1.1</td>
</tr>
<tr>
<td>SWITCHING_PROTOCOLS</td>
<td>101</td>
<td>HTTP/1.1, RFC 2616, Section 10.1.2</td>
</tr>
<tr>
<td>PROCESSING</td>
<td>102</td>
<td>WEBDAV, RFC 2518, Section 10.1</td>
</tr>
<tr>
<td>OK</td>
<td>200</td>
<td>HTTP/1.1, RFC 2616, Section 10.2.1</td>
</tr>
<tr>
<td>CREATED</td>
<td>201</td>
<td>HTTP/1.1, RFC 2616, Section 10.2.2</td>
</tr>
<tr>
<td>ACCEPTED</td>
<td>202</td>
<td>HTTP/1.1, RFC 2616, Section 10.2.3</td>
</tr>
<tr>
<td>NON_AUTHORITATIVE_INFORMATION</td>
<td>203</td>
<td>HTTP/1.1, RFC 2616, Section 10.2.4</td>
</tr>
<tr>
<td>NO_CONTENT</td>
<td>204</td>
<td>HTTP/1.1, RFC 2616, Section 10.2.5</td>
</tr>
<tr>
<td>RESET_CONTENT</td>
<td>205</td>
<td>HTTP/1.1, RFC 2616, Section 10.2.6</td>
</tr>
<tr>
<td>PARTIAL_CONTENT</td>
<td>206</td>
<td>HTTP/1.1, RFC 2616, Section 10.2.7</td>
</tr>
<tr>
<td>MULTI_STATUS</td>
<td>207</td>
<td>WEBDAV RFC 2518, Section 10.2</td>
</tr>
<tr>
<td>IM_USED</td>
<td>226</td>
<td>Delta encoding in HTTP, RFC 3229, Section 10.4.1</td>
</tr>
<tr>
<td>MULTIPLE_CHOICES</td>
<td>300</td>
<td>HTTP/1.1, RFC 2616, Section 10.3.1</td>
</tr>
<tr>
<td>MOVED_PERMANENTLY</td>
<td>301</td>
<td>HTTP/1.1, RFC 2616, Section 10.3.2</td>
</tr>
<tr>
<td>FOUND</td>
<td>302</td>
<td>HTTP/1.1, RFC 2616, Section 10.3.3</td>
</tr>
<tr>
<td>SEE_Other</td>
<td>303</td>
<td>HTTP/1.1, RFC 2616, Section 10.3.4</td>
</tr>
<tr>
<td>NOT_MODIFIED</td>
<td>304</td>
<td>HTTP/1.1, RFC 2616, Section 10.3.5</td>
</tr>
<tr>
<td>USE_PROXY</td>
<td>305</td>
<td>HTTP/1.1, RFC 2616, Section 10.3.6</td>
</tr>
<tr>
<td>TEMPORARY_REDIRECT</td>
<td>307</td>
<td>HTTP/1.1, RFC 2616, Section 10.3.8</td>
</tr>
<tr>
<td>BAD_REQUEST</td>
<td>400</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.1</td>
</tr>
<tr>
<td>UNAUTHORIZED</td>
<td>401</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.2</td>
</tr>
<tr>
<td>PAYMENT_REQUIRED</td>
<td>402</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.3</td>
</tr>
<tr>
<td>FORBIDDEN</td>
<td>403</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.4</td>
</tr>
<tr>
<td>NOT_FOUND</td>
<td>404</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.5</td>
</tr>
<tr>
<td>METHOD_NOT_ALLOWED</td>
<td>405</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.6</td>
</tr>
<tr>
<td>NOT_ACCEPTABLE</td>
<td>406</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.7</td>
</tr>
<tr>
<td>PROXY_AUTHENTICATION_REQUIRED</td>
<td>407</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.8</td>
</tr>
<tr>
<td>REQUEST_TIMEOUT</td>
<td>408</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.9</td>
</tr>
<tr>
<td>CONFLICT</td>
<td>409</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.10</td>
</tr>
<tr>
<td>GONE</td>
<td>410</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.11</td>
</tr>
<tr>
<td>LENGTH_REQUIRED</td>
<td>411</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.12</td>
</tr>
<tr>
<td>PRECONDITION_FAILED</td>
<td>412</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.13</td>
</tr>
<tr>
<td>REQUEST_ENTITY_TOO_LARGE</td>
<td>413</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.14</td>
</tr>
<tr>
<td>REQUEST_URI_TOO_LARGE</td>
<td>414</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.15</td>
</tr>
<tr>
<td>UNSUPPORTED_MEDIA_TYPE</td>
<td>415</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.16</td>
</tr>
<tr>
<td>REQUESTED_RANGE_NOT_SATISFIABLE</td>
<td>416</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.17</td>
</tr>
<tr>
<td>EXPECTATION_FAILED</td>
<td>417</td>
<td>HTTP/1.1, RFC 2616, Section 10.4.18</td>
</tr>
<tr>
<td>UNPROCESSABLE_ENTITY</td>
<td>422</td>
<td>WEBDAV RFC 2518, Section 10.3</td>
</tr>
<tr>
<td>LOCKED</td>
<td>423</td>
<td>WEBDAV RFC 2518, Section 10.4</td>
</tr>
<tr>
<td>FAILED_DEPENDENCY</td>
<td>424</td>
<td>HTTP Upgrade to TLS, RFC 2817, Section 6</td>
</tr>
<tr>
<td>UPGRADE_REQUIRED</td>
<td>426</td>
<td>Additional HTTP Status Codes, RFC 6585, Section 3</td>
</tr>
<tr>
<td>PRECONDITION_REQUIRED</td>
<td>428</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
Table 21.1 – continued from previous page

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
<th>HTTP Status Codes, RFC 6585, Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOO_MANY_REQUESTS</td>
<td></td>
<td>429</td>
</tr>
<tr>
<td>REQUEST_HEADER_FIELDS_TOO_LARGE</td>
<td></td>
<td>431</td>
</tr>
<tr>
<td>INTERNAL_SERVER_ERROR</td>
<td></td>
<td>500</td>
</tr>
<tr>
<td>NOT_IMPLEMENTED</td>
<td></td>
<td>501</td>
</tr>
<tr>
<td>BAD_GATEWAY</td>
<td></td>
<td>502</td>
</tr>
<tr>
<td>SERVICE_UNAVAILABLE</td>
<td></td>
<td>503</td>
</tr>
<tr>
<td>GATEWAY_TIMEOUT</td>
<td></td>
<td>504</td>
</tr>
<tr>
<td>HTTP_VERSION_NOT_SUPPORTED</td>
<td></td>
<td>505</td>
</tr>
<tr>
<td>INSUFFICIENT_STORAGE</td>
<td></td>
<td>507</td>
</tr>
<tr>
<td>NOT_EXTENDED</td>
<td></td>
<td>510</td>
</tr>
<tr>
<td>NETWORK_AUTHENTICATION_REQUIRED</td>
<td></td>
<td>511</td>
</tr>
</tbody>
</table>

Changed in version 3.3: Added codes 428, 429, 431 and 511 from RFC 6585.

http.client.responses

This dictionary maps the HTTP 1.1 status codes to the W3C names.


21.12.1 HTTPConnection Objects

HTTPConnection instances have the following methods:

HTTPConnection.request (method, url, body=None, headers={})

This will send a request to the server using the HTTP request method method and the selector url. If the body argument is present, it should be string or bytes object of data to send after the headers are finished. Strings are encoded as ISO-8859-1, the default charset for HTTP. To use other encodings, pass a bytes object. The Content-Length header is set to the length of the string.

The body may also be an open file object, in which case the contents of the file is sent; this file object should support fileno() and read() methods. The header Content-Length is automatically set to the length of the file as reported by stat. The body argument may also be an iterable and Content-Length header should be explicitly provided when the body is an iterable.

The headers argument should be a mapping of extra HTTP headers to send with the request. New in version 3.2: body can now be an iterable.

HTTPConnection.getresponse ()

Should be called after a request is sent to get the response from the server. Returns an HTTPResponse instance.

**Note:** Note that you must have read the whole response before you can send a new request to the server.

HTTPConnection.set_debuglevel (level)

Set the debugging level. The default debug level is 0, meaning no debugging output is printed. Any value greater than 0 will cause all currently defined debug output to be printed to stdout. The debuglevel is passed to any new HTTPResponse objects that are created. New in version 3.1.

HTTPConnection.set_tunnel (host, port=None, headers=None)

Set the host and the port for HTTP Connect Tunnelling. Normally used when it is required to a HTTPS Connection through a proxy server.

The headers argument should be a mapping of extra HTTP headers to send with the CONNECT request. New in version 3.2.

HTTPConnection.connect ()

Connect to the server specified when the object was created.

HTTPConnection.close ()

Close the connection to the server.
As an alternative to using the `request()` method described above, you can also send your request step by step,
by using the four functions below.

**HTTPConnection.putrequest** (request, selector, skip_host=False, skip_accept_encoding=False)
This should be the first call after the connection to the server has been made. It sends a line to the server
consisting of the request string, the selector string, and the HTTP version (HTTP/1.1). To disable au-
tomatic sending of Host: or Accept-Encoding: headers (for example to accept additional content
encodings), specify skip_host or skip_accept_encoding with non-False values.

**HTTPConnection.putheader** (header, argument[,...])
Send an RFC 822-style header to the server. It sends a line to the server consisting of the header, a colon
and a space, and the first argument. If more arguments are given, continuation lines are sent, each consisting
of a tab and an argument.

**HTTPConnection.endheaders** (message_body=None)
Send a blank line to the server, signalling the end of the headers. The optional message_body argument can
be used to pass a message body associated with the request. The message body will be sent in the same
packet as the message headers if it is string, otherwise it is sent in a separate packet.

**HTTPConnection.send** (data)
Send data to the server. This should be used directly only after the endheaders() method has been
called and before getresponse() is called.

### 21.12.2 HTTPResponse Objects

An HTTPResponse instance wraps the HTTP response from the server. It provides access to the request headers
and the entity body. The response is an iterable object and can be used in a with statement.

HTTPResponse.read([amt])
Reads and returns the response body, or up to the next amt bytes.

HTTPResponse.readinto (b)
Reads up to the next len(b) bytes of the response body into the buffer b. Returns the number of bytes read.
New in version 3.3.

HTTPResponse.getheader (name, default=None)
Return the value of the header name, or default if there is no header matching name. If there is more than
one header with the name name, return all of the values joined by ‘,’. If ‘default’ is any iterable other than
a single string, its elements are similarly returned joined by commas.

HTTPResponse.getheaders ()
Return a list of (header, value) tuples.

HTTPResponse.fileno ()
Return the fileno of the underlying socket.

HTTPResponse.msg
A http.client.HTTPMessage instance containing the response headers.
http.client.HTTPMessage is a subclass of email.message.Message.

HTTPResponse.version
HTTP protocol version used by server. 10 for HTTP/1.0, 11 for HTTP/1.1.

HTTPResponse.status
Status code returned by server.

HTTPResponse.reason
Reason phrase returned by server.

HTTPResponse.debuglevel
A debugging hook. If debuglevel is greater than zero, messages will be printed to stdout as the response
is read and parsed.

HTTPResponse.closed
Is True if the stream is closed.
Here is an example session that uses the **GET** method:

```python
>>> import http.client
>>> conn = http.client.HTTPConnection("www.python.org")
>>> conn.request("GET", "/index.html")
>>> r1 = conn.getresponse()
>>> print(r1.status, r1.reason)
200 OK
>>> data1 = r1.read()  # This will return entire content.
>>> # The following example demonstrates reading data in chunks.
>>> conn.request("GET", "/index.html")
>>> r1 = conn.getresponse()
>>> while not r1.closed:
...    print(r1.read(200))  # 200 bytes
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"...
...  
>>> # Example of an invalid request
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
>>> print(r2.status, r2.reason)
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

Here is an example session that uses the **HEAD** method. Note that the **HEAD** method never returns any data.

```python
>>> import http.client
>>> conn = http.client.HTTPConnection("www.python.org")
>>> conn.request("HEAD","/index.html")
>>> res = conn.getresponse()
>>> print(res.status, res.reason)
200 OK
>>> data = res.read()
>>> print(len(data))
0
>>> data == b''
True
```

Here is an example session that shows how to **POST** requests:

```python
>>> import http.client, urllib.parse
>>> params = urllib.parse.urlencode({'@number': 12524, '@type': 'issue', '@action': 'show'})
>>> headers = {'Content-type': 'application/x-www-form-urlencoded',
...            'Accept': 'text/plain'}
>>> conn = http.client.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "/", params, headers)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
302 Found
>>> data = response.read()
>>> print(data)
b'Redirecting to <a href="http://bugs.python.org/issue12524">http://bugs.python.org/issue12524</a>"
>>> conn.close()
```

Client side **HTTP PUT** requests are very similar to **POST** requests. The difference lies only on the server side where HTTP server will allow resources to be created via PUT request. It should be noted that custom HTTP methods are also handled in `urllib.request.Request` by sending the appropriate method attribute.Here is an example session that shows how to do **PUT** request using `http.client`:

```python
```
>>> # This creates an HTTP message
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/foobar
...  
>>> import http.client
>>> BODY = "***filecontents***"
>>> conn = http.client.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200, OK

21.12.4 HTTPMessage Objects

An http.client.HTTPMessage instance holds the headers from an HTTP response. It is implemented using the email.message.Message class.

21.13 ftplib — FTP protocol client

Source code: Lib/ftplib.py

This module defines the class FTP and a few related items. The FTP class implements the client side of the FTP protocol. You can use this to write Python programs that perform a variety of automated FTP jobs, such as mirroring other ftp servers. It is also used by the module urllib.request to handle URLs that use FTP. For more information on FTP (File Transfer Protocol), see Internet RFC 959.

Here’s a sample session using the ftplib module:

```python
>>> from ftplib import FTP
>>> ftp = FTP(‘ftp.debian.org’)  # connect to host, default port
>>> ftp.login()  # user anonymous, passwd anonymous@‘230 Login successful.’
>>> ftp.cwd(‘debian’)  # change into "debian" directory
>>> ftp.retrlines(‘LIST’)  # list directory contents
-rw-rw-r-- 1 1176 1176 1063 Jun 15 10:18 README
...  
-rw-rw-r-- 1 1176 1176 4096 Dec 19 2000 pool  
-rw-rw-r-- 1 1176 1176 4096 Nov 17 2008 project
-rw-rw-r-- 1 1176 1176 4096 Oct 10 2012 tools
‘226 Directory send OK.’
>>> ftp.retrbinary(‘RETR README’, open(‘README’, ‘wb’).write)
‘226 Transfer complete.’
>>> ftp.quit()
```

The module defines the following items:

```python
class ftplib.FTP (host='', user='', passwd='', acct='', timeout=None, source_address=None)
```

Return a new instance of the FTP class. When host is given, the method call connect(host) is made. When user is given, additionally the method call login(user, passwd, acct) is made (where passwd and acct default to the empty string when not given). The optional timeout parameter specifies a timeout in seconds for blocking operations like the connection attempt (if is not specified, the global default timeout setting will be used). source_address is a 2-tuple (host, port) for the socket to bind to as its source address before connecting.

FTP class supports the with statement. Here is a sample on how using it:
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
...     ftp.login()
...     ftp.dir()
...
'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x 9 ftp ftp 154 May 6 10:43 .
dr-xr-xr-x 9 ftp ftp 154 May 6 10:43 ..
dr-xr-xr-x 5 ftp ftp 4096 May 6 10:43 CentOS
dr-xr-xr-x 3 ftp ftp 18 Jul 10 2008 Fedora

>>> Changed in version 3.2: Support for the with statement was added. Changed in version 3.3: source_address parameter was added.

class ftplib.FTP_TLS(host='', user='', passwd='', acct='', keyfile=None, certfile=None, context=None, timeout=None, source_address=None):
A FTP subclass which adds TLS support to FTP as described in RFC 4217. Connect as usual to port 21 implicitly securing the FTP control connection before authenticating. Securing the data connection requires the user to explicitly ask for it by calling the prot_p() method. keyfile and certfile are optional – they can contain a PEM formatted private key and certificate chain file name for the SSL connection. context parameter is a ssl.SSLContext object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. source_address is a 2-tuple (host, port) for the socket to bind to as its source address before connecting. New in version 3.2. Changed in version 3.3: source_address parameter was added. Here’s a sample session using the FTP_TLS class:

>>> from ftplib import FTP_TLS
>>> ftps = FTP_TLS('ftp.python.org')
>>> ftps.login() # login anonymously before securing control channel
>>> ftps.prot_p() # switch to secure data connection
>>> ftps.retrlines('LIST') # list directory content securely
total 9
drwxr-xr-x 8 root wheel 1024 Jan 3 1994 .
drwxr-xr-x 8 root wheel 1024 Jan 3 1994 ..
drwxr-xr-x 2 root wheel 1024 Jan 3 1994 bin
drwxr-xr-x 2 root wheel 1024 Jan 3 1994 etc
d-wxrwxr-x 2 ftp wheel 1024 Sep 5 13:43 incoming
drwxr-xr-x 2 root wheel 1024 Nov 17 1993 lib
drwxr-xr-x 6 1094 wheel 1024 Sep 13 19:07 pub
drwxr-xr-x 3 root wheel 1024 Jan 3 1994 usr
-rwxr--r-- 1 root root 312 Aug 1 1994 welcome.msg
'226 Transfer complete.'
>>> ftps.quit()

>>> exception ftplib.error_reply
Exception raised when an unexpected reply is received from the server.

exception ftplib.error_temp
Exception raised when an error code signifying a temporary error (response codes in the range 400–499) is received.

exception ftplib.error_perm
Exception raised when an error code signifying a permanent error (response codes in the range 500–599) is received.

exception ftplib.error_proto
Exception raised when a reply is received from the server that does not fit the response specifications of the File Transfer Protocol, i.e. begin with a digit in the range 1–5.
The set of all exceptions (as a tuple) that methods of FTP instances may raise as a result of problems with the FTP connection (as opposed to programming errors made by the caller). This set includes the four exceptions listed above as well as OSError.

See Also:

Module netrc Parser for the .netrc file format. The file .netrc is typically used by FTP clients to load user authentication information before prompting the user.

The file Tools/scripts/ftpmirror.py in the Python source distribution is a script that can mirror FTP sites, or portions thereof, using the ftplib module. It can be used as an extended example that applies this module.

**21.13.1 FTP Objects**

Several methods are available in two flavors: one for handling text files and another for binary files. These are named for the command which is used followed by lines for the text version or binary for the binary version.

FTP instances have the following methods:

**FTP.set_debuglevel** (level)
Set the instance’s debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

**FTP.connect** (host='', port=0, timeout=None, source_address=None)
Connect to the given host and port. The default port number is 21, as specified by the FTP protocol specification. It is rarely needed to specify a different port number. This function should be called only once for each instance; it should not be called at all if a host was given when the instance was created. All other methods can only be used after a connection has been made. The optional timeout parameter specifies a timeout in seconds for the connection attempt. If no timeout is passed, the global default timeout setting will be used. source_address is a 2-tuple (host, port) for the socket to bind to as its source address before connecting. Changed in version 3.3: source_address parameter was added.

**FTP.getwelcome** ()
Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

**FTP.login** (user='anonymous', passwd='', acct='')
Log in as the given user. The passwd and acct parameters are optional and default to the empty string. If no user is specified, it defaults to ‘anonymous’. If user is ‘anonymous’, the default passwd is ‘anonymous@’. This function should be called only once for each instance, after a connection has been established; it should not be called at all if a host and user were given when the instance was created. Most FTP commands are only allowed after the client has logged in. The acct parameter supplies “accounting information”; few systems implement this.

**FTP.abort ()**
Abort a file transfer that is in progress. Using this does not always work, but it’s worth a try.

**FTP.sendcmd** (cmd)
Send a simple command string to the server and return the response string.

**FTP.voidcmd** (cmd)
Send a simple command string to the server and handle the response. Return nothing if a response code corresponding to success (codes in the range 200–299) is received. Raise error_reply otherwise.

**FTP.retrbinary** (cmd, callback, blocksize=8192, rest=None)
Retrieve a file in binary transfer mode. cmd should be an appropriate RETR command: ‘RETR filename’. The callback function is called for each block of data received, with a single string argument giving the data block. The optional blocksize argument specifies the maximum chunk size to read on the low-level socket object created to do the actual transfer (which will also be the largest size of the data blocks...
passed to `callback`). A reasonable default is chosen. `rest` means the same thing as in the `transfercmd()` method.

**FTP .retrlines** *(cmd, callback=None)*
Retrieve a file or directory listing in ASCII transfer mode. `cmd` should be an appropriate `RETR` command (see `retrbinary()`) or a command such as `LIST` or `NLST` (usually just the string ‘`LIST’`). `LIST` retrieves a list of files and information about those files. `NLST` retrieves a list of file names. The `callback` function is called for each line with a string argument containing the line with the trailing CRLF stripped. The default `callback` prints the line to `sys.stdout`.

**FTP .set_pasv**(boolean)
Enable “passive” mode if `boolean` is true, other disable passive mode. Passive mode is on by default.

**FTP .storbinary** *(cmd, file, blocksize=8192, callback=None, rest=None)*
Store a file in binary transfer mode. `cmd` should be an appropriate `STOR` command: "STOR filename". `file` is a `file object` (opened in binary mode) which is read until EOF using its method in blocks of size `blocksize` to provide the data to be stored. The `blocksize` argument defaults to 8192. `callback` is an optional single parameter callable that is called on each block of data after it is sent. `rest` means the same thing as in the `transfercmd()` method. Changed in version 3.2: `rest` parameter added.

**FTP .storlines** *(cmd, file, callback=None)*
Store a file in ASCII transfer mode. `cmd` should be an appropriate `STOR` command (see `storbinary()`). Lines are read until EOF from the `file object` `file` (opened in binary mode) using its method to provide the data to be stored. `callback` is an optional single parameter callable that is called on each line after it is sent.

**FTP .transfercmd** *(cmd, rest=None)*
Initiate a transfer over the data connection. If the transfer is active, send a `EPRT` or `PORT` command and the transfer command specified by `cmd`, and accept the connection. If the server is passive, send a `EPSV` or `PASV` command, connect to it, and start the transfer command. Either way, return the socket for the connection.

If optional `rest` is given, a `REST` command is sent to the server, passing `rest` as an argument. `rest` is usually a byte offset into the requested file, telling the server to restart sending the file’s bytes at the requested offset, skipping over the initial bytes. Note however that RFC 959 requires only that `rest` be a string containing characters in the printable range from ASCII code 33 to ASCII code 126. The `transfercmd()` method, therefore, converts `rest` to a string, but no check is performed on the string’s contents. If the server does not recognize the `REST` command, an `error_reply` exception will be raised. If this happens, simply call `transfercmd()` without a `rest` argument.

**FTP .ntransfercmd** *(cmd, rest=None)*
Like `transfercmd()`, but returns a tuple of the data connection and the expected size of the data. If the expected size could not be computed, `None` will be returned as the expected size. `cmd` and `rest` means the same thing as in `transfercmd()`.

**FTP .mlsd**(path="", facts=[])
List a directory in a standardized format by using MLSD command (RFC 3659). If `path` is omitted the current directory is assumed. `facts` is a list of strings representing the type of information desired (e.g. ["type", "size", "perm"]). Return a generator object yielding a tuple of two elements for every file found in path. First element is the file name, the second one is a dictionary containing facts about the file name. Content of this dictionary might be limited by the `facts` argument but server is not guaranteed to return all requested facts. New in version 3.3.

**FTP .nlst**(argument=[], ...)
Return a list of file names as returned by the `NLST` command. The optional `argument` is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `NLST` command. Deprecated since version 3.3: use `mlsd()` instead.

**FTP .dir**(argument=[], ...)
Produce a directory listing as returned by the `LIST` command, printing it to standard output. The optional `argument` is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `LIST` command. If the last argument is a function, it is used as a
callback function as for retrlines(); the default prints to sys.stdout. This method returns None. Deprecated since version 3.3: use mlsd() instead.

FTP.rename(fromname, toname)
   Rename file fromname on the server to toname.

FTP.delete(filename)
   Remove the file named filename from the server. If successful, returns the text of the response, otherwise raises error_perm on permission errors or error_reply on other errors.

FTP.cwd(pathname)
   Set the current directory on the server.

FTP.mkd(pathname)
   Create a new directory on the server.

FTP.pwd()
   Return the pathname of the current directory on the server.

FTP.rmd(dirname)
   Remove the directory named dirname on the server.

FTP.size(filename)
   Request the size of the file named filename on the server. On success, the size of the file is returned as an integer, otherwise None is returned. Note that the SIZE command is not standardized, but is supported by many common server implementations.

FTP.quit()
   Send a QUIT command to the server and close the connection. This is the “polite” way to close a connection, but it may raise an exception if the server responds with an error to the QUIT command. This implies a call to the close() method which renders the FTP instance useless for subsequent calls (see below).

FTP.close()
   Close the connection unilaterally. This should not be applied to an already closed connection such as after a successful call to quit(). After this call the FTP instance should not be used any more (after a call to close() or quit() you cannot reopen the connection by issuing another login() method).

21.13.2 FTP_TLS Objects

FTP_TLS class inherits from FTP, defining these additional objects:

FTP_TLS.ssl_version
   The SSL version to use (defaults to TLSv1).

FTP_TLS.auth()
   Set up secure control connection by using TLS or SSL, depending on what specified in ssl_version() attribute.

FTP_TLS.ccc()
   Revert control channel back to plaintext. This can be useful to take advantage of firewalls that know how to handle NAT with non-secure FTP without opening fixed ports. New in version 3.3.

FTP_TLS.prot_p()
   Set up secure data connection.

FTP_TLS.prot_c()
   Set up clear text data connection.

21.14 poplib — POP3 protocol client

Source code: Lib/poplib.py
This module defines a class, `POP3`, which encapsulates a connection to a POP3 server and implements the protocol as defined in RFC 1725. The `POP3` class supports both the minimal and optional command sets. Additionally, this module provides a class `POP3_SSL`, which provides support for connecting to POP3 servers that use SSL as an underlying protocol layer.

Note that POP3, though widely supported, is obsolescent. The implementation quality of POP3 servers varies widely, and too many are quite poor. If your mailserver supports IMAP, you would be better off using the `imaplib.IMAP4` class, as IMAP servers tend to be better implemented.

The `poplib` module provides two classes:

- **class `poplib.POP3`**
  
  ```python
  class poplib.POP3 (host, port=POP3_PORT[, timeout])
  ```
  
  This class implements the actual POP3 protocol. The connection is created when the instance is initialized. If `port` is omitted, the standard POP3 port (110) is used. The optional `timeout` parameter specifies a timeout in seconds for the connection attempt (if not specified, the global default timeout setting will be used).

- **class `poplib.POP3_SSL`**
  
  ```python
  class poplib.POP3_SSL (host, port=POP3_SSL_PORT, keyfile=None, certfile=None, timeout=None, context=None)
  ```
  
  This is a subclass of `POP3` that connects to the server over an SSL encrypted socket. If `port` is not specified, 995, the standard POP3-over-SSL port is used. `keyfile` and `certfile` are also optional - they can contain a PEM formatted private key and certificate chain file for the SSL connection. `timeout` works as in the `POP3` constructor. `context` parameter is a `ssl.SSLContext` object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Changed in version 3.2: `context` parameter added.

One exception is defined as an attribute of the `poplib` module:

- **exception `poplib.error_proto`**
  
  Exception raised on any errors from this module (errors from `socket` module are not caught). The reason for the exception is passed to the constructor as a string.

See Also:

- **Module `imaplib`** The standard Python IMAP module.
- **Frequently Asked Questions About Fetchmail** The FAQ for the `fetchmail` POP/IMAP client collects information on POP3 server variations and RFC noncompliance that may be useful if you need to write an application based on the POP protocol.

### 21.14.1 POP3 Objects

All POP3 commands are represented by methods of the same name, in lower-case; most return the response text sent by the server.

An `POP3` instance has the following methods:

- **POP3.set_debuglevel (level)**
  
  Set the instance’s debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

- **POP3.getwelcome ()**
  
  Returns the greeting string sent by the POP3 server.

- **POP3.user (username)**
  
  Send user command, response should indicate that a password is required.

- **POP3.pass (password)**
  
  Send password, response includes message count and mailbox size. Note: the mailbox on the server is locked until `quit ()` is called.

- **POP3.apop (user, secret)**
  
  Use the more secure APOP authentication to log into the POP3 server.
POP3.**rpop**(user)

Use RPOP authentication (similar to UNIX r-commands) to log into POP3 server.

POP3.**stat**()

Get mailbox status. The result is a tuple of 2 integers: (message count, mailbox size).

POP3.**stat**([which])

Request message list, result is in the form (response, ['msg_num octets', ...], octets). If which is set, it is the message to list.

POP3.**retr**(which)

Retrieve whole message number which, and set its seen flag. Result is in form (response, ['line', ...], octets).

POP3.**dele**(which)

Flag message number which for deletion. On most servers deletions are not actually performed until QUIT (the major exception is Eudora QPOP, which deliberately violates the RFCs by doing pending deletes on any disconnect).

POP3.**rset**()

Remove any deletion marks for the mailbox.

POP3.**noop**()

Do nothing. Might be used as a keep-alive.

POP3.**quit**()

Signoff: commit changes, unlock mailbox, drop connection.

POP3.**top**(which, howmuch)

Retrieves the message header plus howmuch lines of the message after the header of message number which. Result is in form (response, ['line', ...], octets).

The POP3 TOP command this method uses, unlike the RETR command, doesn’t set the message’s seen flag; unfortunately, TOP is poorly specified in the RFCs and is frequently broken in off-brand servers. Test this method by hand against the POP3 servers you will use before trusting it.

POP3.**uidl**(which=None)

Return message digest (unique id) list. If which is specified, result contains the unique id for that message in the form 'response mesgnum uid, otherwise result is list (response, ['mesgnum uid', ...], octets).

Instances of POP3_SSL have no additional methods. The interface of this subclass is identical to its parent.

### 21.14.2 POP3 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```python
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print(j)
```

At the end of the module, there is a test section that contains a more extensive example of usage.

### 21.15 imaplib — IMAP4 protocol client

Source code: Lib/imaplib.py
This module defines three classes, **IMAP4**, **IMAP4_SSL** and **IMAP4_stream**, which encapsulate a connection to an IMAP4 server and implement a large subset of the IMAP4rev1 client protocol as defined in **RFC 2060**. It is backward compatible with IMAP4 (RFC 1730) servers, but note that the **STATUS** command is not supported in IMAP4.

Three classes are provided by the `imaplib` module, **IMAP4** is the base class:

```python
class imaplib.IMAP4 (host='', port=IMAP4_PORT)
```
This class implements the actual IMAP4 protocol. The connection is created and protocol version (IMAP4 or IMAP4rev1) is determined when the instance is initialized. If `host` is not specified, " (the local host) is used. If `port` is omitted, the standard IMAP4 port (143) is used.

Three exceptions are defined as attributes of the **IMAP4** class:

```python
exception IMAP4.error
Exception raised on any errors. The reason for the exception is passed to the constructor as a string.
```

```python
exception IMAP4.abort
IMAP4 server errors cause this exception to be raised. This is a sub-class of IMAP4.error. Note that closing the instance and instantiating a new one will usually allow recovery from this exception.
```

```python
exception IMAP4.readonly
This exception is raised when a writable mailbox has its status changed by the server. This is a sub-class of IMAP4.error. Some other client now has write permission, and the mailbox will need to be re-opened to re-obtain write permission.
```

There’s also a subclass for secure connections:

```python
class imaplib.IMAP4_SSL (host='', port=IMAP4_SSL_PORT, keyfile=None, certfile=None, ssl_context=None)
```
This is a subclass derived from **IMAP4** that connects over an SSL encrypted socket (to use this class you need a socket module that was compiled with SSL support). If `host` is not specified, " (the local host) is used. If `port` is omitted, the standard IMAP4-over-SSL port (993) is used. `keyfile` and `certfile` are also optional - they can contain a PEM formatted private key and certificate chain file for the SSL connection. The `ssl_context` parameter is a `ssl.SSLContext` object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Note that the `keyfile/certfile` parameters are mutually exclusive with `ssl_context`, a `ValueError` is raised if `keyfile/certfile` is provided along with `ssl_context`. Changed in version 3.3: `ssl_context` parameter added.

The second subclass allows for connections created by a child process:

```python
class imaplib.IMAP4_stream (command)
```
This is a subclass derived from **IMAP4** that connects to the stdin/stdout file descriptors created by passing `command` to `subprocess.Popen()`.

The following utility functions are defined:

```python
imaplib.Internaldate2tuple (datestr)
Parse an IMAP4 INTERNALDATE string and return corresponding local time. The return value is a `time.struct_time` tuple or None if the string has wrong format.
```

```python
imaplib.Int2AP (num)
Converts an integer into a string representation using characters from the set [A .. P].
```

```python
imaplib.ParseFlags (flagstr)
Converts an IMAP4 FLAGS response to a tuple of individual flags.
```

```python
imaplib.Time2Internaldate (date_time)
Convert `date_time` to an IMAP4 INTERNALDATE representation. The return value is a string in the form: "DD–Mmm–YYYY HH:MM:SS +HHMM" (including double-quotes). The `date_time` argument can be a number (int or float) representing seconds since epoch (as returned by `time.time()`), a 9-tuple representing local time an instance of `time.struct_time` (as returned by `time.localtime()`), an aware instance of `datetime.datetime`, or a double-quoted string. In the last case, it is assumed to already be in the correct format.
```
Note that IMAP4 message numbers change as the mailbox changes; in particular, after an `EXPUNGE` command performs deletions the remaining messages are renumbered. So it is highly advisable to use UIDs instead, with the UID command.

At the end of the module, there is a test section that contains a more extensive example of usage.

See Also:

Documents describing the protocol, and sources and binaries for servers implementing it, can all be found at the University of Washington’s IMAP Information Center (http://www.washington.edu/imap/).

21.15.1 IMAP4 Objects

All IMAP4rev1 commands are represented by methods of the same name, either upper-case or lower-case.

All arguments to commands are converted to strings, except for AUTHENTICATE, and the last argument to APPEND which is passed as an IMAP4 literal. If necessary (the string contains IMAP4 protocol-sensitive characters and isn’t enclosed with either parentheses or double quotes) each string is quoted. However, the `password` argument to the LOGIN command is always quoted. If you want to avoid having an argument string quoted (eg: the `flags` argument to STORE then enclose the string in parentheses (eg: `r’(\Deleted)’`).

Each command returns a tuple: `(type, [data, ...])` where `type` is usually ‘OK’ or ‘NO’, and `data` is either the text from the command response, or mandated results from the command. Each `data` is either a string, or a tuple. If a tuple, then the first part is the header of the response, and the second part contains the data (ie: ‘literal’ value).

The `message_set` options to commands below is a string specifying one or more messages to be acted upon. It may be a simple message number (‘1′), a range of message numbers (‘2:4’), or a group of non-contiguous ranges separated by commas (‘1:3,6:9’). A range can contain an asterisk to indicate an infinite upper bound (‘3:*’).

An IMAP4 instance has the following methods:

**IMAP4.append**(mailbox, flags, date_time, message)

Append message to named mailbox.

**IMAP4.authenticate**(mechanism, authobject)

Authenticate command — requires response processing.

`mechanism` specifies which authentication mechanism is to be used - it should appear in the instance variable `capabilities` in the form AUTH=mechanism.

`authobject` must be a callable object:

```python
data = authobject(response)
```

It will be called to process server continuation responses; the `response` argument it is passed will be bytes. It should return bytes `data` that will be base64 encoded and sent to the server. It should return `None` if the client abort response * should be sent instead.

**IMAP4.check**()

Checkpoint mailbox on server.

**IMAP4.close**()

Close currently selected mailbox. Deleted messages are removed from writable mailbox. This is the recommended command before LOGOUT.

**IMAP4.copy**(message_set, new_mailbox)

Copy `message_set` messages onto end of `new_mailbox`.

**IMAP4.create**(mailbox)

Create new mailbox named `mailbox`.

**IMAP4.delete**(mailbox)

Delete old mailbox named `mailbox`.
IMAP4.**deleteacl** *(mailbox, who)*
Delete the ACLs (remove any rights) set for who on mailbox.

IMAP4.**expunge** ()
Permanently remove deleted items from selected mailbox. Generates an EXPUNGE response for each deleted message. Returned data contains a list of EXPUNGE message numbers in order received.

IMAP4.**fetch** *(message_set, message_parts)*
Fetch (parts of) messages. *message_parts* should be a string of message part names enclosed within parentheses, eg: "(UID BODY[TEXT])". Returned data are tuples of message part envelope and data.

IMAP4.**getacl** *(mailbox)*
Get the ACLs for mailbox. The method is non-standard, but is supported by the Cyrus server.

IMAP4.**getannotation** *(mailbox, entry, attribute)*
Retrieve the specified ANNOTATIONs for mailbox. The method is non-standard, but is supported by the Cyrus server.

IMAP4.**getquota** *(root)*
Get the quota root's resource usage and limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.**getquotaroot** *(mailbox)*
Get the list of quotaroots for the named mailbox. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.**list** *(directory, pattern)*
List mailbox names in directory matching pattern. *directory* defaults to the top-level mail folder, and *pattern* defaults to match anything. Returned data contains a list of LIST responses.

IMAP4.**login** *(user, password)*
Identify the client using a plaintext password. The *password* will be quoted.

IMAP4.**login_cram_md5** *(user, password)*
Force use of CRAM-MD5 authentication when identifying the client to protect the password. Will only work if the server CAPABILITY response includes the phrase AUTH=CRAM-MD5.

IMAP4.**logout** ()
Shut down connection to server. Returns server BYE response.

IMAP4.**lsub** *(directory="", pattern="*)
List subscribed mailbox names in directory matching pattern. *directory* defaults to the top level directory and *pattern* defaults to match any mailbox. Returned data are tuples of message part envelope and data.

IMAP4.**myrights** *(mailbox)*
Show my ACLs for a mailbox (i.e. the rights that I have on mailbox).

IMAP4.**namespace** ()
Returns IMAP namespaces as defined in RFC2342.

IMAP4.**noop** ()
Send NOOP to server.

IMAP4.**open** *(host, port)*
Opens socket to port at host. This method is implicitly called by the IMAP4 constructor. The connection objects established by this method will be used in the IMAP4.read(), IMAP4.readline(), IMAP4.send(), and IMAP4.shutdown() methods. You may override this method.

IMAP4.**partial** *(message_num, message_part, start, length)*
Fetch truncated part of a message. Returned data is a tuple of message part envelope and data.

IMAP4.**proxyauth** *(user)*
Assume authentication as user. Allows an authorised administrator to proxy into any user’s mailbox.

IMAP4.**read** *(size)*
Reads size bytes from the remote server. You may override this method.
**IMAP4.readline()**

Reads one line from the remote server. You may override this method.

**IMAP4.recent()**

Prompt server for an update. Returned data is None if no new messages, else value of RECENT response.

**IMAP4.rename(oldmailbox, newmailbox)**

Rename mailbox named oldmailbox to newmailbox.

**IMAP4.response(code)**

Return data for response code if received, or None. Returns the given code, instead of the usual type.

**IMAP4.search(charset, criterion[,...])**

Search mailbox for matching messages. charset may be None, in which case no CHARSET will be specified in the request to the server. The IMAP protocol requires that at least one criterion be specified; an exception will be raised when the server returns an error.

Example:

```python
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

**IMAP4.select(mailbox='INBOX', readonly=False)**

Select a mailbox. Returned data is the count of messages in mailbox (EXISTS response). The default mailbox is 'INBOX'. If the readonly flag is set, modifications to the mailbox are not allowed.

**IMAP4.send(data)**

Sends data to the remote server. You may override this method.

**IMAP4.setacl(mailbox, who, what)**

Set an ACL for mailbox. The method is non-standard, but is supported by the Cyrus server.

**IMAP4.setannotation(mailbox, entry, attribute[,...])**

Set ANNOTATIONs for mailbox. The method is non-standard, but is supported by the Cyrus server.

**IMAP4.setquota(root, limits)**

Set the quota root’s resource limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

**IMAP4.shutdown()**

Close connection established in open. This method is implicitly called by IMAP4.logout(). You may override this method.

**IMAP4.socket()**

Returns socket instance used to connect to server.

**IMAP4.sort(sort_criteria, charset, search_criterion[,...])**

The sort command is a variant of search with sorting semantics for the results. Returned data contains a space separated list of matching message numbers.

Sort has two arguments before the search_criterion argument(s): a parenthesized list of sort_criteria, and the searching charset. Note that unlike search, the searching charset argument is mandatory. There is also a uid sort command which corresponds to sort the way that uid search corresponds to search. The sort command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the numbers of matching messages.

This is an IMAP4rev1 extension command.

**IMAP4.starttls(ssl_context=None)**

Send a STARTTLS command. The ssl_context argument is optional and should be a ssl.SSLContext object. This will enable encryption on the IMAP connection. New in version 3.2.
IMAP4. **status** *(mailbox, names)*  
Request named status conditions for *mailbox*.

IMAP4. **store** *(message_set, command, flag_list)*  
Alters flag dispositions for messages in mailbox. *command* is specified by section 6.4.6 of RFC 2060 as being one of “FLAGS”, “+FLAGS”, or “-FLAGS”, optionally with a suffix of ”.SILENT”.

For example, to set the delete flag on all messages:

```python
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    M.store(num, '+FLAGS', '\Deleted')
M.expunge()
```

IMAP4. **subscribe** *(mailbox)*  
Subscribe to new mailbox.

IMAP4. **thread** *(threading_algorithm, charset, search_criterion[, ...]*)  
The thread command is a variant of search with threading semantics for the results. Returned data contains a space separated list of thread members.

Thread members consist of zero or more messages numbers, delimited by spaces, indicating successive parent and child.

Thread has two arguments before the *search_criterion* argument(s); a *threading_algorithm*, and the searching *charset*. Note that unlike search, the searching *charset* argument is mandatory. There is also a uid thread command which corresponds to thread the way that uid search corresponds to search. The thread command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the matching messages threaded according to the specified threading algorithm.

This is an IMAP4rev1 extension command.

IMAP4. **uid** *(command, arg[, ...]*)  
Execute command args with messages identified by UID, rather than message number. Returns response appropriate to command. At least one argument must be supplied; if none are provided, the server will return an error and an exception will be raised.

IMAP4. **unsubscribe** *(mailbox)*  
Unsubscribe from old mailbox.

IMAP4. **xatom** *(name[, ...]*)  
Allow simple extension commands notified by server in CAPABILITY response.

The following attributes are defined on instances of **IMAP4**:

**IMAP4.PROTOCOL_VERSION**  
The most recent supported protocol in the CAPABILITY response from the server.

**IMAP4.debug**  
Integer value to control debugging output. The initialize value is taken from the module variable Debug. Values greater than three trace each command.

### 21.15.2 IMAP4 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```python
import getpass, imaplib

M = imaplib.IMAP4()
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
```

Chapter 21. Internet Protocols and Support
typ, data = M.fetch(num, ‘(RFC822)’)
print(‘Message %s
%s
’ % (num, data[0][1]))
M.close()
M.logout()

21.16 nntplib — NNTP protocol client

Source code: Lib/nntplib.py

This module defines the class NNTP which implements the client side of the Network News Transfer Protocol. It can be used to implement a news reader or poster, or automated news processors. It is compatible with RFC 3977 as well as the older RFC 977 and RFC 2980.

Here are two small examples of how it can be used. To list some statistics about a newsgroup and print the subjects of the last 10 articles:

>>> s = nntplib.NNTP(‘news.gmane.org’)  
>>> resp, count, first, last, name = s.group(‘gmane.comp.python.committers’)  
>>> print(‘Group’, name, ‘has’, count, ‘articles, range’, first, ‘to’, last)  
Group gmane.comp.python.committers has 1096 articles, range 1 to 1096

>>> resp, overviews = s.over((last - 9, last))  
>>> for id, overview in overviews:  
...    print(id, nntplib.decode_header(overview[‘subject’]))

... 1087 Re: Commit privileges for Łukasz Langa  
1088 Re: 3.2 alpha 2 freeze  
1089 Re: 3.2 alpha 2 freeze  
1090 Re: Commit privileges for Łukasz Langa  
1091 Re: Commit privileges for Łukasz Langa  
1092 Updated ssh key  
1093 Re: Updated ssh key  
1094 Re: Updated ssh key  
1095 Hello fellow committers!  
1096 Re: Hello fellow committers!

>>> s.quit()  
’205 Bye!’

To post an article from a binary file (this assumes that the article has valid headers, and that you have right to post on the particular newsgroup):

>>> s = nntplib.NNTP(‘news.gmane.org’)  
>>> f = open(‘article.txt’, ‘rb’)  
>>> s.post(f)  
’240 Article posted successfully.’

>>> s.quit()  
’205 Bye!’

The module itself defines the following classes:

class nntplib.NNTP (host, port=119, user=None, password=None, readermode=None, usenetrc=False, timeout)

Return a new NNTP object, representing a connection to the NNTP server running on host host, listening at port port. An optional timeout can be specified for the socket connection. If the optional user and password are provided, or if suitable credentials are present in ~/.netrc and the optional flag usenetrc is true, the AUTHINFO USER and AUTHINFO PASS commands are used to identify and authenticate the user to the server. If the optional flag readermode is true, then a mode reader command is sent before authentication is performed. Reader mode is sometimes necessary if you are connecting to an NNTP server on the local machine and intend to call reader-specific commands, such as group. If you get unexpected
NNTPPermanentErrors, you might need to set readermode. NNTP class supports the with statement to unconditionally consume socket.error exceptions and to close the NNTP connection when done. Here is a sample on how using it:

```python
>>> from nntplib import NNTP
>>> with NNTP('news.gmane.org') as n:
...     n.group('gmane.comp.python.committers')
... ('211 1755 1 1755 gmane.comp.python.committers', 1755, 1, 1755, 'gmane.comp.python.committers')
```

Changed in version 3.2: usenetrc is now False by default. Changed in version 3.3: Support for the with statement was added.

class nntplib.NNTP_SSL (host, port=563, user=None, password=None, ssl_context=None, readermode=None, usenetrc=False, timeout=None):

Return a new NNTP_SSL object, representing an encrypted connection to the NNTP server running on host host, listening at port port. NNTP_SSL objects have the same methods as NNTP objects. If port is omitted, port 563 (NNTPS) is used. ssl_context is also optional, and is a SSLContext object. All other parameters behave the same as for NNTP.

Note that SSL-on-563 is discouraged per RFC 4642, in favor of STARTTLS as described below. However, some servers only support the former. New in version 3.2.

class nntplib.NNTPError

Derived from the standard exception Exception, this is the base class for all exceptions raised by the nntplib module. Instances of this class have the following attribute:

response

The response of the server if available, as a str object.

class nntplib.NNTPReplyError

Exception raised when an unexpected reply is received from the server.

class nntplib.NNTPTemporaryError

Exception raised when a response code in the range 400–499 is received.

class nntplib.NNTPPermanentError

Exception raised when a response code in the range 500–599 is received.

class nntplib.NNTPProtocolError

Exception raised when a reply is received from the server that does not begin with a digit in the range 1–5.

class nntplib.NNTPDataError

Exception raised when there is some error in the response data.

### 21.16.1 NNTP Objects

When connected, NNTP and NNTP_SSL objects support the following methods and attributes.

**Attributes**

NNTP.nntp_version

An integer representing the version of the NNTP protocol supported by the server. In practice, this should be 2 for servers advertising RFC 3977 compliance and 1 for others. New in version 3.2.

NNTP.nntp_implementation

A string describing the software name and version of the NNTP server, or None if not advertised by the server. New in version 3.2.
Methods

The response that is returned as the first item in the return tuple of almost all methods is the server’s response: a string beginning with a three-digit code. If the server’s response indicates an error, the method raises one of the above exceptions.

Many of the following methods take an optional keyword-only argument file. When the file argument is supplied, it must be either a file object opened for binary writing, or the name of an on-disk file to be written to. The method will then write any data returned by the server (except for the response line and the terminating dot) to the file; any list of lines, tuples or objects that the method normally returns will be empty. Changed in version 3.2: Many of the following methods have been reworked and fixed, which makes them incompatible with their 3.1 counterparts.

NNTP.quit()
Send a QUIT command and close the connection. Once this method has been called, no other methods of the NNTP object should be called.

NNTP.getwelcome()
Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

NNTP.getcapabilities()
Return the RFC 3977 capabilities advertised by the server, as a dict instance mapping capability names to (possibly empty) lists of values. On legacy servers which don’t understand the CAPABILITIES command, an empty dictionary is returned instead.

>>> s = NNTP('news.gmane.org')
>>> 'POST' in s.getcapabilities()
True

New in version 3.2.

NNTP.login(user=None, password=None, usenetrc=True)
Send AUTHINFO commands with the user name and password. If user and password are None and usenetrc is True, credentials from ~/.netrc will be used if possible. Unless intentionally delayed, login is normally performed during the NNTP object initialization and separately calling this function is unnecessary. To force authentication to be delayed, you must not set user or password when creating the object, and must set usenetrc to False. New in version 3.2.

NNTP.starttls(ssl_context=None)
Send a STARTTLS command. The ssl_context argument is optional and should be a ssl.SSLContext object. This will enable encryption on the NNTP connection.

Note that this may not be done after authentication information has been transmitted, and authentication occurs by default if possible during a NNTP object initialization. See NNTP.login() for information on suppressing this behavior. New in version 3.2.

NNTP.newgroups(date, *, file=None)
Send a NEWGROUPS command. The date argument should be a datetime.date or datetime.datetime object. Return a pair (response, groups) where groups is a list representing the groups that are new since the given date. If file is supplied, though, then groups will be empty.

>>> from datetime import date, timedelta
>>> resp, groups = s.newgroups(date.today() - timedelta(days=3))
>>> len(groups)
85
>>> groups[0]
GroupInfo(group='gmane.network.tor.devel', last='4', first='1', flag='m')

NNTP.newnews(group, date, *, file=None)
Send a NEWNEWS command. Here, group is a group name or ‘*’, and date has the same meaning as for newgroups(). Return a pair (response, articles) where articles is a list of message ids.
This command is frequently disabled by NNTP server administrators.

**NNTP.list** *(group_pattern=None, *, file=None)*

Send a LIST or LIST ACTIVE command. Return a pair `(response, list)` where `list` is a list of tuples representing all the groups available from this NNTP server, optionally matching the pattern string `group_pattern`. Each tuple has the form `(group, last, first, flag)`, where `group` is a group name, `last` and `first` are the last and first article numbers, and `flag` usually takes one of these values:

- **y**: Local postings and articles from peers are allowed.
- **m**: The group is moderated and all postings must be approved.
- **n**: No local postings are allowed, only articles from peers.
- **j**: Articles from peers are filed in the junk group instead.
- **x**: No local postings, and articles from peers are ignored.
- **=foo.bar**: Articles are filed in the `foo.bar` group instead.

If `flag` has another value, then the status of the newsgroup should be considered unknown.

This command can return very large results, especially if `group_pattern` is not specified. It is best to cache the results offline unless you really need to refresh them. Changed in version 3.2: `group_pattern` was added.

**NNTP.descriptions** *(grouppattern)*

Send a LIST NEWSGROUPS command, where `grouppattern` is a wildmat string as specified in RFC 3977 (it's essentially the same as DOS or UNIX shell wildcard strings). Return a pair `(response, descriptions)`, where `descriptions` is a dictionary mapping group names to textual descriptions.

```python
gmane.comp.python.*
```

```python
295
```

```python
('gmane.comp.python.bio.general', 'BioPython discussion list (Moderated)')
```

**NNTP.description** *(group)*

Get a description for a single group `group`. If more than one group matches (if ‘group’ is a real wildmat string), return the first match. If no group matches, return an empty string.

This elides the response code from the server. If the response code is needed, use `descriptions()`.

**NNTP.group** *(name)*

Send a GROUP command, where `name` is the group name. The group is selected as the current group, if it exists. Return a tuple `(response, count, first, last, name)` where `count` is the (estimated) number of articles in the group, `first` is the first article number in the group, `last` is the last article number in the group, and `name` is the group name.

**NNTP.over** *(message_spec, *, file=None)*

Send a OVER command, or a XOVER command on legacy servers. `message_spec` can be either a string representing a message id, or a `(first, last)` tuple of numbers indicating a range of articles in the current group, or a `(first, None)` tuple indicating a range of articles starting from `first` to the last article in the current group, or `None` to select the current article in the current group.

Return a pair `(response, overviews)`. `overviews` is a list of `(article_number, overview)` tuples, one for each article selected by `message_spec`. Each `overview` is a dictionary with the same number of items, but this number depends on the server. These items are either message headers (the key is then the lower-cased header name) or metadata items (the key is then the metadata name prepended with "@:")]. The following items are guaranteed to be present by the NNTP specification:

- **the subject, from, date, message-id and references headers**
- **the :bytes metadata: the number of bytes in the entire raw article (including headers and body)**
- **the :lines metadata: the number of lines in the article body**
The value of each item is either a string, or `None` if not present.

It is advisable to use the `decode_header()` function on header values when they may contain non-ASCII characters:

```python
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, overviews = s.over((last, last))
>>> art_num, over = overviews[0]
>>> art_num
117216
>>> list(over.keys())
['xref', 'from', ':lines', ':bytes', 'references', 'date', 'message-id', 'subject']
>>> over['from']
'=?UTF-8?B?Ik1hcnRpbiB2LiBMw7Z3aXMi?= <martin@v.loewis.de>'
>>> nntplib.decode_header(over['from'])
'"Martin v. Löwis" <martin@v.loewis.de>'
```

New in version 3.2.

**NNTP`help`(*, `file=None`)**

Send a HELP command. Return a pair `(response, list)` where `list` is a list of help strings.

**NNTP`stat`(message_spec=None)**

Send a STAT command, where `message_spec` is either a message id (enclosed in `<` and `>`) or an article number in the current group. If `message_spec` is omitted or `None`, the current article in the current group is considered. Return a triple `(response, number, id)` where `number` is the article number and `id` is the message id.

```python
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, number, message_id = s.stat(first)
>>> number, message_id
(9099, '<20030112190404.GE29873@epoch.metaslash.com>')
```

**NNTP`next`()**

Send a NEXT command. Return as for `stat()`.

**NNTP`last`()**

Send a LAST command. Return as for `stat()`.

**NNTP`article`(message_spec=None, *, `file=None`)**

Send an ARTICLE command, where `message_spec` has the same meaning as for `stat()`. Return a tuple `(response, info)` where `info` is a namedtuple with three attributes `number`, `message_id` and `lines` (in that order). `number` is the article number in the group (or 0 if the information is not available), `message_id` the message id as a string, and `lines` a list of lines (without terminating newlines) comprising the raw message including headers and body.

```python
>>> resp, info = s.article('<20030112190404.GE29873@epoch.metaslash.com>')
>>> info.number
0
>>> info.message_id
'<20030112190404.GE29873@epoch.metaslash.com>'
>>> len(info.lines)
65
>>> info.lines[0]
b'Path: main.gmane.org!not-for-mail'
>>> info.lines[1]
b'From: Neal Norwitz <neal@metaslash.com>'
>>> info.lines[-3:]
[b'There is a patch for 2.3 as well as 2.2.', b'', b'Neal']
```
NNTP . **head** (message_spec=None, *, file=None)

Same as **article**(), but sends a **HEAD** command. The lines returned (or written to **file**) will only contain the message headers, not the body.

NNTP . **body** (message_spec=None, *, file=None)

Same as **article**(), but sends a **BODY** command. The lines returned (or written to **file**) will only contain the message body, not the headers.

NNTP . **post** (data)

Post an article using the **POST** command. The **data** argument is either a **file object** opened for binary reading, or any iterable of bytes objects (representing raw lines of the article to be posted). It should represent a well-formed news article, including the required headers. The **post()** method automatically escapes lines beginning with . and appends the termination line.

If the method succeeds, the server’s response is returned. If the server refuses posting, a **NNTPReplyError** is raised.

NNTP . **ihave** (message_id, data)

Send an **IHAVE** command. **message_id** is the id of the message to send to the server (enclosed in ’<’ and ’>’). The **data** parameter and the return value are the same as for **post()**.

NNTP . **date** ()

Return a pair (response, **date**). **date** is a **datetime** object containing the current date and time of the server.

NNTP . **slave** ()

Send a **SLAVE** command. Return the server’s **response**.

NNTP . **set_debuglevel** (level)

Set the instance’s debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request or response. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the connection (including message text).

The following are optional NNTP extensions defined in RFC 2980. Some of them have been superseded by newer commands in RFC 3977.

NNTP . **xhdr** (hdr, str, *, file=None)

Send an **XHDR** command. The **hdr** argument is a header keyword, e.g. ‘subject’. The **str** argument should have the form ’first-last’ where first and last are the first and last article numbers to search. Return a pair (response, list), where list is a list of pairs (id, text), where id is an article number (as a string) and text is the text of the requested header for that article. If the **file** parameter is supplied, then the output of the XHDR command is stored in a file. If **file** is a string, then the method will open a file with that name, write to it then close it. If **file** is a **file object**, then it will start calling **write()** on it to store the lines of the command output. If **file** is supplied, then the returned **list** is an empty list.

NNTP . **xover** (start, end, *, file=None)

Send an **XOVER** command. **start** and **end** are article numbers delimiting the range of articles to select. The return value is the same of for **over()**. It is recommended to use **over()** instead, since it will automatically use the newer **OVER** command if available.

NNTP . **xpath** (id)

Return a pair (resp, **path**), where **path** is the directory path to the article with message ID **id**. Most of the time, this extension is not enabled by NNTP server administrators. Deprecated since version 3.3: The XPATH extension is not actively used.

### 21.16.2 Utility functions

The module also defines the following utility function:

**ntplib**.decode_header (header_str)

Decode a header value, un-escaping any escaped non-ASCII characters. **header_str** must be a str object.
The unescaped value is returned. Using this function is recommended to display some headers in a human readable form:

```python
>>> decode_header("Some subject")
'Some subject'
```

```python
'Débuter en Python'
```

```python
>>> decode_header("Re: =?UTF-8?B?cHJvYmzDqG1lIGRlIG1hdHJpY2U=?=")
'Re: problème de matrice'
```

## 21.17 smtplib — SMTP protocol client

**Source code:** Lib/smtplib.py

The `smtplib` module defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon. For details of SMTP and ESMTP operation, consult RFC 821 (Simple Mail Transfer Protocol) and RFC 1869 (SMTP Service Extensions).

The `smtplib.SMTP` class supports the `with` statement. When used like this, the SMTP QUIT command is issued automatically when the `with` statement exits. E.g.:

```python
>>> from smtplib import SMTP
>>> with SMTP("domain.org") as smtp:
...     smtp.noop()
... (250, b‘Ok‘)
```

```
changed in version 3.3: Support for the with statement was added.
```

The `smtplib.SMTP_SSL` class behaves exactly the same as instances of `smtplib.SMTP`. `smtplib.SMTP_SSL` should be used for situations where SSL is required from the beginning of the connection and using `starttls()` is not appropriate. If `host` is not specified, the local host is used. If `port` is zero, the standard SMTP-over-SSL port (465) is used. The optional arguments `local_hostname` and `source_address` have the same meaning as they do in the `smtplib.SMTP` class. `keyfile` and `certfile` are also optional, and can contain a PEM formatted private key and certificate chain file for the SSL connection. `context` also optional, can contain a SSLContext, and is an alternative to `keyfile` and `certfile`. If it is specified both `keyfile` and `certfile` must be None. The optional `timeout` parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not
specified, the global default timeout setting will be used). The optional source_address parameter allows
this to bind to some specific source address in a machine with multiple network interfaces, and/or to some
specific source tcp port. It takes a 2-tuple (host, port), for the socket to bind to as its source address before
connecting. If omitted (or if host or port are ”’ and/or 0 respectively) the OS default behavior will be used.
Changed in version 3.3: context was added.Changed in version 3.3: source_address argument was added.

```python
class smtplib.LMTP (host='', port=LMTP_PORT, local_hostname=None, source_address=None)
```
The LMTP protocol, which is very similar to ESMTP, is heavily based on the standard SMTP client. It’s
common to use Unix sockets for LMTP, so our connect () method must support that as well as a regular
host:port server. The optional arguments local_hostname and source_address have the same meaning as
they do in the SMTP class. To specify a Unix socket, you must use an absolute path for host, starting with a
’/’.

Authentication is supported, using the regular SMTP mechanism. When using a Unix socket, LMTP
generally don’t support or require any authentication, but your mileage might vary.

A nice selection of exceptions is defined as well:

```python
exception smtplib.SMTPException
    The base exception class for all the other exceptions provided by this module.

exception smtplib.SMTPServerDisconnected
    This exception is raised when the server unexpectedly disconnects, or when an attempt is made to use the
    SMTP instance before connecting it to a server.

exception smtplib.SMTPResponseException
    Base class for all exceptions that include an SMTP error code. These exceptions are generated in some
    instances when the SMTP server returns an error code. The error code is stored in the smtp_code attribute
    of the error, and the smtp_error attribute is set to the error message.

exception smtplib.SMTPSenderRefused
    Sender address refused. In addition to the attributes set by on all SMTPResponseException exceptions,
    this sets ‘sender’ to the string that the SMTP server refused.

exception smtplib.SMTPRecipientsRefused
    All recipient addresses refused. The errors for each recipient are accessible through the attribute
    recipients, which is a dictionary of exactly the same sort as SMTP.sendmail() returns.

exception smtplib.SMTPDataError
    The SMTP server refused to accept the message data.

exception smtplib.SMTPConnectError
    Error occurred during establishment of a connection with the server.

exception smtplib.SMTPHeloError
    The server refused our HELO message.

exception smtplib.SMTPAuthenticationError
    SMTP authentication went wrong. Most probably the server didn’t accept the username/password combi-
    nation provided.
```

See Also:

- **RFC 821 - Simple Mail Transfer Protocol** Protocol definition for SMTP. This document covers the model, op-
  erating procedure, and protocol details for SMTP.

- **RFC 1869 - SMTP Service Extensions** Definition of the ESMTP extensions for SMTP. This describes a frame-
  work for extending SMTP with new commands, supporting dynamic discovery of the commands provided
  by the server, and defines a few additional commands.

### 21.17.1 SMTP Objects

An SMTP instance has the following methods:
**SMTP.set_debuglevel(level)**

Set the debug output level. A true value for level results in debug messages for connection and for all messages sent to and received from the server.

**SMTP.docmd(cmd, args='')**

Send a command cmd to the server. The optional argument args is simply concatenated to the command, separated by a space.

This returns a 2-tuple composed of a numeric response code and the actual response line (multiline responses are joined into one long line.)

In normal operation it should not be necessary to call this method explicitly. It is used to implement other methods and may be useful for testing private extensions.

If the connection to the server is lost while waiting for the reply, `SMTPServerDisconnected` will be raised.

**SMTP.connect(host='localhost', port=0)**

Connect to a host on a given port. The defaults are to connect to the local host at the standard SMTP port (25). If the hostname ends with a colon (':' followed by a number, that suffix will be stripped off and the number interpreted as the port number to use. This method is automatically invoked by the constructor if a host is specified during instantiation. Returns a 2-tuple of the response code and message sent by the server in its connection response.

**SMTP.helo(name='')**

Identify yourself to the SMTP server using HELO. The hostname argument defaults to the fully qualified domain name of the local host. The message returned by the server is stored as the `helo_resp` attribute of the object.

In normal operation it should not be necessary to call this method explicitly. It will be implicitly called by the `sendmail()` when necessary.

**SMTP.ehlo(name='')**

Identify yourself to an ESMTP server using EHLO. The hostname argument defaults to the fully qualified domain name of the local host. Examine the response for ESMTP option and store them for use by `has_extn()`. Also sets several informational attributes: the message returned by the server is stored as the `ehlo_resp` attribute, `does_esmtp` is set to true or false depending on whether the server supports ESMTP, and `esmtp_features` will be a dictionary containing the names of the SMTP service extensions this server supports, and their parameters (if any).

Unless you wish to use `has_extn()` before sending mail, it should not be necessary to call this method explicitly. It will be implicitly called by `sendmail()` when necessary.

**SMTP.ehlo_or_helo_if_needed()**

This method call `ehlo()` and or `helo()` if there has been no previous EHLO or HELO command this session. It tries ESMTP EHLO first.

**SMTPHeloError** The server didn’t reply properly to the HELO greeting.

**SMTP.has_extn(name)**

Return `True` if name is in the set of SMTP service extensions returned by the server, `False` otherwise. Case is ignored.

**SMTP.verify(address)**

Check the validity of an address on this server using SMTP VRFY. Returns a tuple consisting of code 250 and a full RFC 822 address (including human name) if the user address is valid. Otherwise returns an SMTP error code of 400 or greater and an error string.

**Note:** Many sites disable SMTP VRFY in order to foil spammers.

**SMTP.login(user, password)**

Log in on an SMTP server that requires authentication. The arguments are the username and the password to authenticate with. If there has been no previous EHLO or HELO command this session, this method tries
ESMTP EHLO first. This method will return normally if the authentication was successful, or may raise the following exceptions:

- **SMTPHeloError**: The server didn’t reply properly to the HELO greeting.
- **SMTPAuthenticationError**: The server didn’t accept the username/password combination.
- **SMTPException**: No suitable authentication method was found.

**SMTP.starttls(keyfile=None, certfile=None, context=None)**

Put the SMTP connection in TLS (Transport Layer Security) mode. All SMTP commands that follow will be encrypted. You should then call `ehlo()` again.

If `keyfile` and `certfile` are provided, these are passed to the `socket` module’s `ssl()` function.

Optional `context` parameter is a `ssl.SSLContext` object; This is an alternative to using a keyfile and a certfile and if specified both `keyfile` and `certfile` should be `None`.

If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first. If the server does ESMTP, message size and each of the specified options will be passed to it (if the option is in the feature set the server advertises). If EHLO fails, HELO will be tried and ESMTP options suppressed.

This method will return normally if the mail is accepted for at least one recipient. Otherwise it will raise an exception. That is, if this method does not raise an exception, then someone should get your mail. If this method does not raise an exception, it returns a dictionary, with one entry for each recipient that was refused. Each entry contains a tuple of the SMTP error code and the accompanying error message sent by the server.

This method may raise the following exceptions:

- **SMTPRecipientsRefused**: All recipients were refused. Nobody got the mail. The `recipients` attribute of the exception object is a dictionary with information about the refused recipients (like the one returned when at least one recipient was accepted).
- **SMTPHeloError**: The server didn’t reply properly to the HELO greeting.
- **SMTPSenderRefused**: The server didn’t accept the `from_addr`.
- **SMTPDataError**: The server replied with an unexpected error code (other than a refusal of a recipient).

Note: The `from_addr` and `to_addr` parameters are used to construct the message envelope used by the transport agents. `sendmail` does not modify the message headers in any way.

`msg` may be a string containing characters in the ASCII range, or a byte string. A string is encoded to bytes using the ascii codec, and lone \r and \n characters are converted to \r\n characters. A byte string is not modified.

If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first. If the server does ESMTP, message size and each of the specified options will be passed to it (if the option is in the feature set the server advertises). If EHLO fails, HELO will be tried and ESMTP options suppressed.

This method will return normally if the mail is accepted for at least one recipient. Otherwise it will raise an exception. That is, if this method does not raise an exception, then someone should get your mail. If this method does not raise an exception, it returns a dictionary, with one entry for each recipient that was refused. Each entry contains a tuple of the SMTP error code and the accompanying error message sent by the server.

This method may raise the following exceptions:

- **SMTPRecipientsRefused**: All recipients were refused. Nobody got the mail. The `recipients` attribute of the exception object is a dictionary with information about the refused recipients (like the one returned when at least one recipient was accepted).
- **SMTPHeloError**: The server didn’t reply properly to the HELO greeting.
- **SMTPSenderRefused**: The server didn’t accept the `from_addr`.
- **SMTPDataError**: The server replied with an unexpected error code (other than a refusal of a recipient).

Note: Unless otherwise noted, the connection will be open even after an exception is raised. Changed in version 3.2: `msg` may be a byte string.
The Python Library Reference, Release 3.3.3

**SMTP.send_message** *(msg, from_addr=None, to_addr=None, mail_options=[], rcpt_options=[])*

This is a convenience method for calling sendmail() with the message represented by an email.message.Message object. The arguments have the same meaning as for sendmail(), except that msg is a Message object.

If from_addr is None or to_addr is None, send_message fills those arguments with addresses extracted from the headers of msg as specified in RFC 2822: from_addr is set to the Sender field if it is present, and otherwise to the From field. to_addresses combines the values (if any) of the To, Cc, and Bcc fields from msg. If exactly one set of Resent-* headers appear in the message, the regular headers are ignored and the Resent-* headers are used instead. If the message contains more than one set of Resent-* headers, a ValueError is raised, since there is no way to unambiguously detect the most recent set of Resent-headers.

send_message serializes msg using BytesGenerator with \r\n as the linesep, and calls sendmail() to transmit the resulting message. Regardless of the values of from_addr and to_addr, send_message does not transmit any Bcc or Resent-Bcc headers that may appear in msg. New in version 3.2.

**SMTP.quit** ()

Terminate the SMTP session and close the connection. Return the result of the SMTP QUIT command.

Low-level methods corresponding to the standard SMTP/ESMTP commands HELP, RSET, NOOP, MAIL, RCPT, and DATA are also supported. Normally these do not need to be called directly, so they are not documented here. For details, consult the module code.

### 21.17.2 SMTP Example

This example prompts the user for addresses needed in the message envelope ('To' and 'From' addresses), and the message to be delivered. Note that the headers to be included with the message must be included in the message as entered; this example doesn’t do any processing of the RFC 822 headers. In particular, the ‘To’ and ‘From’ addresses must be included in the message headers explicitly.

```python
import smtplib

def prompt(prompt):
    return input(prompt).strip()

fromaddr = prompt("From: ")
toaddrs = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")

# Add the From: and To: headers at the start!
msg = ("From: %s\nTo: %s\r\n\r\n" % (fromaddr, "", ".join(toaddrs)))

while True:
    try:
        line = input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line

print("Message length is", len(msg))

server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
sender.quit()
```
Note: In general, you will want to use the email package’s features to construct an email message, which you can then send via send_message(); see email: Examples.

21.18 smtpd — SMTP Server

Source code: Lib/smtpd.py

This module offers several classes to implement SMTP (email) servers. Several server implementations are present; one is a generic do-nothing implementation, which can be overridden, while the other two offer specific mail-sending strategies. Additionally the SMTPChannel may be extended to implement very specific interaction behaviour with SMTP clients.

The code supports RFC 5321, plus the RFC 1870 SIZE extension.

21.18.1 SMTPServer Objects

class smtpd.SMTPServer (localaddr, remoteaddr, data_size_limit=33554432)

Create a new SMTPServer object, which binds to local address localaddr. It will treat remoteaddr as an upstream SMTP relayer. It inherits from asyncore.dispatcher, and so will insert itself into asyncore's event loop on instantiation.

data_size_limit specifies the maximum number of bytes that will be accepted in a DATA command. A value of None or 0 means no limit.

process_message (peer, mailfrom, rcpttos, data)

Raise NotImplementedError exception. Override this in subclasses to do something useful with this message. Whatever was passed in the constructor as remoteaddr will be available as the _remoteaddr attribute. peer is the remote host’s address, mailfrom is the envelope originator, rcpttos are the envelope recipients and data is a string containing the contents of the e-mail (which should be in RFC 2822 format).

channel_class

Override this in subclasses to use a custom SMTPChannel for managing SMTP clients.

21.18.2 DebuggingServer Objects

class smtpd.DebuggingServer (localaddr, remoteaddr)

Create a new debugging server. Arguments are as per SMTPServer. Messages will be discarded, and printed on stdout.

21.18.3 PureProxy Objects

class smtpd.PureProxy (localaddr, remoteaddr)

Create a new pure proxy server. Arguments are as per SMTPServer. Everything will be relayed to remoteaddr. Note that running this has a good chance to make you into an open relay, so please be careful.
21.18.4 MailmanProxy Objects

class smtpd.MailmanProxy(localaddr, remoteaddr)
Create a new pure proxy server. Arguments are as per SMPServer. Everything will be relayed to remoteaddr, unless local mailman configurations knows about an address, in which case it will be handled via mailman. Note that running this has a good chance to make you into an open relay, so please be careful.

21.18.5 SMTPChannel Objects

class smtpd.SMTPChannel(server, conn, addr)
Create a new SMTPChannel object which manages the communication between the server and a single SMTP client.

To use a custom SMTPChannel implementation you need to override the SMPServer.channel_class of your SMPServer.

The SMTPChannel has the following instance variables:

smtp_server
   Holds the SMPServer that spawned this channel.

conn
   Holds the socket object connecting to the client.

addr
   Holds the address of the client, the second value returned by socket.accept

received_lines
   Holds a list of the line strings (decoded using UTF-8) received from the client. The lines have their "\r\n" line ending translated to "\n".

smtp_state
   Holds the current state of the channel. This will be either COMMAND initially and then DATA after the client sends a “DATA” line.

seen_greeting
   Holds a string containing the greeting sent by the client in its “HELO”.

mailfrom
   Holds a string containing the address identified in the “MAIL FROM:” line from the client.

rcpttos
   Holds a list of strings containing the addresses identified in the “RCPT TO:” lines from the client.

received_data
   Holds a string containing all of the data sent by the client during the DATA state, up to but not including the terminating "\r\n.\r\n".

fqdn
   Holds the fully-qualified domain name of the server as returned by socket.getfqdn().

peer
   Holds the name of the client peer as returned by conn.getpeername() where conn is conn.

The SMTPChannel operates by invoking methods named smtp_<command> upon reception of a command line from the client. Built into the base SMTPChannel class are methods for handling the following commands (and responding to them appropriately):
<table>
<thead>
<tr>
<th>Command</th>
<th>Action taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>HELO</td>
<td>Accepts the greeting from the client and stores it in <code>seen_greeting</code>. Sets server to base command mode.</td>
</tr>
<tr>
<td>EHLO</td>
<td>Accepts the greeting from the client and stores it in <code>seen_greeting</code>. Sets server to extended command mode.</td>
</tr>
<tr>
<td>NOOP</td>
<td>Takes no action.</td>
</tr>
<tr>
<td>QUIT</td>
<td>Closes the connection cleanly.</td>
</tr>
<tr>
<td>MAIL</td>
<td>Accepts the “MAIL FROM:” syntax and stores the supplied address as <code>mailfrom</code>. In extended command mode, accepts the <code>RFC 1870 SIZE</code> attribute and responds appropriately based on the value of <code>data_size_limit</code>.</td>
</tr>
<tr>
<td>RCPT</td>
<td>Accepts the “RCPT TO:” syntax and stores the supplied addresses in the <code>rcpttos</code> list.</td>
</tr>
<tr>
<td>RSET</td>
<td>Resets the <code>mailfrom</code>, <code>rcpttos</code>, and <code>received_data</code>, but not the greeting.</td>
</tr>
<tr>
<td>DATA</td>
<td>Sets the internal state to DATA and stores remaining lines from the client in <code>received_data</code> until the terminator <code>&quot;\r\n.\r\n&quot;</code> is received.</td>
</tr>
<tr>
<td>HELP</td>
<td>Returns minimal information on command syntax</td>
</tr>
<tr>
<td>VRFY</td>
<td>Returns code 252 (the server doesn’t know if the address is valid)</td>
</tr>
<tr>
<td>EXPN</td>
<td>Reports that the command is not implemented.</td>
</tr>
</tbody>
</table>

## 21.19 telnetlib — Telnet client

**Source code:** Lib/telnetlib.py

The `telnetlib` module provides a Telnet class that implements the Telnet protocol. See RFC 854 for details about the protocol. In addition, it provides symbolic constants for the protocol characters (see below), and for the telnet options. The symbolic names of the telnet options follow the definitions in `arpa/telnet.h`, with the leading `TELOPT_` removed. For symbolic names of options which are traditionally not included in `arpa/telnet.h`, see the module source itself.

The symbolic constants for the telnet commands are: IAC, DONT, DO, WONT, WILL, SE (Subnegotiation End), NOP (No Operation), DM (Data Mark), BRK (Break), IP (Interrupt process), AO (Abort output), AYT (Are You There), EC (Erase Character), EL (Erase Line), GA (Go Ahead), SB (Subnegotiation Begin).

```python
class telnetlib.Telnet(host=None, port=0[, timeout])
```

`Telnet` represents a connection to a Telnet server. The instance is initially not connected by default; the `open()` method must be used to establish a connection. Alternatively, the host name and optional port number can be passed to the constructor, to, in which case the connection to the server will be established before the constructor returns. The optional `timeout` parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).

Do not reopen an already connected instance.

This class has many `read_*()` methods. Note that some of them raise `EOFError` when the end of the connection is read, because they can return an empty string for other reasons. See the individual descriptions below.

**See Also:**


### 21.19.1 Telnet Objects

`Telnet` instances have the following methods:

```python
telnetlib.Telnet.read_until(expected, timeout=None)
```

Read until a given byte string, `expected`, is encountered or until `timeout` seconds have passed.

When no match is found, return whatever is available instead, possibly empty bytes. Raise `EOFError` if the connection is closed and no cooked data is available.
Telnet.read_all()
    Read all data until EOF as bytes; block until connection closed.

Telnet.read_some()
    Read at least one byte of cooked data unless EOF is hit. Return b"" if EOF is hit. Block if no data is immediately available.

Telnet.read_very_eager()
    Read everything that can be without blocking in I/O (eager).
    Raise EOFError if connection closed and no cooked data available. Return b"" if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

Telnet.read_eager()
    Read readily available data.
    Raise EOFError if connection closed and no cooked data available. Return b"" if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

Telnet.read_lazy()
    Process and return data already in the queues (lazy).
    Raise EOFError if connection closed and no data available. Return b"" if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

Telnet.read_very_lazy()
    Return any data available in the cooked queue (very lazy).
    Raise EOFError if connection closed and no data available. Return b"" if no cooked data available otherwise. This method never blocks.

Telnet.read_sb_data()
    Return the data collected between a SB/SE pair (suboption begin/end). The callback should access these data when it was invoked with a SE command. This method never blocks.

Telnet.open(host, port=0, timeout=)
    Connect to a host. The optional second argument is the port number, which defaults to the standard Telnet port (23). The optional timeout parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).
    Do not try to reopen an already connected instance.

Telnet.msg(msg, *args)
    Print a debug message when the debug level is > 0. If extra arguments are present, they are substituted in the message using the standard string formatting operator.

Telnet.set_debuglevel(debuglevel)
    Set the debug level. The higher the value of debuglevel, the more debug output you get (on sys.stdout).

Telnet.close()
    Close the connection.

Telnet.get_socket()
    Return the socket object used internally.

Telnet.fileno()
    Return the file descriptor of the socket object used internally.

Telnet.write(buffer)
    Write a byte string to the socket, doubling any IAC characters. This can block if the connection is blocked. May raise OSError if the connection is closed. Changed in version 3.3: This method used to raise socket.error, which is now an alias of OSError.

Telnet.interact()
    Interaction function, emulates a very dumb Telnet client.

Telnet.mt_interact()
    Multithreaded version of interact().
Telnet.expect(list, timeout=None)

Read until one from a list of a regular expressions matches.

The first argument is a list of regular expressions, either compiled (regex objects) or uncompiled (byte strings). The optional second argument is a timeout, in seconds; the default is to block indefinitely.

Return a tuple of three items: the index in the list of the first regular expression that matches; the match object returned; and the bytes read up till and including the match.

If end of file is found and no bytes were read, raise EOFError. Otherwise, when nothing matches, return (-1, None, data) where data is the bytes received so far (may be empty bytes if a timeout happened).

If a regular expression ends with a greedy match (such as .*) or if more than one expression can match the same input, the results are non-deterministic, and may depend on the I/O timing.

Telnet.set_option_negotiation_callback(callback)

Each time a telnet option is read on the input flow, this callback (if set) is called with the following parameters: callback(telnet socket, command (DO/DONT/WILL/WONT), option). No other action is done afterwards by telnetlib.

21.19.2 Telnet Example

A simple example illustrating typical use:

```python
import getpass
import telnetlib

HOST = "localhost"
user = input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)

tn.read_until(b"login: ")
if password:
    tn.read_until(b"Password: ")
    tn.write(password.encode('ascii') + b"\n")

tn.write(b"ls\n")

print(tn.read_all().decode('asci'))
```

21.20 uuid — UUID objects according to RFC 4122

This module provides immutable UUID objects (the UUID class) and the functions uuid1(), uuid3(), uuid4(), uuid5() for generating version 1, 3, 4, and 5 UUIDs as specified in RFC 4122.

If all you want is a unique ID, you should probably call uuid1() or uuid4(). Note that uuid1() may compromise privacy since it creates a UUID containing the computer’s network address. uuid4() creates a random UUID.

class uuid.UUID(hex=None, bytes=None, bytes_le=None, fields=None, int=None, version=None)

Create a UUID from either a string of 32 hexadecimal digits, a string of 16 bytes as the bytes argument, a string of 16 bytes in little-endian order as the bytes_le argument, a tuple of six integers (32-bit time_low, 16-bit time_mid, 16-bit time Hi_version, 8-bit clock_seq_hi_variant, 8-bit clock_seq_low, 48-bit node) as the fields argument, or a single 128-bit integer as the int argument. When a string of hex digits is given, curly braces, hyphens, and a URN prefix are all optional. For example, these expressions all yield the same UUID:
Exactly one of `hex`, `bytes`, `bytes_le`, `fields`, or `int` must be given. The `version` argument is optional; if given, the resulting UUID will have its variant and version number set according to RFC 4122, overriding bits in the given `hex`, `bytes`, `bytes_le`, `fields`, or `int`.

**UUID** instances have these read-only attributes:

**UUID.** `bytes`

The UUID as a 16-byte string (containing the six integer fields in big-endian byte order).

**UUID.** `bytes_le`

The UUID as a 16-byte string (with `time_low`, `time_mid`, and `time_hi_version` in little-endian byte order).

**UUID.** `fields`

A tuple of the six integer fields of the UUID, which are also available as six individual attributes and two derived attributes:

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>time_low</code></td>
<td>the first 32 bits of the UUID</td>
</tr>
<tr>
<td><code>time_mid</code></td>
<td>the next 16 bits of the UUID</td>
</tr>
<tr>
<td><code>time_hi_version</code></td>
<td>the next 16 bits of the UUID</td>
</tr>
<tr>
<td><code>clock_seq_hi_variant</code></td>
<td>the next 8 bits of the UUID</td>
</tr>
<tr>
<td><code>clock_seq_low</code></td>
<td>the last 48 bits of the UUID</td>
</tr>
<tr>
<td><code>node</code></td>
<td>the last 48 bits of the UUID</td>
</tr>
<tr>
<td><code>time</code></td>
<td>the 60-bit timestamp</td>
</tr>
<tr>
<td><code>clock_seq</code></td>
<td>the 14-bit sequence number</td>
</tr>
</tbody>
</table>

**UUID.** `hex`

The UUID as a 32-character hexadecimal string.

**UUID.** `int`

The UUID as a 128-bit integer.

**UUID.** `urn`

The UUID as a URN as specified in RFC 4122.

**UUID.** `variant`

The UUID variant, which determines the internal layout of the UUID. This will be one of the integer constants `RESERVED_NCS`, `RFC_4122`, `RESERVED_MICROSOFT`, or `RESERVED_FUTURE`.

**UUID.** `version`

The UUID version number (1 through 5, meaningful only when the variant is `RFC_4122`).

The `uuid` module defines the following functions:

**uuid.** `getnode()`

Get the hardware address as a 48-bit positive integer. The first time this runs, it may launch a separate program, which could be quite slow. If all attempts to obtain the hardware address fail, we choose a random 48-bit number with its eighth bit set to 1 as recommended in RFC 4122. “Hardware address” means the MAC address of a network interface, and on a machine with multiple network interfaces the MAC address of any one of them may be returned.

**uuid.** `uuid1`(*node=None*, *clock_seq=None*)

Generate a UUID from a host ID, sequence number, and the current time. If `node` is not given, `getnode()` is used to obtain the hardware address. If `clock_seq` is given, it is used as the sequence number; otherwise a random 14-bit sequence number is chosen.
The Python Library Reference, Release 3.3.3

uuid.uuid3 (namespace, name)
Generate a UUID based on the MD5 hash of a namespace identifier (which is a UUID) and a name (which is a string).

uuid.uuid4 ()
Generate a random UUID.

uuid.uuid5 (namespace, name)
Generate a UUID based on the SHA-1 hash of a namespace identifier (which is a UUID) and a name (which is a string).

The uuid module defines the following namespace identifiers for use with uuid3() or uuid5().

uuid.NAMESPACE_DNS
When this namespace is specified, the name string is a fully-qualified domain name.

uuid.NAMESPACE_URL
When this namespace is specified, the name string is a URL.

uuid.NAMESPACE_OID
When this namespace is specified, the name string is an ISO OID.

uuid.NAMESPACE_X500
When this namespace is specified, the name string is an X.500 DN in DER or a text output format.

The uuid module defines the following constants for the possible values of the variant attribute:

uuid.RESERVED_NCS
Reserved for NCS compatibility.

uuid.RFC_4122
Specifies the UUID layout given in RFC 4122.

uuid.RESERVED_MICROSOFT
Reserved for Microsoft compatibility.

uuid.RESERVED_FUTURE
Reserved for future definition.

See Also:
RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace   This specification defines a Uniform Resource Name namespace for UUIDs, the internal format of UUIDs, and methods of generating UUIDs.

21.20.1 Example

Here are some examples of typical usage of the uuid module:

>>> import uuid

>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9ae199e5d')
>>> # make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
>>> x.bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f'

>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')

21.21 socketserver — A framework for network servers

Source code: Lib/socketserver.py

The socketserver module simplifies the task of writing network servers.

There are four basic server classes: TCPServer uses the Internet TCP protocol, which provides for continuous streams of data between the client and server. UDPServer uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The more infrequently used UnixStreamServer and UnixDatagramServer classes are similar, but use Unix domain sockets; they’re not available on non-Unix platforms. For more details on network programming, consult a book such as W. Richard Steven’s UNIX Network Programming or Ralph Davis’s Win32 Network Programming.

These four classes process requests synchronously; each request must be completed before the next request can be started. This isn’t suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process. The solution is to create a separate process or thread to handle each request; the ForkingMixIn and ThreadingMixIn mix-in classes can be used to support asynchronous behaviour.

Creating a server requires several steps. First, you must create a request handler class by subclassing the BaseRequestHandler class and overriding its handle() method; this method will process incoming requests. Second, you must instantiate one of the server classes, passing it the server’s address and the request handler class. Finally, call the handle_request() or serve_forever() method of the server object to process one or many requests.

When inheriting from ThreadingMixIn for threaded connection behavior, you should explicitly declare how you want your threads to behave on an abrupt shutdown. The ThreadingMixIn class defines an attribute daemon_threads, which indicates whether or not the server should wait for thread termination. You should set the flag explicitly if you would like threads to behave autonomously; the default is False, meaning that Python will not exit until all threads created by ThreadingMixIn have exited.

Server classes have the same external methods and attributes, no matter what network protocol they use.

21.21.1 Server Creation Notes

There are five classes in an inheritance diagram, four of which represent synchronous servers of four types:

```
+------------+
| BaseServer |
+------------+
```
Note that `UnixDatagramServer` derives from `UDPServer`, not from `UnixStreamServer` — the only difference between an IP and a Unix stream server is the address family, which is simply repeated in both Unix server classes.

Forking and threading versions of each type of server can be created using the `ForkingMixIn` and `ThreadingMixIn` mix-in classes. For instance, a threading UDP server class is created as follows:

```python
class ThreadingUDPServer(ThreadingMixIn, UDPServer): pass
```

The mix-in class must come first, since it overrides a method defined in `UDPServer`. Setting the various attributes also change the behavior of the underlying server mechanism.

To implement a service, you must derive a class from `BaseRequestHandler` and redefine its `handle()` method. You can then run various versions of the service by combining one of the server classes with your request handler class. The request handler class must be different for datagram or stream services. This can be hidden by using the handler subclasses `StreamRequestHandler` or `DatagramRequestHandler`.

Of course, you still have to use your head! For instance, it makes no sense to use a forking server if the service contains state in memory that can be modified by different requests, since the modifications in the child process would never reach the initial state kept in the parent process and passed to each child. In this case, you can use a threading server, but you will probably have to use locks to protect the integrity of the shared data.

On the other hand, if you are building an HTTP server where all data is stored externally (for instance, in the file system), a synchronous class will essentially render the service “deaf” while one request is being handled — which may be for a very long time if a client is slow to receive all the data it has requested. Here a threading or forking server is appropriate.

In some cases, it may be appropriate to process part of a request synchronously, but to finish processing in a forked child depending on the request data. This can be implemented by using a synchronous server and doing an explicit fork in the request handler class `handle()` method.

Another approach to handling multiple simultaneous requests in an environment that supports neither threads nor `fork()` (or where these are too expensive or inappropriate for the service) is to maintain an explicit table of partially finished requests and to use `select()` to decide which request to work on next (or whether to handle a new incoming request). This is particularly important for stream services where each client can potentially be connected for a long time (if threads or subprocesses cannot be used). See `asyncore` for another way to manage this.

### 21.21.2 Server Objects

```python
class socketserver.BaseServer
    This is the superclass of all Server objects in the module. It defines the interface, given below, but does not implement most of the methods, which is done in subclasses.

BaseServer.fileno()
    Return an integer file descriptor for the socket on which the server is listening. This function is most commonly passed to `select.select()`, to allow monitoring multiple servers in the same process.

BaseServer.handle_request()
    Process a single request. This function calls the following methods in order: `get_request()`, `verify_request()`, and `process_request()`. If the user-provided `handle()` method of the handler class raises an exception, the server's `handle_error()` method will be called. If
no request is received within `self.timeout` seconds, `handle_timeout()` will be called and
`handle_request()` will return.

**BaseServer.** `serve_forever` (*poll_interval=0.5*)

Handle requests until an explicit `shutdown()` request. Poll for shutdown every `poll_interval` sec-
donds. Ignores `self.timeout`. It also calls `service_actions()`, which may be used by a sub-
class or mixin to provide actions specific to a given service. For example, the `ForkingMixIn` class
uses `service_actions()` to clean up zombie child processes. Changed in version 3.3: Added
`service_actions` call to the `serve_forever` method.

**BaseServer.** `service_actions()`

This is called in the `serve_forever()` loop. This method can be overridden by subclasses or mixin
classes to perform actions specific to a given service, such as cleanup actions. New in version 3.3.

**BaseServer.** `shutdown()`

Tell the `serve_forever()` loop to stop and wait until it does.

**BaseServer.** `address_family`

The family of protocols to which the server’s socket belongs. Common examples are `socket.AF_INET`
and `socket.AF_UNIX`.

**BaseServer.** `RequestHandlerClass`

The user-provided request handler class; an instance of this class is created for each request.

**BaseServer.** `server_address`

The address on which the server is listening. The format of addresses varies depending on the protocol
family; see the documentation for the socket module for details. For Internet protocols, this is a tuple
containing a string giving the address, and an integer port number: `('127.0.0.1', 80)`, for example.

**BaseServer.** `socket`

The socket object on which the server will listen for incoming requests.

The server classes support the following class variables:

**BaseServer.** `allow_reuse_address`

Whether the server will allow the reuse of an address. This defaults to `False`, and can be set in subclasses
to change the policy.

**BaseServer.** `request_queue_size`

The size of the request queue. If it takes a long time to process a single request, any requests that arrive
while the server is busy are placed into a queue, up to `request_queue_size` requests. Once the queue
is full, further requests from clients will get a “Connection denied” error. The default value is usually 5, but
this can be overridden by subclasses.

**BaseServer.** `socket_type`

The type of socket used by the server; `socket.SOCK_STREAM` and `socket.SOCK_DGRAM` are two
common values.

**BaseServer.** `timeout`

Timeout duration, measured in seconds, or `None` if no timeout is desired. If `handle_request()` re-
ceives no incoming requests within the timeout period, the `handle_timeout()` method is called.

There are various server methods that can be overridden by subclasses of base server classes like `TCPServer`;
these methods aren’t useful to external users of the server object.

**BaseServer.** `finish_request()`

Actually processes the request by instantiating `RequestHandlerClass` and calling its `handle()`
method.

**BaseServer.** `get_request()`

Must accept a request from the socket, and return a 2-tuple containing the `new` socket object to be used to
communicate with the client, and the client’s address.

**BaseServer.** `handle_error` (*request, client_address*)

This function is called if the `RequestHandlerClass`'s `handle()` method raises an exception. The
default action is to print the traceback to standard output and continue handling further requests.
This function is called when the `timeout` attribute has been set to a value other than `None` and the timeout period has passed with no requests being received. The default action for forking servers is to collect the status of any child processes that have exited, while in threading servers this method does nothing.

**BaseServer.** `process_request` *(request, client_address)*

Calls `finish_request()` to create an instance of the `RequestHandlerClass`. If desired, this function can create a new process or thread to handle the request; the `ForkingMixIn` and `ThreadingMixIn` classes do this.

**BaseServer.** `server_activate` *

Called by the server's constructor to activate the server. The default behavior just `listen()`s to the server's socket. May be overridden.

**BaseServer.** `server_bind` *

Called by the server's constructor to bind the socket to the desired address. May be overridden.

**BaseServer.** `verify_request` *(request, client_address)*

Must return a Boolean value; if the value is `True`, the request will be processed, and if it's `False`, the request will be denied. This function can be overridden to implement access controls for a server. The default implementation always returns `True`.

## 21.21.3 RequestHandler Objects

The request handler class must define a new `handle()` method, and can override any of the following methods. A new instance is created for each request.

**RequestHandler.** `finish()` *

Called after the `handle()` method to perform any clean-up actions required. The default implementation does nothing. If `setup()` raises an exception, this function will not be called.

**RequestHandler.** `handle()` *

This function must do all the work required to service a request. The default implementation does nothing. Several instance attributes are available to it; the request is available as `self.request`; the client address as `self.client_address`; and the server instance as `self.server`, in case it needs access to per-server information.

The type of `self.request` is different for datagram or stream services. For stream services, `self.request` is a socket object; for datagram services, `self.request` is a pair of string and socket. However, this can be hidden by using the request handler subclasses `StreamRequestHandler` or `DatagramRequestHandler`, which override the `setup()` and `finish()` methods, and provide `self.rfile` and `self.wfile` attributes. `self.rfile` and `self.wfile` can be read or written, respectively, to get the request data or return data to the client.

**RequestHandler.** `setup()` *

Called before the `handle()` method to perform any initialization actions required. The default implementation does nothing.

## 21.21.4 Examples

**socketserver.** `TCPServer Example`

This is the server side:

```python
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    The RequestHandler class for our server.
    It is instantiated once per connection to the server, and must
```
override the handle() method to implement communication to the client.

```python
def handle(self):
    # self.request is the TCP socket connected to the client
    self.data = self.request.recv(1024).strip()
    print("{} wrote:",format(self.client_address[0]))
    print(self.data)
    # just send back the same data, but upper-cased
    self.request.sendall(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    server = socketserver.TCPServer((HOST, PORT), MyTCPHandler)

    # Activate the server; this will keep running until you
    # interrupt the program with Ctrl-C
    server.serve_forever()
```

An alternative request handler class that makes use of streams (file-like objects that simplify communication by providing the standard file interface):

```python
class MyTCPHandler(socketserver.StreamRequestHandler):
    def handle(self):
        # self.rfile is a file-like object created by the handler;
        # we can now use e.g. readline() instead of raw recv() calls
        self.data = self.rfile.readline().strip()
        print("{} wrote:",format(self.client_address[0]))
        print(self.data)
        # Likewise, self.wfile is a file-like object used to write back
        # to the client
        self.wfile.write(self.data.upper())
```

The difference is that the readline() call in the second handler will call recv() multiple times until it encounters a newline character, while the single recv() call in the first handler will just return what has been sent from the client in one sendall() call.

This is the client side:

```python
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data + "\n", "utf-8"))

    # Receive data from the server and shut down
    received = str(sock.recv(1024), "utf-8")

finally:
    sock.close()
```
print("Sent: \{\}.format(data))
print("Received: \{\}.format(received))

The output of the example should look something like this:

Server:
$ python TCPServer.py
127.0.0.1 wrote:
b'hello world with TCP'
127.0.0.1 wrote:
b'python is nice'

Client:
$ python TCPClient.py hello world with TCP
Sent: hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent: python is nice
Received: PYTHON IS NICE

socketserver.UDPServer Example

This is the server side:

```python
import socketserver
class MyUDPHandler(socketserver.BaseRequestHandler):
    
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and since
    there is no connection the client address must be given explicitly
    when sending data back via sendto().
    """
    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print("{} wrote:{}.format(self.client_address[0]))
        print(data)
        socket.sendto(data.upper(), self.client_address)
```

```python
if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    server = socketserver.UDPServer((HOST, PORT), MyUDPHandler)
    server.serve_forever()
```

This is the client side:

```python
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no connections.
Instead, data is directly sent to the recipient via sendto().

```python
sock.sendto(bytes(data + '
', 'utf-8'), (HOST, PORT))
received = str(sock.recv(1024), 'utf-8')

print("Sent: {}".format(data))
print("Received: {}".format(received))
```

The output of the example should look exactly like for the TCP server example.

### Asynchronous Mixins

To build asynchronous handlers, use the ThreadMixIn and ForkMixIn classes.

An example for the ThreadMixIn class:

```python
import socket
import threading
import socketserver

class ThreadedTCPRequestHandler(socketserver.BaseRequestHandler):
    def handle(self):
        data = str(self.request.recv(1024), 'ascii')
        cur_thread = threading.current_thread()
        response = bytes('{}: {}'.format(cur_thread.name, data), 'ascii')
        self.request.sendall(response)

class ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
    pass

def client(ip, port, message):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((ip, port))
    try:
        sock.sendall(bytes(message, 'ascii'))
        response = str(sock.recv(1024), 'ascii')
        print("Received: {}".format(response))
    finally:
        sock.close()

if __name__ == '__main__':
    # Port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer(((HOST, PORT), ThreadedTCPRequestHandler)
ip, port = server.server_address

    # Start a thread with the server -- that thread will then start one
    # more thread for each request
    server_thread = threading.Thread(target=server.serve_forever)
    server_thread.daemon = True
    server_thread.start()
    print("Server loop running in thread":, server_thread.name)

    client(ip, port, "Hello World 1")
    client(ip, port, "Hello World 2")
    client(ip, port, "Hello World 3")
```

21.21. `socketserver` — A framework for network servers
server.shutdown()

The output of the example should look something like this:
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3

The ForkingMixIn class is used in the same way, except that the server will spawn a new process for each request.

## 21.22 http.server — HTTP servers

**Source code:** Lib/http/server.py

This module defines classes for implementing HTTP servers (Web servers).

One class, `HTTPServer`, is a `socketserver.TCPServer` subclass. It creates and listens at the HTTP socket, dispatching the requests to a handler. Code to create and run the server looks like this:

```python
def run(server_class=HTTPServer, handler_class=BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()

class http.server.HTTPServer(server_address, RequestHandlerClass)
    This class builds on the TCPServer class by storing the server address as instance variables named server_name and server_port. The server is accessible by the handler, typically through the handler's server instance variable.

    The HTTPServer must be given a RequestHandlerClass on instantiation, of which this module provides three different variants:

class http.server.BaseHTTPRequestHandler(request, client_address, server)
    This class is used to handle the HTTP requests that arrive at the server. By itself, it cannot respond to any actual HTTP requests; it must be subclassed to handle each request method (e.g. GET or POST). BaseHTTPRequestHandler provides a number of class and instance variables, and methods for use by subclasses.

    The handler will parse the request and the headers, then call a method specific to the request type. The method name is constructed from the request. For example, for the request method SPAM, the do_SPAM() method will be called with no arguments. All of the relevant information is stored in instance variables of the handler. Subclasses should not need to override or extend the __init__() method.

    BaseHTTPRequestHandler has the following instance variables:

    client_address
        Contains a tuple of the form (host, port) referring to the client’s address.

    server
        Contains the server instance.

    command
        Contains the command (request type). For example, ‘GET’.

    path
        Contains the request path.

    request_version
        Contains the version string from the request. For example, ‘HTTP/1.0’.
headers
Holds an instance of the class specified by the `MessageClass` class variable. This instance parses and manages the headers in the HTTP request.

rfile
Contains an input stream, positioned at the start of the optional input data.

wfile
Contains the output stream for writing a response back to the client. Proper adherence to the HTTP protocol must be used when writing to this stream.

`BaseHTTPRequestHandler` has the following class variables:

server_version
Specifies the server software version. You may want to override this. The format is multiple whitespace-separated strings, where each string is of the form name[/version]. For example, ‘BaseHTTP/0.2’.

sys_version
Contains the Python system version, in a form usable by the `version_string` method and the `server_version` class variable. For example, ‘Python/1.4’.

error_message_format
Specifies a format string for building an error response to the client. It uses parenthesized, keyed format specifiers, so the format operand must be a dictionary. The `code` key should be an integer, specifying the numeric HTTP error code value. `message` should be a string containing a (detailed) error message of what occurred, and `explain` should be an explanation of the error code number. Default `message` and `explain` values can found in the `responses` class variable.

error_content_type
Specifies the Content-Type HTTP header of error responses sent to the client. The default value is ‘text/html’.

protocol_version
This specifies the HTTP protocol version used in responses. If set to ‘HTTP/1.1’, the server will permit HTTP persistent connections; however, your server must then include an accurate `Content-Length` header (using `send_header()`) in all of its responses to clients. For backwards compatibility, the setting defaults to ‘HTTP/1.0’.

`MessageClass`
Specifies an `email.message.Message`-like class to parse HTTP headers. Typically, this is not overridden, and it defaults to `http.client.HTTPMessage`.

`responses`
This variable contains a mapping of error code integers to two-element tuples containing a short and long message. For example, `(code: (shortmessage, longmessage))`. The `shortmessage` is usually used as the `message` key in an error response, and `longmessage` as the `explain` key (see the `error_message_format` class variable).

A `BaseHTTPRequestHandler` instance has the following methods:

`handle()`
Calls `handle_one_request()` once (or, if persistent connections are enabled, multiple times) to handle incoming HTTP requests. You should never need to override it; instead, implement appropriate `do_*()` methods.

`handle_one_request()`
This method will parse and dispatch the request to the appropriate `do_*()` method. You should never need to override it.

`handle_expect_100()`
When a HTTP/1.1 compliant server receives a `Expect: 100-continue` request header it responds back with a 100 `Continue` followed by 200 `OK` headers. This method can be overridden to raise an error if the server does not want the client to continue. For e.g. server can chose to send 417 Expectation Failed as a response header and return `False`. New in version 3.2.
**send_error**(code, message=None)

Sends and logs a complete error reply to the client. The numeric `code` specifies the HTTP error code, with `message` as optional, more specific text. A complete set of headers is sent, followed by text composed using the `error_message_format` class variable.

**send_response**(code, message=None)

Adds a response header to the headers buffer and logs the accepted request. The HTTP response line is written to the internal buffer, followed by `Server` and `Date` headers. The values for these two headers are picked up from the `version_string()` and `date_time_string()` methods, respectively. If the server does not intend to send any other headers using the `send_header()` method, then `send_response()` should be followed by a `end_headers()` call. Changed in version 3.3: Headers are stored to an internal buffer and `end_headers()` needs to be called explicitly.

**send_header**(keyword, value)

Adds the HTTP header to an internal buffer which will be written to the output stream when either `end_headers()` or `flush_headers()` is invoked. `keyword` should specify the header keyword, with `value` specifying its value. Note that, after the send_header calls are done, `end_headers()` MUST BE called in order to complete the operation. Changed in version 3.2: Headers are stored in an internal buffer.

**send_response_only**(code, message=None)

Sends the reponse header only, used for the purposes when **100 Continue** response is sent by the server to the client. The headers not buffered and sent directly the output stream. If the `message` is not specified, the HTTP message corresponding the response `code` is sent. New in version 3.2.

**end_headers**()

Adds a blank line (indicating the end of the HTTP headers in the response) to the headers buffer and calls `flush_headers()`. Changed in version 3.2: The buffered headers are written to the output stream.

**flush_headers**()

Finally send the headers to the output stream and flush the internal headers buffer. New in version 3.3.

**log_request**(code='-', size='-')

Logs an accepted (successful) request. `code` should specify the numeric HTTP code associated with the response. If a size of the response is available, then it should be passed as the `size` parameter.

**log_error**(...)

Logs an error when a request cannot be fulfilled. By default, it passes the message to `log_message()`, so it takes the same arguments (`format` and additional values).

**log_message**(format, ...)

Logs an arbitrary message to `sys.stderr`. This is typically overridden to create custom error logging mechanisms. The `format` argument is a standard printf-style format string, where the additional arguments to `log_message()` are applied as inputs to the formatting. The client ip address and current date and time are prefixed to every message logged.

**version_string**()

Returns the server software’s version string. This is a combination of the `server_version` and `sys_version` class variables.

**date_time_string**(timestamp=None)

Returns the date and time given by `timestamp` (which must be None or in the format returned by `time.time()`), formatted for a message header. If `timestamp` is omitted, it uses the current date and time.

The result looks like ‘Sun, 06 Nov 1994 08:49:37 GMT’.

**log_date_time_string**()

Returns the current date and time, formatted for logging.

**address_string**()

Returns the client address. Changed in version 3.3: Previously, a name lookup was performed. To avoid name resolution delays, it now always returns the IP address.
class http.server.SimpleHTTPRequestHandler(request, client_address, server)

This class serves files from the current directory and below, directly mapping the directory structure to
HTTP requests.

A lot of the work, such as parsing the request, is done by the base class BaseHTTPRequestHandler. This class implements the do_GET() and do_HEAD() functions.

The following are defined as class-level attributes of SimpleHTTPRequestHandler:

server_version
This will be "SimpleHTTP/" + __version__, where __version__ is defined at the module level.

extensions_map
A dictionary mapping suffixes into MIME types. The default is signified by an empty string, and is
considered to be application/octet-stream. The mapping is used case-insensitively, and so
should contain only lower-cased keys.

The SimpleHTTPRequestHandler class defines the following methods:

do_HEAD()
This method serves the ‘HEAD’ request type: it sends the headers it would send for the equivalent
GET request. See the do_GET() method for a more complete explanation of the possible headers.

do_GET()
The request is mapped to a local file by interpreting the request as a path relative to the current working
directory.

If the request was mapped to a directory, the directory is checked for a file named index.html or
index.htm (in that order). If found, the file’s contents are returned; otherwise a directory listing is
generated by calling the list_directory() method. This method uses os.listdir() to scan
the directory, and returns a 404 error response if the listdir() fails.

If the request was mapped to a file, it is opened and the contents are returned. Any OSError exception
in opening the requested file is mapped to a 404,’File not found’ error. Otherwise, the content type is guessed by calling the guess_type() method, which in turn uses the extensions_map
variable.

A ‘Content-type:’ header with the guessed content type is output, followed by a
’Content-Length:’ header with the file’s size and a ’Last-Modified:’ header with the
file’s modification time.

Then follows a blank line signifying the end of the headers, and then the contents of the file are output.
If the file’s MIME type starts with text/ the file is opened in text mode; otherwise binary mode is
used.

For example usage, see the implementation of the test() function invocation in the http.server
module.

The SimpleHTTPRequestHandler class can be used in the following manner in order to create a very basic
webserver serving files relative to the current directory.

import http.server
import socketserver

PORT = 8000

Handler = http.server.SimpleHTTPRequestHandler

httpd = socketserver.TCPServer(("", PORT), Handler)

print("serving at port", PORT)
httpd.serve_forever()

http.server can also be invoked directly using the -m switch of the interpreter with a port number argument. Similar to the previous example, this serves files relative to the current directory.
python -m http.server 8000

class http.server.CGIHTTPRequestHandler (request, client_address, server)
This class is used to serve either files or output of CGI scripts from the current directory and be-
low. Note that mapping HTTP hierarchic structure to local directory structure is exactly as in
SimpleHTTPRequestHandler.

Note: CGI scripts run by the CGIHTTPRequestHandler class cannot execute redirects (HTTP code
302), because code 200 (script output follows) is sent prior to execution of the CGI script. This pre-empts
the status code.

The class will however, run the CGI script, instead of serving it as a file, if it guesses it to be a CGI script.
Only directory-based CGI are used — the other common server configuration is to treat special extensions
as denoting CGI scripts.

The do_GET() and do_HEAD() functions are modified to run CGI scripts and serve the output, instead
of serving files, if the request leads to somewhere below the cgi_directories path.

The CGIHTTPRequestHandler defines the following data member:

cgi_directories
This defaults to ['/cgi-bin', '/htbin'] and describes directories to treat as containing CGI
scripts.

The CGIHTTPRequestHandler defines the following method:
do_POST()
This method serves the 'POST' request type, only allowed for CGI scripts. Error 501, “Can only
POST to CGI scripts”, is output when trying to POST to a non-CGI url.

Note that CGI scripts will be run with UID of user nobody, for security reasons. Problems with the CGI
script will be translated to error 403.

CGIHTTPRequestHandler can be enabled in the command line by passing the --cgi option:
python -m http.server --cgi 8000

21.23 http.cookies — HTTP state management

Source code: Lib/http/cookies.py

The http.cookies module defines classes for abstracting the concept of cookies, an HTTP state management
mechanism. It supports both simple string-only cookies, and provides an abstraction for having any serializable
data-type as cookie value.

The module formerly strictly applied the parsing rules described in the
RFC 2109 and RFC 2068 specifications. It has since been discovered that MSIE 3.0x doesn’t follow the character
rules outlined in those specs and also many current day browsers and servers have relaxed parsing rules when
comes to Cookie handling. As a result, the parsing rules used are a bit less strict.

The character set, string.ascii_letters, string.digits and !#$%&'*+-.^_'|~: denote the set
of valid characters allowed by this module in Cookie name (as key). Changed in version 3.3: Allowed ':' as a
valid Cookie name character.

Note: On encountering an invalid cookie, CookieError is raised, so if your cookie data comes from a browser
you should always prepare for invalid data and catch CookieError on parsing.

exception http.cookies.CookieError
Exception failing because of RFC 2109 invalidity: incorrect attributes, incorrect Set-Cookie header, etc.
class `http.cookies.BaseCookie`([`input`])

This class is a dictionary-like object whose keys are strings and whose values are `Morsel` instances. Note that upon setting a key to a value, the value is first converted to a `Morsel` containing the key and the value.

If `input` is given, it is passed to the `load()` method.

class `http.cookies.SimpleCookie`([`input`])

This class derives from `BaseCookie` and overrides `value_decode()` and `value_encode()` to be the identity and `str()` respectively.

See Also:

Module `http.cookiejar` HTTP cookie handling for web clients. The `http.cookiejar` and `http.cookies` modules do not depend on each other.

**RFC 2109 - HTTP State Management Mechanism** This is the state management specification implemented by this module.

### 21.23.1 Cookie Objects

`BaseCookie.value_decode`(`val`)

Return a decoded value from a string representation. Return value can be any type. This method does nothing in `BaseCookie` — it exists so it can be overridden.

`BaseCookie.value_encode`(`val`)

Return an encoded value. `val` can be any type, but return value must be a string. This method does nothing in `BaseCookie` — it exists so it can be overridden.

In general, it should be the case that `value_encode()` and `value_decode()` are inverses on the range of `value_decode`.

`BaseCookie.output`(`attrs=None`, `header='Set-Cookie:'`, `sep='\r\n'`)

Return a string representation suitable to be sent as HTTP headers. `attrs` and `header` are sent to each `Morsel`'s `output()` method. `sep` is used to join the headers together, and is by default the combination `'
'` (CRLF).

`BaseCookie.js_output`(`attrs=None`)

Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP headers was sent.

The meaning for `attrs` is the same as in `output()`.

`BaseCookie.load`(`rawdata`)

If `rawdata` is a string, parse it as an `HTTP_COOKIE` and add the values found there as `Morsels`. If it is a dictionary, it is equivalent to:

```python
for k, v in rawdata.items():
    cookie[k] = v
```

### 21.23.2 Morsel Objects

class `http.cookies.Morsel`

Abstract a key/value pair, which has some **RFC 2109** attributes.

Morsels are dictionary-like objects, whose set of keys is constant — the valid

**RFC 2109** attributes, which are

* `expires`
* `path`
* `comment`
* `domain`
• max-age
• secure
• version
• httponly

The attribute httponly specifies that the cookie is only transferred in HTTP requests, and is not accessible through JavaScript. This is intended to mitigate some forms of cross-site scripting.

The keys are case-insensitive.

Morsel.value
The value of the cookie.

Morsel.coded_value
The encoded value of the cookie — this is what should be sent.

Morsel.key
The name of the cookie.

Morsel.set(key, value, coded_value)
Set the key, value and coded_value attributes.

Morsel.isReservedKey(K)
Whether K is a member of the set of keys of a Morsel.

Morsel.output(attrs=None, header='Set-Cookie: ')
Return a string representation of the Morsel, suitable to be sent as an HTTP header. By default, all the attributes are included, unless attrs is given, in which case it should be a list of attributes to use. header is by default "Set-Cookie: ".

Morsel.js_output(attrs=None)
Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP header was sent.

The meaning for attrs is the same as in output().

Morsel.OutputString(attrs=None)
Return a string representing the Morsel, without any surrounding HTTP or JavaScript.

The meaning for attrs is the same as in output().

21.23.3 Example

The following example demonstrates how to use the http.cookies module.

```python
>>> from http import cookies
>>> C = cookies.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print(C)  # generate HTTP headers
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print(C.output())  # same thing
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print(C.output(header="Cookie:"))
Cookie: rocky=road; Path=/cookie
>>> print(C.output(attrs=[], header="Cookie:"))
Cookie: rocky=road
```
>>> C = cookies.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger")  # load from a string (HTTP header)
>>> print(C)
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger

>>> C = cookies.SimpleCookie()
>>> C.load('keebler="E=everybody; L="Loves"; fudge=\012;"')
>>> print(C)
Set-Cookie: keebler="E=everybody; L="Loves"; fudge=\012;"

>>> C = cookies.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]['path'] = "/"
>>> print(C)
Set-Cookie: oreo=doublestuff; Path=/

>>> C = cookies.SimpleCookie()
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'

>>> C = cookies.SimpleCookie()
>>> C["number"] = 7  # equivalent to C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'

>>> print(C)
Set-Cookie: number=7
Set-Cookie: string=seven

21.24 http.cookiejar — Cookie handling for HTTP clients

Source code: Lib/http/cookiejar.py

The `http.cookiejar` module defines classes for automatic handling of HTTP cookies. It is useful for accessing web sites that require small pieces of data—cookies—to be set on the client machine by an HTTP response from a web server, and then returned to the server in later HTTP requests.

Both the regular Netscape cookie protocol and the protocol defined by

RFC 2965 are handled. RFC 2965 handling is switched off by default.

RFC 2109 cookies are parsed as Netscape cookies and subsequently treated either as Netscape or RFC 2965 cookies according to the `policy` in effect. Note that the great majority of cookies on the Internet are Netscape cookies. `http.cookiejar` attempts to follow the de-facto Netscape cookie protocol (which differs substantially from that set out in the original Netscape specification), including taking note of the `max-age` and `port` cookie-attributes introduced with RFC 2965.

Note: The various named parameters found in `Set-Cookie` and `Set-Cookie2` headers (e.g. `domain` and `expires`) are conventionally referred to as `attributes`. To distinguish them from Python attributes, the documentation for this module uses the term `cookie-attribute` instead.

The module defines the following exception:

`exception http.cookiejar.LoadError`

Instances of `FileCookieJar` raise this exception on failure to load cookies from a file. `LoadError` is a subclass of `OSError`. Changed in version 3.3: `LoadError` was made a subclass of `OSError` instead of `IOError`.

The following classes are provided:

```python
class http.cookiejar.CookieJar (policy=None)

policy is an object implementing the CookiePolicy interface.

The CookieJar class stores HTTP cookies. It extracts cookies from HTTP requests, and returns them in HTTP responses. CookieJar instances automatically expire contained cookies when necessary. Subclasses are also responsible for storing and retrieving cookies from a file or database.
```

```python
class http.cookiejar.FileCookieJar (filename, delayload=None, policy=None)

policy is an object implementing the CookiePolicy interface. For the other arguments, see the documentation for the corresponding attributes.

A CookieJar which can load cookies from, and perhaps save cookies to, a file on disk. Cookies are NOT loaded from the named file until either the load() or revert() method is called. Subclasses of this class are documented in section FileCookieJar subclasses and co-operation with web browsers.
```

```python
class http.cookiejar.DefaultCookiePolicy (blocked_domains=None, allowed_domains=None, netscape=True, rfc2965=False, rfc2109_as_netscape=None, hide_cookie2=False, strict_domain=False, strict_rfc2965_unverifiable=True, strict_ns_unverifiable=False, strict_ns_domain=DefaultCookiePolicy.DomainLiberal, strict_ns_set_initial_dollar=False, strict_ns_set_path=False)

Constructor arguments should be passed as keyword arguments only. blocked_domains is a sequence of domain names that we never accept cookies from, nor return cookies to. allowed_domains if not None, this is a sequence of the only domains for which we accept and return cookies. For all other arguments, see the documentation for CookiePolicy and DefaultCookiePolicy objects.

DefaultCookiePolicy implements the standard accept / reject rules for Netscape and RFC 2965 cookies. By default, RFC 2109 cookies (ie. cookies received in a Set-Cookie header with a version cookie-attribute of 1) are treated according to the RFC 2965 rules. However, if RFC 2965 handling is turned off or rfc2109_as_netscape is True, RFC 2109 cookies are ‘downgraded’ by the CookieJar instance to Netscape cookies, by setting the version attribute of the Cookie instance to 0. DefaultCookiePolicy also provides some parameters to allow some fine-tuning of policy.
```

```python
class http.cookiejar.Cookie

This class represents Netscape, RFC 2109 and RFC 2965 cookies. It is not expected that users of http.cookiejar construct their own Cookie instances. Instead, if necessary, call make_cookies() on a CookieJar instance.
```

See Also:

Module urllib.request URL opening with automatic cookie handling.

Module http.cookies HTTP cookie classes, principally useful for server-side code. The http.cookiejar and http.cookies modules do not depend on each other.

http://wp.netscape.com/newsref/std/cookie_spec.html The specification of the original Netscape cookie protocol. Though this is still the dominant protocol, the ‘Netscape cookie protocol’ implemented by all the major browsers (and http.cookiejar) only bears a passing resemblance to the one sketched out in cookie_spec.html.

RFC 2109 - HTTP State Management Mechanism Obsoleted by RFC 2965. Uses Set-Cookie with version=1.

RFC 2965 - HTTP State Management Mechanism The Netscape protocol with the bugs fixed. Uses Set-Cookie2 in place of Set-Cookie. Not widely used.

http://kristol.org/cookie/errata.html Unfinished errata to RFC 2965.
RFC 2964 - Use of HTTP State Management

21.24.1 CookieJar and FileCookieJar Objects

CookieJar objects support the iterator protocol for iterating over contained Cookie objects.

CookieJar has the following methods:

CookieJar.add_cookie_header(request)
Add correct Cookie header to request.

If policy allows (i.e. the rfc2965 and hide_cookie2 attributes of the CookieJar's CookiePolicy instance are true and false respectively), the Cookie2 header is also added when appropriate.

The request object (usually a urllib.request.Request instance) must support the methods get_full_url(), get_host(), get_type(), unverifiable(), has_header(), get_header(), header_items(), add_unredirected_header() and origin_req_host attribute as documented by urllib.request. Changed in version 3.3.

CookieJar.extract_cookies(response, request)
Extract cookies from HTTP response and store them in the CookieJar, where allowed by policy.

The CookieJar will look for allowable Set-Cookie and Set-Cookie2 headers in the response argument, and store cookies as appropriate (subject to the CookiePolicy.set_ok() method's approval).

The response object (usually the result of a call to urllib.request.urlopen(), or similar) should support an info() method, which returns a email.message.Message instance.

The request object (usually a urllib.request.Request instance) must support the methods get_full_url(), get_host(), unverifiable(), and origin_req_host attribute, as documented by urllib.request. The request is used to set default values for cookie-attributes as well as for checking that the cookie is allowed to be set. Changed in version 3.3.

CookieJar.set_policy(policy)
Set the CookiePolicy instance to be used.

CookieJar.make_cookies(response, request)
Return sequence of Cookie objects extracted from response object.

See the documentation for extract_cookies() for the interfaces required of the response and request arguments.

CookieJar.set_cookie_if_ok(cookie, request)
Set a Cookie if policy says it's OK to do so.

CookieJar.set_cookie(cookie)
Set a Cookie, without checking with policy to see whether or not it should be set.

CookieJar.clear([domain[, path[, name ]]])
Clear some cookies.

If invoked without arguments, clear all cookies. If given a single argument, only cookies belonging to that domain will be removed. If given two arguments, cookies belonging to the specified domain and URL path are removed. If given three arguments, then the cookie with the specified domain, path and name is removed.

Raises KeyError if no matching cookie exists.

CookieJar.clear_session_cookies()
Discard all session cookies.

Discards all contained cookies that have a true discard attribute (usually because they had either no max-age or expires cookie-attribute, or an explicit discard cookie-attribute). For interactive browsers, the end of a session usually corresponds to closing the browser window.

Note that the save() method won’t save session cookies anyway, unless you ask otherwise by passing a true ignore_discard argument.
FileCookieJar implements the following additional methods:

FileCookieJar.save(filename=None, ignore_discard=False, ignore_expires=False)
Save cookies to a file.

This base class raises NotImplementedError. Subclasses may leave this method unimplemented.

_filename_ is the name of file in which to save cookies. If filename is not specified, self.filename is used
(whose default is the value passed to the constructor, if any); if self.filename is None, ValueError is raised.

ignore_discard: save even cookies set to be discarded. ignore_expires: save even cookies that have expired

The file is overwritten if it already exists, thus wiping all the cookies it contains. Saved cookies can be
restored later using the load() or revert() methods.

FileCookieJar.load(filename=None, ignore_discard=False, ignore_expires=False)
Load cookies from a file.

Old cookies are kept unless overwritten by newly loaded ones.

Arguments are as for save().

The named file must be in the format understood by the class, or LoadError will be raised. Also,
OSErr or may be raised, for example if the file does not exist. Changed in version 3.3: IOError used to
be raised, it is now an alias of OSError.

FileCookieJar.revert(filename=None, ignore_discard=False, ignore_expires=False)
Clear all cookies and reload cookies from a saved file.

revert() can raise the same exceptions as load(). If there is a failure, the object’s state will not be
altered.

FileCookieJar instances have the following public attributes:

FileCookieJar.filename
Filename of default file in which to keep cookies. This attribute may be assigned to.

FileCookieJar.delayload
If true, load cookies lazily from disk. This attribute should not be assigned to. This is only a hint, since this
only affects performance, not behaviour (unless the cookies on disk are changing). A CookieJar object
may ignore it. None of the FileCookieJar classes included in the standard library lazily loads cookies.

21.24.2 FileCookieJar subclasses and co-operation with web browsers

The following CookieJar subclasses are provided for reading and writing.

class http.cookiejar.MozillaCookieJar(filename, delayload=None, policy=None)
A FileCookieJar that can load from and save cookies to disk in the Mozilla cookies.txt file format
(which is also used by the Lynx and Netscape browsers).

Note: This loses information about RFC 2965 cookies, and also about newer or non-standard cookie-
attributes such as port.

Warning: Back up your cookies before saving if you have cookies whose loss / corruption would be
inconvenient (there are some subtleties which may lead to slight changes in the file over a load / save
round-trip).

Also note that cookies saved while Mozilla is running will get clobbered by Mozilla.

class http.cookiejar.LWPCookieJar(filename, delayload=None, policy=None)
A FileCookieJar that can load from and save cookies to disk in format compatible with the libwww-
perl library’s Set-Cookie3 file format. This is convenient if you want to store cookies in a human-
readable file.
21.24.3 CookiePolicy Objects

Objects implementing the CookiePolicy interface have the following methods:

**CookiePolicy.set_ok**(cookie, request)

Return boolean value indicating whether cookie should be accepted from server.

- *cookie* is a Cookie instance. *request* is an object implementing the interface defined by the documentation for CookieJar.extract_cookies().

**CookiePolicy.return_ok**(cookie, request)

Return boolean value indicating whether cookie should be returned to server.

- *cookie* is a Cookie instance. *request* is an object implementing the interface defined by the documentation for CookieJar.add_cookie_header().

**CookiePolicy.domain_return_ok**(domain, request)

Return false if cookies should not be returned, given cookie domain.

This method is an optimization. It removes the need for checking every cookie with a particular domain (which might involve reading many files). Returning true from domain_return_ok() and path_return_ok() leaves all the work to return_ok().

If domain_return_ok() returns true for the cookie domain, path_return_ok() is called for the cookie path. Otherwise, path_return_ok() and return_ok() are never called for that cookie domain. If path_return_ok() returns true, return_ok() is called with the Cookie object itself for a full check. Otherwise, return_ok() is never called for that cookie path.

Note that domain_return_ok() is called for every cookie domain, not just for the request domain. For example, the function might be called with both ".example.com" and "www.example.com" if the request domain is "www.example.com". The same goes for path_return_ok().

The *request* argument is as documented for return_ok().

**CookiePolicy.path_return_ok**(path, request)

Return false if cookies should not be returned, given cookie path.

See the documentation for domain_return_ok().

In addition to implementing the methods above, implementations of the CookiePolicy interface must also supply the following attributes, indicating which protocols should be used, and how. All of these attributes may be assigned to.

**CookiePolicy.netscape**

Implement Netscape protocol.

**CookiePolicy.rfc2965**

Implement RFC 2965 protocol.

**CookiePolicy.hide_cookie2**

Don’t add Cookie2 header to requests (the presence of this header indicates to the server that we understand RFC 2965 cookies).

The most useful way to define a CookiePolicy class is by subclassing from DefaultCookiePolicy and overriding some or all of the methods above. CookiePolicy itself may be used as a ‘null policy’ to allow setting and receiving any and all cookies (this is unlikely to be useful).

21.24.4 DefaultCookiePolicy Objects

Implements the standard rules for accepting and returning cookies.

Both RFC 2965 and Netscape cookies are covered. RFC 2965 handling is switched off by default.

The easiest way to provide your own policy is to override this class and call its methods in your overridden implementations before adding your own additional checks:
import http.cookiejar

class MyCookiePolicy(http.cookiejar.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not http.cookiejar.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True

In addition to the features required to implement the CookiePolicy interface, this class allows you to block and allow domains from setting and receiving cookies. There are also some strictness switches that allow you to tighten up the rather loose Netscape protocol rules a little bit (at the cost of blocking some benign cookies).

A domain blacklist and whitelist is provided (both off by default). Only domains not in the blacklist and present in the whitelist (if the whitelist is active) participate in cookie setting and returning. Use the blocked_domains constructor argument, and blocked_domains() and set_blocked_domains() methods (and the corresponding argument and methods for allowed_domains). If you set a whitelist, you can turn it off again by setting it to None.

Domains in block or allow lists that do not start with a dot must equal the cookie domain to be matched. For example, "example.com" matches a blacklist entry of "example.com", but "www.example.com" does not. Domains that do start with a dot are matched by more specific domains too. For example, both "www.example.com" and "www.coyote.example.com" match ".example.com" (but "example.com" itself does not). IP addresses are an exception, and must match exactly. For example, if blocked_domains contains "192.168.1.2" and ".168.1.2", 192.168.1.2 is blocked, but 193.168.1.2 is not.

DefaultCookiePolicy implements the following additional methods:

DefaultCookiePolicy.blocked_domains()
    Return the sequence of blocked domains (as a tuple).

DefaultCookiePolicy.set_blocked_domains(blocked_domains)
    Set the sequence of blocked domains.

DefaultCookiePolicy.is_blocked(domain)
    Return whether domain is on the blacklist for setting or receiving cookies.

DefaultCookiePolicy.allowed_domains()
    Return None, or the sequence of allowed domains (as a tuple).

DefaultCookiePolicy.set_allowed_domains(allowed_domains)
    Set the sequence of allowed domains, or None.

DefaultCookiePolicy.is_not_allowed(domain)
    Return whether domain is not on the whitelist for setting or receiving cookies.

DefaultCookiePolicy instances have the following attributes, which are all initialised from the constructor arguments of the same name, and which may all be assigned to.

DefaultCookiePolicy.rfc2109_as_netscape
    If true, request that the CookieJar instance downgrade RFC 2109 cookies (ie. cookies received in a Set-Cookie header with a version cookie-attribute of 1) to Netscape cookies by setting the version attribute of the Cookie instance to 0. The default value is None, in which case RFC 2109 cookies are downgraded if and only if RFC 2965 handling is turned off. Therefore, RFC 2109 cookies are downgraded by default.

General strictness switches:

DefaultCookiePolicy.strict_domain
    Don’t allow sites to set two-component domains with country-code top-level domains like .co.uk, .gov.uk, .co.nz, etc. This is far from perfect and isn’t guaranteed to work!

RFC 2965 protocol strictness switches:
DefaultCookiePolicy\_strict\_rfc2965\_unverifiable
Follow RFC 2965 rules on unverifiable transactions (usually, an unverifiable transaction is one resulting from a redirect or a request for an image hosted on another site). If this is false, cookies are never blocked on the basis of verifiability.

Netscape protocol strictness switches:
DefaultCookiePolicy\_strict\_ns\_unverifiable
apply RFC 2965 rules on unverifiable transactions even to Netscape cookies.

DefaultCookiePolicy\_strict\_ns\_domain
Flags indicating how strict to be with domain-matching rules for Netscape cookies. See below for acceptable values.

DefaultCookiePolicy\_strict\_ns\_set\_initial\_dollar
Ignore cookies in Set-Cookie: headers that have names starting with ‘\$’.

DefaultCookiePolicy\_strict\_ns\_set\_path
Don’t allow setting cookies whose path doesn’t path-match request URI.

strict\_ns\_domain is a collection of flags. Its value is constructed by or-ing together (for example, DomainStrictNoDots|DomainStrictNonDomain means both flags are set).

DefaultCookiePolicy\_DomainStrictNoDots
When setting cookies, the ‘host prefix’ must not contain a dot (eg. www.foo.bar.com can’t set a cookie for .bar.com, because www.foo contains a dot).

DefaultCookiePolicy\_DomainStrictNonDomain
Cookies that did not explicitly specify a domain cookie-attribute can only be returned to a domain equal to the domain that set the cookie (eg. spam.example.com won’t be returned cookies from example.com that had no domain cookie-attribute).

DefaultCookiePolicy\_DomainRFC2965Match
When setting cookies, require a full RFC 2965 domain-match.

The following attributes are provided for convenience, and are the most useful combinations of the above flags:
DefaultCookiePolicy\_DomainLiberal
Equivalent to 0 (ie. all of the above Netscape domain strictness flags switched off).

DefaultCookiePolicy\_DomainStrict
Equivalent to DomainStrictNoDots|DomainStrictNonDomain.

21.24.5 Cookie Objects

Cookie instances have Python attributes roughly corresponding to the standard cookie-attributes specified in the various cookie standards. The correspondence is not one-to-one, because there are complicated rules for assigning default values, because the max-age and expires cookie-attributes contain equivalent information, and because RFC 2109 cookies may be ‘downgraded’ by http.cookiejar from version 1 to version 0 (Netscape) cookies.

Assignment to these attributes should not be necessary other than in rare circumstances in a CookiePolicy method. The class does not enforce internal consistency, so you should know what you’re doing if you do that.

Cookie.\_version
Integer or None. Netscape cookies have version 0. RFC 2965 and RFC 2109 cookies have a version cookie-attribute of 1. However, note that http.cookiejar may ‘downgrade’ RFC 2109 cookies to Netscape cookies, in which case version is 0.

Cookie.\_name
Cookie name (a string).

Cookie.\_value
Cookie value (a string), or None.
Cookie.\texttt{port}

String representing a port or a set of ports (eg. ‘80’, or ‘80,8080’), or \texttt{None}.

Cookie.\texttt{path}

Cookie path (a string, eg. ‘/acme/rocket_launchers’).

Cookie.\texttt{secure}

True if cookie should only be returned over a secure connection.

Cookie.\texttt{expires}

Integer expiry date in seconds since epoch, or \texttt{None}. See also the \texttt{is_expired()} method.

Cookie.\texttt{discard}

True if this is a session cookie.

Cookie.\texttt{comment}

String comment from the server explaining the function of this cookie, or \texttt{None}.

Cookie.\texttt{comment_url}

URL linking to a comment from the server explaining the function of this cookie, or \texttt{None}.

Cookie.\texttt{rfc2109}

True if this cookie was received as an RFC 2109 cookie (ie. the cookie arrived in a \texttt{Set-Cookie} header, and the value of the Version cookie-attribute in that header was 1). This attribute is provided because \texttt{http.cookiejar} may ‘downgrade’ RFC 2109 cookies to Netscape cookies, in which case \texttt{version} is 0.

Cookie.\texttt{port_specified}

True if a port or set of ports was explicitly specified by the server (in the \texttt{Set-Cookie} header).

Cookie.\texttt{domain_specified}

True if a domain was explicitly specified by the server.

Cookie.\texttt{domain_initial_dot}

True if the domain explicitly specified by the server began with a dot (‘.’).

Cookies may have additional non-standard cookie-attributes. These may be accessed using the following methods:

Cookie.\texttt{has_nonstandard_attr}(name)

Return true if cookie has the named cookie-attribute.

Cookie.\texttt{get_nonstandard_attr}(name, default=\texttt{None})

If cookie has the named cookie-attribute, return its value. Otherwise, return \texttt{default}.

Cookie.\texttt{set_nonstandard_attr}(name, value)

Set the value of the named cookie-attribute.

The \texttt{Cookie} class also defines the following method:

Cookie.\texttt{is_expired}(now=\texttt{None})

True if cookie has passed the time at which the server requested it should expire. If \texttt{now} is given (in seconds since the epoch), return whether the cookie has expired at the specified time.

\section*{21.24.6 Examples}

The first example shows the most common usage of \texttt{http.cookiejar}:

\begin{verbatim}
import http.cookiejar, urllib.request
cj = http.cookiejar.CookieJar()  
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))  
r = opener.open("http://example.com/")
\end{verbatim}

This example illustrates how to open a URL using your Netscape, Mozilla, or Lynx cookies (assumes Unix/Netscape convention for location of the cookies file):
```python
import os, http.cookiejar, urllib.request

cj = http.cookiejar.MozillaCookieJar()
  cj.load(os.path.join(os.path.expanduser("~"), ".netscape", "cookies.txt"))
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

The next example illustrates the use of `DefaultCookiePolicy`. Turn on RFC 2965 cookies, be more strict about domains when setting and returning Netscape cookies, and block some domains from setting cookies or having them returned:

```python
import urllib.request
from http.cookiejar import CookieJar, DefaultCookiePolicy

policy = DefaultCookiePolicy(
  rfc2965=True, strict_ns_domain=Policy.DomainStrict,
  blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

## 21.25 `xmlrpc` — XMLRPC server and client modules

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data. `xmlrpc` is a package that collects server and client modules implementing XML-RPC. The modules are:

- `xmlrpc.client`
- `xmlrpc.server`

## 21.26 `xmlrpc.client` — XML-RPC client access

**Source code:** Lib/xmlrpc/client.py

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data. This module supports writing XML-RPC client code; it handles all the details of translating between conformable Python objects and XML on the wire.

**Warning:** The `xmlrpc.client` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see [XML vulnerabilities](#).

```python
class xmlrpc.client.ServerProxy (uri, transport=None, encoding=None, verbose=False, allow_none=False, use_datetime=False, use_builtin_types=False)
```

Changed in version 3.3: The `use_builtin_types` flag was added. A `ServerProxy` instance is an object that manages communication with a remote XML-RPC server. The required first argument is a URI (Uniform Resource Indicator), and will normally be the URL of the server. The optional second argument is a transport factory instance; by default it is an internal `SafeTransport` instance for https: URLs and an internal HTTP `Transport` instance otherwise. The optional third argument is an encoding, by default UTF-8. The optional fourth argument is a debugging flag. If `allow_none` is true, the Python constant `None` will be translated into XML; the default behaviour is for `None` to raise a `TypeError`. This is a commonly-used extension to the XML-RPC specification, but isn’t supported by all clients and servers; see [http://ontosys.com/xml-rpc/extensions.php](http://ontosys.com/xml-rpc/extensions.php) for a description. The `use_builtin_types` flag can be used to cause date/time values to be presented as `datetime.datetime` objects and binary data to be presented...
as `bytes` objects; this flag is false by default. `datetime.datetime` and `bytes` objects may be passed to calls.

The obsolete `use_datetime` flag is similar to `use_builtin_types` but it applies only to date/time values.

Both the HTTP and HTTPS transports support the URL syntax extension for HTTP Basic Authentication: `http://user:pass@host:port/path`. The `user:pass` portion will be base64-encoded as an HTTP `Authorization` header, and sent to the remote server as part of the connection process when invoking an XML-RPC method. You only need to use this if the remote server requires a Basic Authentication user and password.

The returned instance is a proxy object with methods that can be used to invoke corresponding RPC calls on the remote server. If the remote server supports the introspection API, the proxy can also be used to query the remote server for the methods it supports (service discovery) and fetch other server-associated metadata.

`ServerProxy` instance methods take Python basic types and objects as arguments and return Python basic types and classes. Types that are conformable (e.g. that can be marshalled through XML), include the following (and except where noted, they are unmarshalled as the same Python type):

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>The <code>True</code> and <code>False</code> constants</td>
</tr>
<tr>
<td>integers</td>
<td>Pass in directly</td>
</tr>
<tr>
<td>floating-point numbers</td>
<td>Pass in directly</td>
</tr>
<tr>
<td>strings</td>
<td>Pass in directly</td>
</tr>
<tr>
<td>arrays</td>
<td>Any Python sequence type containing conformable elements. Arrays are returned as lists</td>
</tr>
<tr>
<td>structures</td>
<td>A Python dictionary. Keys must be strings, values may be any conformable type. Objects of user-defined classes can be passed in; only their <code>__dict__</code> attribute is transmitted.</td>
</tr>
<tr>
<td>dates</td>
<td>In seconds since the epoch. Pass in an instance of the <code>DateTime</code> class or a <code>datetime.datetime</code> instance.</td>
</tr>
<tr>
<td>binary data</td>
<td>Pass in an instance of the <code>Binary</code> wrapper class or a <code>bytes</code> instance.</td>
</tr>
</tbody>
</table>

This is the full set of data types supported by XML-RPC. Method calls may also raise a special `Fault` instance, used to signal XML-RPC server errors, or `ProtocolError` used to signal an error in the HTTP/HTTPS transport layer. Both `Fault` and `ProtocolError` derive from a base class called `Error`. Note that the `xmlrpc` client module currently does not marshal instances of subclasses of built-in types.

When passing strings, characters special to XML such as `<`, `>`, and `&` will be automatically escaped. However, it’s the caller’s responsibility to ensure that the string is free of characters that aren’t allowed in XML, such as the control characters with ASCII values between 0 and 31 (except, of course, tab, newline and carriage return); failing to do this will result in an XML-RPC request that isn’t well-formed XML. If you have to pass arbitrary bytes via XML-RPC, use the `bytes` class or the class:`Binary` wrapper class described below.

`Server` is retained as an alias for `ServerProxy` for backwards compatibility. New code should use `ServerProxy`.

See Also:

- **XML-RPC HOWTO** A good description of XML-RPC operation and client software in several languages. Contains pretty much everything an XML-RPC client developer needs to know.
- **XML-RPC Introspection** Describes the XML-RPC protocol extension for introspection.
- **XML-RPC Specification** The official specification.
- **Unofficial XML-RPC Errata** Fredrik Lundh’s “unofficial errata, intended to clarify certain details in the XML-RPC specification, as well as hint at ‘best practices’ to use when designing your own XML-RPC implementations.”
21.26.1 ServerProxy Objects

A ServerProxy instance has a method corresponding to each remote procedure call accepted by the XML-RPC server. Calling the method performs an RPC, dispatched by both name and argument signature (e.g. the same method name can be overloaded with multiple argument signatures). The RPC finishes by returning a value, which may be either returned data in a conformant type or a Fault or ProtocolError object indicating an error.

Servers that support the XML introspection API support some common methods grouped under the reserved system attribute:

ServerProxy.system.listMethods()

This method returns a list of strings, one for each (non-system) method supported by the XML-RPC server.

ServerProxy.system.methodSignature(name)

This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns an array of possible signatures for this method. A signature is an array of types. The first of these types is the return type of the method, the rest are parameters.

Because multiple signatures (ie. overloading) is permitted, this method returns a list of signatures rather than a singleton.

Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply “string, array”. If it expects three integers and returns a string, its signature is “string, int, int, int”.

If no signature is defined for the method, a non-array value is returned. In Python this means that the type of the returned value will be something other than list.

ServerProxy.system.methodHelp(name)

This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned. The documentation string may contain HTML markup.

A working example follows. The server code:

```python
from xmlrpc.server import SimpleXMLRPCServer
def is_even(n):
    return n%2 == 0

server = SimpleXMLRPCServer(('localhost', 8000))
print("Listening on port 8000...")
server.register_function(is_even, "is_even")
server.serve_forever()
```

The client code for the preceding server:

```python
import xmlrpc.client

print("3 is even: %s" % str(proxy.is_even(3)))
print("100 is even: %s" % str(proxy.is_even(100)))
```

21.26.2 DateTime Objects

This class may be initialized with seconds since the epoch, a time tuple, an ISO 8601 time/date string, or a datetime.datetime instance. It has the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

DateTime.decode(string)

Accept a string as the instance’s new time value.
DateTime.encode(out)

Write the XML-RPC encoding of this DateTime item to the out stream object.

It also supports certain of Python’s built-in operators through rich comparison and __repr__() methods.

A working example follows. The server code:

```python
import datetime
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def today():
    today = datetime.datetime.today()
    return xmlrpc.client.DateTime(today)

server = SimpleXMLRPCServer(('localhost', 8000))
print('Listening on port 8000...')
server.register_function(today, 'today')
server.serve_forever()

The client code for the preceding server:

```python
import xmlrpc.client
import datetime


today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, '%Y%m%dT%H:%M:%S')
print('Today: %s % converted.strftime("%d.%m.%Y, %H:%M")

21.26.3 Binary Objects

This class may be initialized from bytes data (which may include NULs). The primary access to the content of a Binary object is provided by an attribute:

Binary.data

The binary data encapsulated by the Binary instance. The data is provided as a bytes object.

Binary objects have the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

Binary.decode(bytes)

Accept a base64 bytes object and decode it as the instance’s new data.

Binary.encode(out)

Write the XML-RPC base 64 encoding of this binary item to the out stream object.

The encoded data will have newlines every 76 characters as per RFC 2045 section 6.8, which was the de facto standard base64 specification when the XML-RPC spec was written.

It also supports certain of Python’s built-in operators through __eq__() and __ne__() methods.

Example usage of the binary objects. We’re going to transfer an image over XMLRPC:

```python
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpc.client.Binary(handle.read())

server = SimpleXMLRPCServer(('localhost', 8000))
```
print("Listening on port 8000...")
server.register_function(python_logo, 'python_logo')
server.serve_forever()

The client gets the image and saves it to a file:

```python
import xmlrpc.client

with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
```

### 21.26.4 Fault Objects

A Fault object encapsulates the content of an XML-RPC fault tag. Fault objects have the following attributes:

- **Fault.faultCode**: A string indicating the fault type.
- **Fault.faultString**: A string containing a diagnostic message associated with the fault.

In the following example we’re going to intentionally cause a Fault by returning a complex type object. The server code:

```python
from xmlrpc.server import SimpleXMLRPCServer

# A marshalling error is going to occur because we're returning a complex number
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(('localhost', 8000))
print("Listening on port 8000...")
server.register_function(add, 'add')
server.serve_forever()
```

The client code for the preceding server:

```python
import xmlrpc.client

try:
    proxy.add(2, 5)
except xmlrpc.client.Fault as err:
    print("A fault occurred")
    print("Fault code: %d" % err.faultCode)
    print("Fault string: %s" % err.faultString)
```

### 21.26.5 ProtocolError Objects

A ProtocolError object describes a protocol error in the underlying transport layer (such as a 404 ‘not found’ error if the server named by the URI does not exist). It has the following attributes:

- **ProtocolError.url**: The URI or URL that triggered the error.
- **ProtocolError.errcode**: The error code.
ProtocolError.errmsg
  The error message or diagnostic string.

ProtocolError.headers
  A dict containing the headers of the HTTP/HTTPS request that triggered the error.

In the following example we’re going to intentionally cause a ProtocolError by providing an invalid URI:

    import xmlrpc.client

    # create a ServerProxy with an URI that doesn’t respond to XMLRPC requests
    proxy = xmlrpc.client.ServerProxy("http://google.com/")

    try:
        proxy.some_method()
    except xmlrpc.client.ProtocolError as err:
        print("A protocol error occurred")
        print("URL: %s" % err.url)
        print("HTTP/HTTPS headers: %s" % err.headers)
        print("Error code: %d" % err.errcode)
        print("Error message: %s" % err.errmsg)

21.26.6 MultiCall Objects

The MultiCall object provides a way to encapsulate multiple calls to a remote server into a single request.³

    class xmlrpc.client.MultiCall(server)

    Create an object used to boxcar method calls. server is the eventual target of the call. Calls can be made to the result object, but they will immediately return None, and only store the call name and parameters in the MultiCall object. Calling the object itself causes all stored calls to be transmitted as a single system.multicall request. The result of this call is a generator; iterating over this generator yields the individual results.

A usage example of this class follows. The server code:

    from xmlrpc.server import SimpleXMLRPCServer

    def add(x, y):
        return x + y

    def subtract(x, y):
        return x - y

    def multiply(x, y):
        return x * y

    def divide(x, y):
        return x // y

    # A simple server with simple arithmetic functions
    server = SimpleXMLRPCServer(("localhost", 8000))
    print("Listening on port 8000...")
    server.register_multicall_functions()
    server.register_function(add, 'add')
    server.register_function(subtract, 'subtract')
    server.register_function(multiply, 'multiply')
    server.register_function(divide, 'divide')
    server.serve_forever()

³ This approach has been first presented in a discussion on xmlrpc.com.
The client code for the preceding server:

```python
import xmlrpc.client

multicall = xmlrpc.client.MultiCall(proxy)
multicall.add(7, 3)
multicall.subtract(7, 3)
multicall.multiply(7, 3)
multicall.divide(7, 3)
result = multicall()

print("7+3=%d, 7-3=%d, 7*3=%d, 7//3=%d" % tuple(result))
```

21.26.7 Convenience Functions

`xmlrpc.client.dumps`(params, methodname=None, methodresponse=None, encoding=None, allow_none=False)

Convert `params` into an XML-RPC request. or into a response if `methodresponse` is true. `params` can be either a tuple of arguments or an instance of the `Fault` exception class. If `methodresponse` is true, only a single value can be returned, meaning that `params` must be of length 1. `encoding`, if supplied, is the encoding to use in the generated XML; the default is UTF-8. Python's `None` value cannot be used in standard XML-RPC; to allow using it via an extension, provide a true value for `allow_none`.

`xmlrpc.client.loads`(data, use_datetime=False, use_builtin_types=False)

Convert an XML-RPC request or response into Python objects, a `(params, methodname)`. `params` is a tuple of argument; `methodname` is a string, or `None` if no method name is present in the packet. If the XML-RPC packet represents a fault condition, this function will raise a `Fault` exception. The `use_builtin_types` flag can be used to cause date/time values to be presented as `datetime.datetime` objects and binary data to be presented as `bytes` objects; this flag is false by default.

The obsolete `use_datetime` flag is similar to `use_builtin_types` but it applies only to date/time values.

Changed in version 3.3: The `use_builtin_types` flag was added.

21.26.8 Example of Client Usage

```python
# simple test program (from the XML-RPC specification)
from xmlrpc.client import ServerProxy, Error

# server = ServerProxy("http://localhost:8000")  # local server
server = ServerProxy("http://betty.userland.com")

print(server)

try:
    print(server.examples.getStateName(41))
except Error as v:
    print("ERROR", v)
```

To access an XML-RPC server through a proxy, you need to define a custom transport. The following example shows how:

```python
import xmlrpc.client, http.client

class ProxiedTransport(xmlrpc.client.Transport):
    def set_proxy(self, proxy):
        self.proxy = proxy
    def make_connection(self, host):
        self.realhost = host
```
h = http.client.HTTP(self.proxy)
return h

def send_request(self, connection, handler, request_body):
    connection.putrequest("POST", 'http://%s%s' % (self.realhost, handler))

def send_host(self, connection, host):
    connection.putheader('Host', self.realhost)

p = ProxiedTransport()
p.set_proxy('proxy-server:8080')
print(server.currentTime.getCurrentTime())

21.26.9 Example of Client and Server Usage

See SimpleXMLRPCServer Example.

21.27 xmlrpc.server — Basic XML-RPC servers

Source code: Lib/xmlrpc/server.py

The xmlrpc.server module provides a basic server framework for XML-RPC servers written in Python. Servers can either be free standing, using SimpleXMLRPCServer, or embedded in a CGI environment, using CGIXMLRPCRequestHandler.

Warning: The xmlrpc.client module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see XML vulnerabilities.

class xmlrpc.server.SimpleXMLRPCServer (addr, requestHandler=SimpleXMLRPCRequestHandler, logRequests=True, allow_none=False, encoding=None, bind_and_activate=True, use_builtin_types=False)
Create a new server instance. This class provides methods for registration of functions that can be called by the XML-RPC protocol. The requestHandler parameter should be a factory for request handler instances; it defaults to SimpleXMLRPCRequestHandler. The addr and requestHandler parameters are passed to the socketserver.TCPServer constructor. If logRequests is true (the default), requests will be logged; setting this parameter to false will turn off logging. The allow_none and encoding parameters are passed on to xmlrpc.client and control the XML-RPC responses that will be returned from the server. The bind_and_activate parameter controls whether server_bind() and server_activate() are called immediately by the constructor; it defaults to true. Setting it to false allows code to manipulate the allow_reuse_address class variable before the address is bound. The use_builtin_types parameter is passed to the loads() function and controls which types are processed when date/times values or binary data are received; it defaults to false. Changed in version 3.3: The use_builtin_types flag was added.

class xmlrpc.server.CGIXMLRPCRequestHandler (allow_none=False, encoding=None, use_builtin_types=False)
Create a new instance to handle XML-RPC requests in a CGI environment. The allow_none and encoding parameters are passed on to xmlrpc.client and control the XML-RPC responses that will be returned from the server. The use_builtin_types parameter is passed to the loads() function and controls which types are processed when date/times values or binary data are received; it defaults to false. Changed in version 3.3: The use_builtin_types flag was added.

class xmlrpc.server.SimpleXMLRPCRequestHandler
Create a new request handler instance. This request handler supports POST requests and modifies logging so that the logRequests parameter to the SimpleXMLRPCServer constructor parameter is honored.
21.27.1 SimpleXMLRPCServer Objects

The SimpleXMLRPCServer class is based on socketserver.TCPServer and provides a means of creating simple, stand alone XML-RPC servers.

**SimpleXMLRPCServer.register_function (function, name=None)**

Register a function that can respond to XML-RPC requests. If name is given, it will be the method name associated with function, otherwise function.__name__ will be used. name can be either a normal or Unicode string, and may contain characters not legal in Python identifiers, including the period character.

**SimpleXMLRPCServer.register_instance (instance, allow_dotted_names=False)**

Register an object which is used to expose method names which have not been registered using register_function(). If instance contains a _dispatch() method, it is called with the requested method name and the parameters from the request. Its API is def _dispatch(self, method, params) (note that params does not represent a variable argument list). If it calls an underlying function to perform its task, that function is called as func(*params), expanding the parameter list. The return value from _dispatch() is returned to the client as the result. If instance does not have a _dispatch() method, it is searched for an attribute matching the name of the requested method.

If the optional allow_dotted_names argument is true and the instance does not have a _dispatch() method, then if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

**Warning:** Enabling the allow_dotted_names option allows intruders to access your module’s global variables and may allow intruders to execute arbitrary code on your machine. Only use this option on a secure, closed network.

**SimpleXMLRPCServer.register_introspection_functions ()**

Registers the XML-RPC introspection functions system.listMethods, system.methodHelp and system.methodSignature.

**SimpleXMLRPCServer.register_multicall_functions ()**

Registers the XML-RPC multicall function system.multicall.

**SimpleXMLRPCRequestHandler.rpc_paths**

An attribute value that must be a tuple listing valid path portions of the URL for receiving XML-RPC requests. Requests posted to other paths will result in a 404 “no such page” HTTP error. If this tuple is empty, all paths will be considered valid. The default value is (’/’, ’/RPC2’).

**SimpleXMLRPCServer Example**

Server code:

```python
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = (’/RPC2’,)

# Create server
server = SimpleXMLRPCServer(("localhost", 8000),
    requestHandler=RequestHandler)
server.register_introspection_functions()

# Register pow() function; this will use the value of
# pow.__name__ as the name, which is just 'pow'.
server.register_function(pow)
```

21.27. xmlrpc.server — Basic XML-RPC servers 951
# Register a function under a different name
def adder_function(x, y):
    return x + y
server.register_function(adder_function, 'add')

# Register an instance; all the methods of the instance are
# published as XML-RPC methods (in this case, just 'mul').
class MyFuncs:
    def mul(self, x, y):
        return x * y

server.register_instance(MyFuncs())

# Run the server's main loop
server.serve_forever()

The following client code will call the methods made available by the preceding server:

```python
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')
print(s.pow(2, 3))  # Returns 2**3 = 8
print(s.add(2, 3))  # Returns 5
print(s.mul(5, 2))  # Returns 5*2 = 10

# Print list of available methods
print(s.system.listMethods())
```

## 21.27.2 CGIXMLRPCRequestHandler

The CGIXMLRPCRequestHandler class can be used to handle XML-RPC requests sent to Python CGI scripts.

**CGI XMLRPCRequestHandler.register_function** *(function, name=None)*

Register a function that can respond to XML-RPC requests. If name is given, it will be the method name associated with function, otherwise function.__name__ will be used. name can be either a normal or Unicode string, and may contain characters not legal in Python identifiers, including the period character.

**CGI XMLRPCRequestHandler.register_instance** *(instance)*

Register an object which is used to expose method names which have not been registered using register_function(). If instance contains a _dispatch() method, it is called with the requested method name and the parameters from the request; the return value is returned to the client as the result. If instance does not have a _dispatch() method, it is searched for an attribute matching the name of the requested method; if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

**CGI XMLRPCRequestHandler.register_introspection_functions** ()

Register the XML-RPC introspection functions system.listMethods, system.methodHelp and system.methodSignature.

**CGI XMLRPCRequestHandler.register_multicall_functions** ()

Register the XML-RPC multicall function system.multicall.

**CGI XMLRPCRequestHandler.handle_request** *(request_text=None)*

Handle a XML-RPC request. If request_text is given, it should be the POST data provided by the HTTP server, otherwise the contents of stdin will be used.

Example:
class MyFuncs:
    def mul(self, x, y):
        return x * y

handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x, y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()

21.27.3 Documenting XMLRPC server

These classes extend the above classes to serve HTML documentation in response to HTTP GET requests. Servers can either be free standing, using DocXMLRPCServer, or embedded in a CGI environment, using DocCGIXMLRPCRequestHandler.

class xmlrpc.server.DocXMLRPCServer(addr, requestHandler=DocXMLRPCRequestHandler, logRequests=True, allow_none=False, encoding=None, bind_and_activate=True, use_builtin_types=True)

Create a new server instance. All parameters have the same meaning as for SimpleXMLRPCServer; requestHandler defaults to DocXMLRPCRequestHandler. Changed in version 3.3: The use_builtin_types flag was added.

class xmlrpc.server.DocCGIXMLRPCRequestHandler

Create a new instance to handle XML-RPC requests in a CGI environment.

class xmlrpc.server.DocXMLRPCRequestHandler

Create a new request handler instance. This request handler supports XML-RPC POST requests, documentation GET requests, and modifies logging so that the logRequests parameter to the DocXMLRPCServer constructor parameter is honored.

21.27.4 DocXMLRPCServer Objects

The DocXMLRPCServer class is derived from SimpleXMLRPCServer and provides a means of creating self-documenting, stand alone XML-RPC servers. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocXMLRPCServer.set_server_title(server_title)
    Set the title used in the generated HTML documentation. This title will be used inside the HTML “title” element.

DocXMLRPCServer.set_server_name(server_name)
    Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a “h1” element.

DocXMLRPCServer.set_server_documentation(server_documentation)
    Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.

21.27.5 DocCGIXMLRPCRequestHandler

The DocCGIXMLRPCRequestHandler class is derived from CGIXMLRPCRequestHandler and provides a means of creating self-documenting, XML-RPC CGI scripts. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.
DocCGIXMLRPCRequestHandler.set_server_title(server_title)

Set the title used in the generated HTML documentation. This title will be used inside the HTML “title”
element.

DocCGIXMLRPCRequestHandler.set_server_name(server_name)

Set the name used in the generated HTML documentation. This name will appear at the top of the generated
documentation inside a “h1” element.

DocCGIXMLRPCRequestHandler.set_server_documentation(server_documentation)

Set the description used in the generated HTML documentation. This description will appear as a paragraph,
below the server name, in the documentation.

21.28 ipaddress — IPv4/IPv6 manipulation library

Source code: Lib/ipaddress.py

Note: The ipaddress module has been included in the standard library on a provisional basis. Backwards
incompatible changes (up to and including removal of the package) may occur if deemed necessary by the core
developers.

ipaddress provides the capabilities to create, manipulate and operate on IPv4 and IPv6 addresses and networks.
The functions and classes in this module make it straightforward to handle various tasks related to IP addresses,
including checking whether or not two hosts are on the same subnet, iterating over all hosts in a particular subnet,
checking whether or not a string represents a valid IP address or network definition, and so on.

This is the full module API reference - for an overview and introduction, see ipaddress-howto. New in version 3.3.

21.28.1 Convenience factory functions

The ipaddress module provides factory functions to conveniently create IP addresses, networks and interfaces:

ipaddress.ip_address(address)

Return an IPv4Address or IPv6Address object depending on the IP address passed as argument.
Either IPv4 or IPv6 addresses may be supplied; integers less than 2**32 will be considered to be IPv4 by
default. A ValueError is raised if address does not represent a valid IPv4 or IPv6 address.

>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')

ipaddress.ip_network(address, strict=True)

Return an IPv4Network or IPv6Network object depending on the IP address passed as argument.
address is a string or integer representing the IP network. Either IPv4 or IPv6 networks may be supplied;
integers less than 2**32 will be considered to be IPv4 by default. strict is passed to IPv4Network or
IPv6Network constructor. A ValueError is raised if address does not represent a valid IPv4 or IPv6
address, or if the network has host bits set.

>>> ipaddress.ip_network('192.168.0.0/28')
IPv4Network('192.168.0.0/28')

ipaddress.ip_interface(address)

Return an IPv4Interface or IPv6Interface object depending on the IP address passed as argument.
address is a string or integer representing the IP address. Either IPv4 or IPv6 addresses may be supplied;
integers less than 2**32 will be considered to be IPv4 by default. A ValueError is raised if address does
not represent a valid IPv4 or IPv6 address.
One downside of these convenience functions is that the need to handle both IPv4 and IPv6 formats means that error messages provide minimal information on the precise error, as the functions don’t know whether the IPv4 or IPv6 format was intended. More detailed error reporting can be obtained by calling the appropriate version specific class constructors directly.

21.28.2 IP Addresses

Address objects

The IPv4Address and IPv6Address objects share a lot of common attributes. Some attributes that are only meaningful for IPv6 addresses are also implemented by IPv4Address objects, in order to make it easier to write code that handles both IP versions correctly.

class ipaddress.IPv4Address(address)

Construct an IPv4 address. An AddressValueError is raised if address is not a valid IPv4 address.

The following constitutes a valid IPv4 address:

1. A string in decimal-dot notation, consisting of four decimal integers in the inclusive range 0-255, separated by dots (e.g. 192.168.0.1). Each integer represents an octet (byte) in the address. Leading zeroes are tolerated only for values less then 8 (as there is no ambiguity between the decimal and octal interpretations of such strings).

2. An integer that fits into 32 bits.

3. An integer packed into a bytes object of length 4 (most significant octet first).

>>> ipaddress.IPv4Address('192.168.0.1')
IPv4Address('192.168.0.1')

>>> ipaddress.IPv4Address(3232235521)
IPv4Address('192.168.0.1')

>>> ipaddress.IPv4Address(b'\x00\xA8\x00\x01')
IPv4Address('192.168.0.1')

version

The appropriate version number: 4 for IPv4, 6 for IPv6.

max_prefixlen

The total number of bits in the address representation for this version: 32 for IPv4, 128 for IPv6.

The prefix defines the number of leading bits in an address that are compared to determine whether or not an address is part of a network.

compressed

The string representation in dotted decimal notation. Leading zeroes are never included in the representation.

As IPv4 does not define a shorthand notation for addresses with octets set to zero, these two attributes are always the same as str(addr) for IPv4 addresses. Exposing these attributes makes it easier to write display code that can handle both IPv4 and IPv6 addresses.

packed

The binary representation of this address - a bytes object of the appropriate length (most significant octet first). This is 4 bytes for IPv4 and 16 bytes for IPv6.

is_multicast

True if the address is reserved for multicast use. See RFC 3171 (for IPv4) or RFC 2373 (for IPv6).
is_private
True if the address is allocated for private networks. See
RFC 1918 (for IPv4) or RFC 4193 (for IPv6).

is_unspecified
True if the address is unspecified. See RFC 5375 (for IPv4) or RFC 2373 (for IPv6).

is_reserved
True if the address is otherwise IETF reserved.

is_loopback
True if this is a loopback address. See RFC 3330 (for IPv4) or RFC 2373 (for IPv6).

is_link_local
True if the address is reserved for link-local usage. See
RFC 3927.

class ipaddress.IPv6Address (address)
Construct an IPv6 address. An AddressValueError is raised if address is not a valid IPv6 address.

The following constitutes a valid IPv6 address:

1. A string consisting of eight groups of four hexadecimal digits, each group representing 16 bits. The
groups are separated by colons. This describes an exploded (longhand) notation. The string can also
be compressed (shorthand notation) by various means. See
RFC 4291 for details. For example, "0000:0000:0000:0000:0000:0abc:0007:0def" can
be compressed to "::abc:7:def".

2. An integer that fits into 128 bits.

3. An integer packed into a bytes object of length 16, big-endian.

>>> ipaddress.IPv6Address('2001:db8::1000')
IPv6Address('2001:db8::1000')

compressed
The short form of the address representation, with leading zeroes in groups omitted and the longest sequence
of groups consisting entirely of zeroes collapsed to a single empty group.
This is also the value returned by str (addr) for IPv6 addresses.

exploded
The long form of the address representation, with all leading zeroes and groups consisting entirely of zeroes
included.

packed
version
max_prefixlen
is_multicast
is_private
is_unspecified
is_reserved
is_loopback
is_link_local
Refer to the corresponding attribute documentation in IPv4Address
is_site_local
True if the address is reserved for site-local usage. Note that the site-local address space has been deprecated by RFC 3879. Use is_private to test if this address is in the space of unique local addresses as defined by RFC 4193.

ipv4_mapped
For addresses that appear to be IPv4 mapped addresses (starting with ::FFFF/96), this property will report the embedded IPv4 address. For any other address, this property will be None.

sixtofour
For addresses that appear to be 6to4 addresses (starting with 2002::/16) as defined by RFC 3056, this property will report the embedded IPv4 address. For any other address, this property will be None.

teredo
For addresses that appear to be Teredo addresses (starting with 2001::/32) as defined by RFC 4380, this property will report the embedded (server, client) IP address pair. For any other address, this property will be None.

Conversion to Strings and Integers

To interoperate with networking interfaces such as the socket module, addresses must be converted to strings or integers. This is handled using the str() and int() builtin functions:

```python
>>> str(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> int(ipaddress.IPv4Address('192.168.0.1'))
3232235521
>>> str(ipaddress.IPv6Address('::1'))
'::1'
>>> int(ipaddress.IPv6Address('::1'))
1
```

Operators

Address objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Comparison operators

Address objects can be compared with the usual set of comparison operators. Some examples:

```python
>>> IPv4Address('127.0.0.2') > IPv4Address('127.0.0.1')
True
>>> IPv4Address('127.0.0.2') == IPv4Address('127.0.0.1')
False
>>> IPv4Address('127.0.0.2') != IPv4Address('127.0.0.1')
True
```

Arithmetic operators

Integers can be added to or subtracted from address objects. Some examples:

```python
>>> IPv4Address('127.0.0.2') + 3
IPv4Address('127.0.0.5')
>>> IPv4Address('127.0.0.2') - 3
IPv4Address('126.255.255.255')
>>> IPv4Address('255.255.255.255') + 1
```
21.28.3 IP Network definitions

The IPv4Network and IPv6Network objects provide a mechanism for defining and inspecting IP network definitions. A network definition consists of a mask and a network address, and as such defines a range of IP addresses that equal the network address when masked (binary AND) with the mask. For example, a network definition with the mask 255.255.255.0 and the network address 192.168.1.0 consists of IP addresses in the inclusive range 192.168.1.0 to 192.168.1.255.

Prefix, net mask and host mask

There are several equivalent ways to specify IP network masks. A prefix /<nbits> is a notation that denotes how many high-order bits are set in the network mask. A net mask is an IP address with some number of high-order bits set. Thus the prefix /24 is equivalent to the net mask 255.255.255.0 in IPv4, or ffff:ff00:: in IPv6. In addition, a host mask is the logical inverse of a net mask, and is sometimes used (for example in Cisco access control lists) to denote a network mask. The host mask equivalent to /24 in IPv4 is 0.0.0.255.

Network objects

All attributes implemented by address objects are implemented by network objects as well. In addition, network objects implement additional attributes. All of these are common between IPv4Network and IPv6Network, so to avoid duplication they are only documented for IPv4Network.

class ipaddress.IPv4Network(address, strict=True)

Construct an IPv4 network definition. address can be one of the following:

1. A string consisting of an IP address and an optional mask, separated by a slash (/). The IP address is the network address, and the mask can be either a single number, which means it’s a prefix, or a string representation of an IPv4 address. If it’s the latter, the mask is interpreted as a net mask if it starts with a non-zero field, or as a host mask if it starts with a zero field. If no mask is provided, it’s considered to be /32.

   For example, the following address specifications are equivalent: 192.168.1.0/24, 192.168.1.0/255.255.255.0 and 192.168.1.0/0.0.0.255.

2. An integer that fits into 32 bits. This is equivalent to a single-address network, with the network address being address and the mask being /32.

3. An integer packed into a bytes object of length 4, big-endian. The interpretation is similar to an integer address.

An AddressValueError is raised if address is not a valid IPv4 address. A NetmaskValueError is raised if the mask is not valid for an IPv4 address.

If strict is True and host bits are set in the supplied address, then ValueError is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

Unless stated otherwise, all network methods accepting other network/address objects will raise TypeError if the argument’s IP version is incompatible to self version.

max_prefixlen

Refer to the corresponding attribute documentation in IPv4Address

is_multicast

is_private
is_unspecified

is_reserved

is_loopback

is_link_local

These attributes are true for the network as a whole if they are true for both the network address and the broadcast address

network_address

The network address for the network. The network address and the prefix length together uniquely define a network.

broadcast_address

The broadcast address for the network. Packets sent to the broadcast address should be received by every host on the network.

host_mask

The host mask, as a string.

with_prefixlen

compressed

exploded

A string representation of the network, with the mask in prefix notation.

with_prefixlen and compressed are always the same as str(network).exploded uses the exploded form the network address.

with_netmask

A string representation of the network, with the mask in net mask notation.

with_hostmask

A string representation of the network, with the mask in host mask notation.

num_addresses

The total number of addresses in the network.

prefixlen

Length of the network prefix, in bits.

hosts()

Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the network address itself and the network broadcast address.

```python
>>> list(ip_network('192.0.2.0/29').hosts())
[IPv4Address('192.0.2.1'), IPv4Address('192.0.2.2'),
 IPv4Address('192.0.2.3'), IPv4Address('192.0.2.4'),
 IPv4Address('192.0.2.5'), IPv4Address('192.0.2.6')]
```

overlaps(other)

True if this network is partly or wholly contained in other or other is wholly contained in this network.

address_exclude(network)

Computes the network definitions resulting from removing the given network from this one. Returns an iterator of network objects. Raises ValueError if network is not completely contained in this network.

```python
>>> n1 = ip_network('192.0.2.0/28')
>>> n2 = ip_network('192.0.2.1/32')
>>> list(n1.address_exclude(n2))
[IPv4Network('192.0.2.8/29'), IPv4Network('192.0.2.4/30'),
 IPv4Network('192.0.2.2/31'), IPv4Network('192.0.2.0/32')]
```
subnets (prefixlen_diff=1, new_prefix=None)
The subnets that join to make the current network definition, depending on the argument values. prefixlen_diff is the amount our prefix length should be increased by. new_prefix is the desired new prefix of the subnets; it must be larger than our prefix. One and only one of prefixlen_diff and new_prefix must be set. Returns an iterator of network objects.

>>> list(ip_network('192.0.2.0/24').subnets())
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]

>>> list(ip_network('192.0.2.0/24').subnets(prefixlen_diff=2))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'), IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]

>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=26))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'), IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]

>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=23))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise
ValueError: new prefix must be longer

>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=25))
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]

supernet (prefixlen_diff=1, new_prefix=None)
The supernet containing this network definition, depending on the argument values. prefixlen_diff is the amount our prefix length should be decreased by. new_prefix is the desired new prefix of the supernet; it must be smaller than our prefix. One and only one of prefixlen_diff and new_prefix must be set. Returns a single network object.

>>> ip_network('192.0.2.0/24').supernet()
IPv4Network('192.0.0.0/23')

>>> ip_network('192.0.2.0/24').supernet(prefixlen_diff=2)
IPv4Network('192.0.0.0/22')

>>> ip_network('192.0.2.0/24').supernet(new_prefix=20)
IPv4Network('192.0.0.0/20')

cmpare_networks (other)
Compare this network to other. In this comparison only the network addresses are considered; host bits aren’t. Returns either -1, 0 or 1.

>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.2/32'))
-1

>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.0/32'))
1

>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.1/32'))
0

class ipaddress.IPv6Network (address, strict=True)
Construct an IPv6 network definition. address can be one of the following:

1. A string consisting of an IP address and an optional mask, separated by a slash (/). The IP address is the network address, and the mask can be either a single number, which means it’s a prefix, or a string representation of an IPv6 address. If it’s the latter, the mask is interpreted as a net mask. If no mask is provided, it’s considered to be /128.

For example, the following address specifications are equivalent: 2001:db00::0/24 and 2001:db00::/ffff:ff00::

2. An integer that fits into 128 bits. This is equivalent to a single-address network, with the network address being address and the mask being /128.
An integer packed into a bytes object of length 16, bit-endian. The interpretation is similar to an integer address.

An AddressValueError is raised if address is not a valid IPv6 address. A NetmaskValueError is raised if the mask is not valid for an IPv6 address.

If strict is True and host bits are set in the supplied address, then ValueError is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

version
max_prefixlen
is_multicast
is_private
is_unspecified
is_reserved
is_loopback
is_link_local
network_address
broadcast_address
host mask
with_prefixlen
compressed
exploded
with_netmask
with_hostmask
num_addresses
prefixlen
hosts()
overlaps(other)
address_exclude(network)
subnets(prefixlen_diff=1, new_prefix=None)
supernet(prefixlen_diff=1, new_prefix=None)
compare_networks(other)
Refer to the corresponding attribute documentation in IPv4Network

is_site_local
These attribute is true for the network as a whole if it is true for both the network address and the broadcast address

Operators

Network objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Logical operators

Network objects can be compared with the usual set of logical operators, similarly to address objects.
Iteration

Network objects can be iterated to list all the addresses belonging to the network. For iteration, all hosts are returned, including unusable hosts (for usable hosts, use the hosts() method). An example:

```python
>>> for addr in IPv4Network('192.0.2.0/28):
...    addr
... IPv4Address('192.0.2.0')
IPv4Address('192.0.2.1')
IPv4Address('192.0.2.2')
IPv4Address('192.0.2.3')
IPv4Address('192.0.2.4')
IPv4Address('192.0.2.5')
IPv4Address('192.0.2.6')
IPv4Address('192.0.2.7')
IPv4Address('192.0.2.8')
IPv4Address('192.0.2.9')
IPv4Address('192.0.2.10')
IPv4Address('192.0.2.11')
IPv4Address('192.0.2.12')
IPv4Address('192.0.2.13')
IPv4Address('192.0.2.14')
IPv4Address('192.0.2.15')
```

Networks as containers of addresses

Network objects can act as containers of addresses. Some examples:

```python
>>> IPv4Network('192.0.2.0/28')[0]
IPv4Address('192.0.2.0')
>>> IPv4Network('192.0.2.0/28')[15]
IPv4Address('192.0.2.15')
>>> IPv4Address('192.0.2.6') in IPv4Network('192.0.2.0/28')
True
>>> IPv4Address('192.0.3.6') in IPv4Network('192.0.2.0/28')
False
```

21.28.4 Interface objects

class `ipaddress.IPv4Interface`

```
address)
```

Construct an IPv4 interface. The meaning of address is as in the constructor of IPv4Network, except that arbitrary host addresses are always accepted.

IPv4Interface is a subclass of IPv4Address, so it inherits all the attributes from that class. In addition, the following attributes are available:

```python
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.ip
IPv4Address('192.0.2.5')
```

```
network)
```

The network (IPv4Network) this interface belongs to.
```python
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.network
IPv4Network('192.0.2.0/24')

with_prefixlen
A string representation of the interface with the mask in prefix notation.

```python
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_prefixlen
'192.0.2.5/24'
```

with_netmask
A string representation of the interface with the network as a net mask.

```python
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_netmask
'192.0.2.5/255.255.255.0'
```

with_hostmask
A string representation of the interface with the network as a host mask.

```python
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_hostmask
'192.0.2.5/0.0.0.255'
```

class ipaddress.IPv6Interface(address)
Construct an IPv6 interface. The meaning of address is as in the constructor of IPv6Network, except that arbitrary host addresses are always accepted.

IPv6Interface is a subclass of IPv6Address, so it inherits all the attributes from that class. In addition, the following attributes are available:

- ip
- network
- with_prefixlen
- with_netmask
- with_hostmask

Refer to the corresponding attribute documentation in IPv4Interface.

21.28.5 Other Module Level Functions

The module also provides the following module level functions:

ipaddress.v4_int_to_packed(address)
Represent an address as 4 packed bytes in network (big-endian) order. address is an integer representation of an IPv4 IP address. A ValueError is raised if the integer is negative or too large to be an IPv4 IP address.

```python
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>> ipaddress.v4_int_to_packed(3221225985)
b'\xc0\x00\x02\x01'
```

ipaddress.v6_int_to_packed(address)
Represent an address as 16 packed bytes in network (big-endian) order. address is an integer representation of an IPv6 IP address. A ValueError is raised if the integer is negative or too large to be an IPv6 IP address.
The Python Library Reference, Release 3.3.3

**ipaddress.summarize_address_range(first, last)**

Return an iterator of the summarized network range given the first and last IP addresses. `first` is the first `IPv4Address` or `IPv6Address` in the range and `last` is the last `IPv4Address` or `IPv6Address` in the range. A `TypeError` is raised if `first` or `last` are not IP addresses or are not of the same version. A `ValueError` is raised if `last` is not greater than `first` or if `first` address version is not 4 or 6.

```python
>>> [ipaddr for ipaddr in ipaddress.summarize_address_range(  
...     ipaddress.IPv4Address('192.0.2.0'),  
...     ipaddress.IPv4Address('192.0.2.130'))]
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/31'), IPv4Network('192.0.2.130/32')]
```

**ipaddress.collapse_addresses(addresses)**

Return an iterator of the collapsed `IPv4Network` or `IPv6Network` objects. `addresses` is an iterator of `IPv4Network` or `IPv6Network` objects. A `TypeError` is raised if `addresses` contains mixed version objects.

```python
>>> [ipaddr for ipaddr in  
...     ipaddress.collapse_addresses([ipaddress.IPv4Network('192.0.2.0/25'),  
...                                   ipaddress.IPv4Network('192.0.2.128/25')])]
[IPv4Network('192.0.2.0/24')]
```

**ipaddress.get_mixed_type_key(obj)**

Return a key suitable for sorting between networks and addresses. Address and Network objects are not sortable by default; they’re fundamentally different, so the expression:

```
IPv4Address('192.0.2.0') <= IPv4Network('192.0.2.0/24')
```

doesn’t make sense. There are some times however, where you may wish to have `ipaddress` sort these anyway. If you need to do this, you can use this function as the `key` argument to `sorted()`.

`obj` is either a network or address object.

### 21.28.6 Custom Exceptions

To support more specific error reporting from class constructors, the module defines the following exceptions:

- **exception ipaddress.AddressValueError(ValueError)**
  Any value error related to the address.

- **exception ipaddress.NetmaskValueError(ValueError)**
  Any value error related to the netmask.
The modules described in this chapter implement various algorithms or interfaces that are mainly useful for multimedia applications. They are available at the discretion of the installation. Here’s an overview:

### 22.1 audioop — Manipulate raw audio data

The audioop module contains some useful operations on sound fragments. It operates on sound fragments consisting of signed integer samples 8, 16 or 32 bits wide, stored in bytes objects. All scalar items are integers, unless specified otherwise.

This module provides support for a-LAW, u-LAW and Intel/DVI ADPCM encodings.

A few of the more complicated operations only take 16-bit samples, otherwise the sample size (in bytes) is always a parameter of the operation.

The module defines the following variables and functions:

**exception audioop.error**

This exception is raised on all errors, such as unknown number of bytes per sample, etc.

**audioop.add(fragment1, fragment2, width)**

Return a fragment which is the addition of the two samples passed as parameters. width is the sample width in bytes, either 1, 2 or 4. Both fragments should have the same length. Samples are truncated in case of overflow.

**audioop.adpcm2lin(adpcmfragment, width, state)**

Decode an Intel/DVI ADPCM coded fragment to a linear fragment. See the description of lin2adpcm() for details on ADPCM coding. Return a tuple (sample, newstate) where the sample has the width specified in width.

**audioop.alaw2lin(fragment, width)**

Convert sound fragments in a-LAW encoding to linearly encoded sound fragments. a-LAW encoding always uses 8 bits samples, so width refers only to the sample width of the output fragment here.

**audioop.avg(fragment, width)**

Return the average over all samples in the fragment.

**audioop.avgpp(fragment, width)**

Return the average peak-peak value over all samples in the fragment. No filtering is done, so the usefulness of this routine is questionable.

**audioop.bias(fragment, width, bias)**

Return a fragment that is the original fragment with a bias added to each sample. Samples wrap around in case of overflow.

**audioop.cross(fragment, width)**

Return the number of zero crossings in the fragment passed as an argument.
audioop.findfactor(fragment, reference)

Return a factor \( F \) such that \( \text{rms}(\text{add}(\text{fragment}, \text{mul}(\text{reference}, -F))) \) is minimal, i.e., return the factor with which you should multiply \text{reference} to make it match as well as possible to \text{fragment}. The fragments should both contain 2-byte samples.

The time taken by this routine is proportional to \( \text{len(fragment)} \).

audioop.findfit(fragment, reference)

Try to match \text{reference} as well as possible to a portion of \text{fragment} (which should be the longer fragment). This is (conceptually) done by taking slices out of \text{fragment}, using \text{findfactor()} to compute the best match, and minimizing the result. The fragments should both contain 2-byte samples. Return a tuple \( (\text{offset}, \text{factor}) \) where \text{offset} is the (integer) offset into \text{fragment} where the optimal match started and \text{factor} is the (floating-point) factor as per \text{findfactor()}.

audioop.findmax(fragment, length)

Search \text{fragment} for a slice of length \text{length} samples (not bytes!) with maximum energy, i.e., return \( i \) for which \( \text{rms}(\text{fragment}[i*2:(i+\text{length})*2]) \) is maximal. The fragments should both contain 2-byte samples.

The routine takes time proportional to \( \text{len(fragment)} \).

audioop.getsample(fragment, width, index)

Return the value of sample \text{index} from the fragment.

audioop.lin2adpcm(fragment, width, state)

Convert samples to 4 bit Intel/DVI ADPCM encoding. ADPCM coding is an adaptive coding scheme, whereby each 4 bit number is the difference between one sample and the next, divided by a (varying) step. The Intel/DVI ADPCM algorithm has been selected for use by the IMA, so it may well become a standard.

\text{state} is a tuple containing the state of the coder. The coder returns a tuple \( (\text{adpcmfrag}, \text{newstate}) \), and the \text{newstate} should be passed to the next call of \text{lin2adpcm()}. In the initial call, \text{None} can be passed as the state. \text{adpcmfrag} is the ADPCM coded fragment packed 2 4-bit values per byte.

audioop.lin2alaw(fragment, width)

Convert samples in the audio fragment to a-LAW encoding and return this as a bytes object. a-LAW is an audio encoding format whereby you get a dynamic range of about 13 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

audioop.lin2lin(fragment, width, newwidth)

Convert samples between 1-, 2- and 4-byte formats.

\textbf{Note:} In some audio formats, such as .WAV files, 16 and 32 bit samples are signed, but 8 bit samples are unsigned. So when converting to 8 bit wide samples for these formats, you need to also add 128 to the result:

\[
\text{new\_frames} = \text{audioop.lin2lin(\text{frames}, \text{old\_width}, 1)} \\
\text{new\_frames} = \text{audioop.bias(\text{new\_frames}, 1, 128)}
\]

The same, in reverse, has to be applied when converting from 8 to 16 or 32 bit width samples.

audioop.lin2ulaw(fragment, width)

Convert samples in the audio fragment to u-LAW encoding and return this as a bytes object. u-LAW is an audio encoding format whereby you get a dynamic range of about 14 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

audioop.max(fragment, width)

Return the maximum of the absolute value of all samples in a fragment.

audioop.maxpp(fragment, width)

Return the maximum peak-peak value in the sound fragment.

audioop.minmax(fragment, width)

Return a tuple consisting of the minimum and maximum values of all samples in the sound fragment.
audioop.mul (fragment, width, factor)
Return a fragment that has all samples in the original fragment multiplied by the floating-point value factor. Samples are truncated in case of overflow.

audioop.ratecv (fragment, width, nchannels, inrate, outrate, state[, weightA[, weightB ]])
Convert the frame rate of the input fragment.

state is a tuple containing the state of the converter. The converter returns a tuple (newfragment, newstate), and newstate should be passed to the next call of ratecv(). The initial call should pass None as the state.

The weightA and weightB arguments are parameters for a simple digital filter and default to 1 and 0 respectively.

audioop.reverse (fragment, width)
Reverse the samples in a fragment and returns the modified fragment.

audioop.rms (fragment, width)
Return the root-mean-square of the fragment, i.e. \( \sqrt{\frac{\sum S_i^2}{n}} \).

This is a measure of the power in an audio signal.

audioop.tomono (fragment, width, lfactor, rfactor)
Convert a stereo fragment to a mono fragment. The left channel is multiplied by lfactor and the right channel by rfactor before adding the two channels to give a mono signal.

audioop.tostereo (fragment, width, lfactor, rfactor)
Generate a stereo fragment from a mono fragment. Each pair of samples in the stereo fragment are computed from the mono sample, whereby left channel samples are multiplied by lfactor and right channel samples by rfactor.

audioop.ulaw2lin (fragment, width)
Convert sound fragments in u-LAW encoding to linearly encoded sound fragments. u-LAW encoding always uses 8 bits samples, so width refers only to the sample width of the output fragment here.

Note that operations such as mul() or max() make no distinction between mono and stereo fragments, i.e. all samples are treated equal. If this is a problem the stereo fragment should be split into two mono fragments first and recombined later. Here is an example of how to do that:

```python
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
    rsample = audioop.tomono(sample, width, 0, 1)
    lsample = audioop.mul(lsample, width, lfactor)
    rsample = audioop.mul(rsample, width, rfactor)
    lsample = audioop.tostereo(lsample, width, 1, 0)
    rsample = audioop.tostereo(rsample, width, 0, 1)
    return audioop.add(lsample, rsample, width)
```

If you use the ADPCM coder to build network packets and you want your protocol to be stateless (i.e. to be able to tolerate packet loss) you should not only transmit the data but also the state. Note that you should send the initial state (the one you passed to lin2adpcm()) along to the decoder, not the final state (as returned by the coder). If you want to use struct.Struct to store the state in binary you can code the first element (the predicted value) in 16 bits and the second (the delta index) in 8.

The ADPCM coders have never been tried against other ADPCM coders, only against themselves. It could well be that I misinterpreted the standards in which case they will not be interoperable with the respective standards.

The find*() routines might look a bit funny at first sight. They are primarily meant to do echo cancellation. A reasonably fast way to do this is to pick the most energetic piece of the output sample, locate that in the input sample and subtract the whole output sample from the input sample:

```python
def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 800)      # one tenth second
    out_test = outputdata[pos*2:]
    in_test = inputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test)
    return audioop.add(out_test, inputdata[ipos*2:], width)
```
```python
# Optional (for better cancellation):
# factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)],
# out_test)
prefill = '\0'*(pos+ipos)*2
postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
outputdata = prefill + audioop.mul(outputdata,2,-factor) + postfill
return audioop.add(inputdata, outputdata, 2)
```

### 22.2 `aifc` — Read and write AIFF and AIFC files

**Source code:** Lib/aifc.py

This module provides support for reading and writing AIFF and AIFC files. AIFF is Audio Interchange File Format, a format for storing digital audio samples in a file. AIFF-C is a newer version of the format that includes the ability to compress the audio data.

**Note:** Some operations may only work under IRIX; these will raise `ImportError` when attempting to import the `cl` module, which is only available on IRIX.

Audio files have a number of parameters that describe the audio data. The sampling rate or frame rate is the number of times per second the sound is sampled. The number of channels indicate if the audio is mono, stereo, or quadro. Each frame consists of one sample per channel. The sample size is the size in bytes of each sample. Thus a frame consists of `nchannels*samplesize` bytes, and a second’s worth of audio consists of `nchannels*samplesize*framerate` bytes.

For example, CD quality audio has a sample size of two bytes (16 bits), uses two channels (stereo) and has a frame rate of 44,100 frames/second. This gives a frame size of 4 bytes (2*2), and a second’s worth occupies `2*2*44100` bytes (176,400 bytes).

Module `aifc` defines the following function:

```python
aifc.open (file, mode=None)
```

Open an AIFF or AIFF-C file and return an object instance with methods that are described below. The argument `file` is either a string naming a file or a `file object. mode` must be ‘r’ or ‘rb’ when the file must be opened for reading, or ‘w’ or ‘wb’ when the file must be opened for writing. If omitted, `file.mode` is used if it exists, otherwise ‘rb’ is used. When used for writing, the file object should be seekable, unless you know ahead of time how many samples you are going to write in total and use `writeframesraw()` and `setnframes()`.

Objects returned by `open()` when a file is opened for reading have the following methods:

- `aifc.getnchannels()`  
  Return the number of audio channels (1 for mono, 2 for stereo).

- `aifc.getsampwidth()`  
  Return the size in bytes of individual samples.

- `aifc.getframerate()`  
  Return the sampling rate (number of audio frames per second).

- `aifc.getnframes()`  
  Return the number of audio frames in the file.

- `aifc.getcomptype()`  
  Return a bytes array of length 4 describing the type of compression used in the audio file. For AIFF files, the returned value is b’NONE’.

- `aifc.getcompname()`  
  Return a bytes array convertible to a human-readable description of the type of compression used in the audio file. For AIFF files, the returned value is b’not compressed’.
aifc.getparams()
    Return a tuple consisting of all of the above values in the above order.

aifc.getmarkers()
    Return a list of markers in the audio file. A marker consists of a tuple of three elements. The first is the
    mark ID (an integer), the second is the mark position in frames from the beginning of the data (an integer),
    the third is the name of the mark (a string).

aifc.getmark(id)
    Return the tuple as described in getmarkers() for the mark with the given id.

aifc.readframes(nframes)
    Read and return the next nframes frames from the audio file. The returned data is a string containing for
    each frame the uncompressed samples of all channels.

aifc.rewind()
    Rewind the read pointer. The next readframes() will start from the beginning.

aifc.tell()
    Return the current frame number.

aifc.close()
    Close the AIFF file. After calling this method, the object can no longer be used.

Objects returned by open() when a file is opened for writing have all the above methods, except for
readframes() and setpos(). In addition the following methods exist. The get*() methods can only
be called after the corresponding set*() methods have been called. Before the first writeframes() or
writeframesraw(), all parameters except for the number of frames must be filled in.

aifc.aiff()
    Create an AIFF file. The default is that an AIFF-C file is created, unless the name of the file ends in
    ‘.aiff’ in which case the default is an AIFF file.

aifc.aifc()
    Create an AIFF-C file. The default is that an AIFF-C file is created, unless the name of the file ends in
    ‘.aiff’ in which case the default is an AIFF file.

aifc.setnchannels(nchannels)
    Specify the number of channels in the audio file.

aifc.setsampwidth(width)
    Specify the size in bytes of audio samples.

aifc.setframerate(rate)
    Specify the sampling frequency in frames per second.

aifc.setnframes(nframes)
    Specify the number of frames that are to be written to the audio file. If this parameter is not set, or not set
correctly, the file needs to support seeking.

aifc.setcomptype(type, name)
    Specify the compression type. If not specified, the audio data will not be compressed. In AIFF files,
    compression is not possible. The name parameter should be a human-readable description of the compression
    type as a bytes array, the type parameter should be a bytes array of length 4. Currently the following
    compression types are supported: b’NONE’, b’ULAW’, b’ALAW’, b’G722’.

aifc.setparams(nchannels, sampwidth, framerate, comptype, compname)
    Set all the above parameters at once. The argument is a tuple consisting of the various parameters. This
    means that it is possible to use the result of a getparams() call as argument to setparams().

aifc.setmark(id, pos, name)
    Add a mark with the given id (larger than 0), and the given name at the given position. This method can be
called at any time before close().

22.2. aifc — Read and write AIFF and AIFC files
The Python Library Reference, Release 3.3.3

```python
aifc.tell()
    Return the current write position in the output file. Useful in combination with setmark().

aifc.writeframes(data)
    Write data to the output file. This method can only be called after the audio file parameters have been set.

aifc.writeframesraw(data)
    Like writeframes(), except that the header of the audio file is not updated.

aifc.close()
    Close the AIFF file. The header of the file is updated to reflect the actual size of the audio data. After calling this method, the object can no longer be used.
```

## 22.3 sunau — Read and write Sun AU files

### Source code: Lib/sunau.py

The `sunau` module provides a convenient interface to the Sun AU sound format. Note that this module is interface-compatible with the modules `aifc` and `wave`.

An audio file consists of a header followed by the data. The fields of the header are:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>magic word</td>
<td>The four bytes <code>.snd</code>.</td>
</tr>
<tr>
<td>header size</td>
<td>Size of the header, including info, in bytes.</td>
</tr>
<tr>
<td>data size</td>
<td>Physical size of the data, in bytes.</td>
</tr>
<tr>
<td>encoding</td>
<td>Indicates how the audio samples are encoded.</td>
</tr>
<tr>
<td>sample rate</td>
<td>The sampling rate.</td>
</tr>
<tr>
<td># of channels</td>
<td>The number of channels in the samples.</td>
</tr>
<tr>
<td>info</td>
<td>ASCII string giving a description of the audio file (padded with null bytes).</td>
</tr>
</tbody>
</table>

Apart from the info field, all header fields are 4 bytes in size. They are all 32-bit unsigned integers encoded in big-endian byte order.

The `sunau` module defines the following functions:

- **`sunau.open(file, mode)`**
  - If `file` is a string, open the file by that name, otherwise treat it as a seekable file-like object. `mode` can be any of
    - `'r'` Read only mode.
    - `'w'` Write only mode.
  - Note that it does not allow read/write files.
  - A `mode` of `'r'` returns a `AU_read` object, while a `mode` of `'w'` or `'wb'` returns a `AU_write` object.

- **`sunau.openfp(file, mode)`**
  - A synonym for `open()`, maintained for backwards compatibility.

The `sunau` module defines the following exception:

- **`exception sunau.Error`**
  - An error raised when something is impossible because of Sun AU specs or implementation deficiency.

The `sunau` module defines the following data items:

- **`sunau.AUDIO_FILE_MAGIC`**
  - An integer every valid Sun AU file begins with, stored in big-endian form. This is the string `.snd` interpreted as an integer.

- **`sunau.AUDIO_FILE_ENCODING_MULAW_8`**
- **`sunau.AUDIO_FILE_ENCODING_LINEAR_8`**
Values of the encoding field from the AU header which are supported by this module.

Additional known values of the encoding field from the AU header, but which are not supported by this module.

### 22.3.1 AU_read Objects

AU_read objects, as returned by `open()` above, have the following methods:

- **AU_read.close()**
  - Close the stream, and make the instance unusable. (This is called automatically on deletion.)

- **AU_read.getnchannels()**
  - Returns number of audio channels (1 for mono, 2 for stereo).

- **AU_read.getsampwidth()**
  - Returns sample width in bytes.

- **AU_read.getframerate()**
  - Returns sampling frequency.

- **AU_read.getnframes()**
  - Returns number of audio frames.

- **AU_read.getcomptype()**
  - Returns compression type. Supported compression types are ‘ULAW’, ‘ALAW’ and ‘NONE’.

- **AU_read.getcompname()**
  - Human-readable version of `getcomptype()`. The supported types have the respective names ‘CCITT G.711 u-law’, ‘CCITT G.711 A-law’ and ‘not compressed’.

- **AU_read.getparams()**
  - Returns a tuple `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, equivalent to output of the `get*()` methods.

- **AU_read.readframes(n)**
  - Reads and returns at most `n` frames of audio, as a string of bytes. The data will be returned in linear format. If the original data is in u-LAW format, it will be converted.

- **AU_read.rewind()**
  - Rewind the file pointer to the beginning of the audio stream.

The following two methods define a term “position” which is compatible between them, and is otherwise implementation dependent.

- **AU_read.setpos(pos)**
  - Set the file pointer to the specified position. Only values returned from `tell()` should be used for `pos`.

- **AU_read.tell()**
  - Return current file pointer position. Note that the returned value has nothing to do with the actual position in the file.

The following two functions are defined for compatibility with the `aiff`, and don’t do anything interesting.
AU_read.getmarkers()
    Returns None.

AU_read.getmark(id)
    Raise an error.

### 22.3.2 AU_write Objects

AU_write objects, as returned by `open()` above, have the following methods:

- **AU_write.setnchannels(n)**
  - Set the number of channels.

- **AU_write.setsampwidth(n)**
  - Set the sample width (in bytes.)

- **AU_write.setframerate(n)**
  - Set the frame rate.

- **AU_write.setnframes(n)**
  - Set the number of frames. This can be later changed, when and if more frames are written.

- **AU_write.setcomptype(type, name)**
  - Set the compression type and description. Only 'NONE' and 'ULAW' are supported on output.

- **AU_write.setparams(tuple)**
  - The tuple should be (nchannels, sampwidth, framerate, nframes, comptype, compname), with values valid for the set*() methods. Set all parameters.

- **AU_write.tell()**
  - Return current position in the file, with the same disclaimer for the AU_read.tell() and AU_read.setpos() methods.

- **AU_write.writeframesraw(data)**
  - Write audio frames, without correcting nframes.

- **AU_write.writeframes(data)**
  - Write audio frames and make sure nframes is correct.

- **AU_write.close()**
  - Make sure nframes is correct, and close the file.
  - This method is called upon deletion.

Note that it is invalid to set any parameters after calling writeframes() or writeframesraw().

### 22.4 wave — Read and write WAV files

**Source code:** Lib/wave.py

The `wave` module provides a convenient interface to the WAV sound format. It does not support compression/decompression, but it does support mono/stereo.

The `wave` module defines the following function and exception:

- **wave.open(file, mode=None)**
  - If file is a string, open the file by that name, otherwise treat it as a seekable file-like object. mode can be any of
    - `'r'`, `'rb'` Read only mode.
    - `'w'`, `'wb'` Write only mode.
Note that it does not allow read/write WAV files.

A `mode` of `'r'` or `'rb'` returns a `Wave_read` object, while a `mode` of `'w'` or `'wb'` returns a `Wave_write` object. If `mode` is omitted and a file-like object is passed as `file`, `file.mode` is used as the default value for `mode` (the `'b'` flag is still added if necessary).

If you pass in a file-like object, the wave object will not close it when its `close()` method is called; it is the caller’s responsibility to close the file object.

```python
wave.openfp(file, mode)
```
A synonym for `open()`, maintained for backwards compatibility.

### exception `wave.Error`
An error raised when something is impossible because it violates the WAV specification or hits an implementation deficiency.

#### 22.4.1 Wave_read Objects

Wave_read objects, as returned by `open()`, have the following methods:

```python
Wave_read.close()
```
Close the stream if it was opened by `wave`, and make the instance unusable. This is called automatically on object collection.

```python
Wave_read.getnchannels()
```
Returns number of audio channels (1 for mono, 2 for stereo).

```python
Wave_read.getsampwidth()
```
Returns sample width in bytes.

```python
Wave_read.getframerate()
```
Returns sampling frequency.

```python
Wave_read.getnframes()
```
Returns number of audio frames.

```python
Wave_read.getcomptype()
```
Returns compression type (`'NONE'` is the only supported type).

```python
Wave_read.getcompname()
```
Human-readable version of `getcomptype()`. Usually ‘not compressed’ parallels ‘NONE’.

```python
Wave_read.getparams()
```
Returns a tuple `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, equivalent to output of the `get*()` methods.

```python
Wave_read.readframes(n)
```
Reads and returns at most `n` frames of audio, as a string of bytes.

```python
Wave_read.rewind()
```
Rewind the file pointer to the beginning of the audio stream.

The following two methods are defined for compatibility with the `aifc` module, and don’t do anything interesting.

```python
Wave_read.getmarkers()
```
Returns `None`.

```python
Wave_read.getmark(id)
```
Raise an error.

The following two methods define a term “position” which is compatible between them, and is otherwise implementation dependent.

```python
Wave_read.setpos(pos)
```
Set the file pointer to the specified position.
Wave_read.tell()
    Return current file pointer position.

## 22.4.2 Wave_write Objects

Wave_write objects, as returned by `open()`, have the following methods:

### Wave_write.close()

Make sure `nframes` is correct, and close the file if it was opened by `wave`. This method is called upon object collection.

### Wave_write.setnchannels(n)

Set the number of channels.

### Wave_write.setsampwidth(n)

Set the sample width to `n` bytes.

### Wave_write.setframerate(n)

Set the frame rate to `n`. Changed in version 3.2: A non-integral input to this method is rounded to the nearest integer.

### Wave_write.setnframes(n)

Set the number of frames to `n`. This will be changed later if more frames are written.

### Wave_write.setcomptype(type, name)

Set the compression type and description. At the moment, only compression type `NONE` is supported, meaning no compression.

### Wave_write.setparams(tuple)

The `tuple` should be `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, with values valid for the `set*()` methods. Sets all parameters.

### Wave_write.tell()

Return current position in the file, with the same disclaimer for the `Wave_read.tell()` and `Wave_read.setpos()` methods.

### Wave_write.writeframesraw(data)

Write audio frames, without correcting `nframes`.

### Wave_write.writeframes(data)

Write audio frames and make sure `nframes` is correct.

Note that it is invalid to set any parameters after calling `writeframes()` or `writeframesraw()`, and any attempt to do so will raise `wave.Error`.

## 22.5 chunk — Read IFF chunked data

This module provides an interface for reading files that use EA IFF 85 chunks. This format is used in at least the Audio Interchange File Format (AIFF/AIFF-C) and the Real Media File Format (RMFF). The WAVE audio file format is closely related and can also be read using this module.

A chunk has the following structure:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>Chunk ID</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Size of chunk in big-endian byte order, not including the header</td>
</tr>
<tr>
<td>8</td>
<td><code>n</code></td>
<td>Data bytes, where <code>n</code> is the size given in the preceding field</td>
</tr>
<tr>
<td><code>8 + n</code></td>
<td>0 or 1</td>
<td>Pad byte needed if <code>n</code> is odd and chunk alignment is used</td>
</tr>
</tbody>
</table>

The ID is a 4-byte string which identifies the type of chunk.

---

The size field (a 32-bit value, encoded using big-endian byte order) gives the size of the chunk data, not including the 8-byte header.

Usually an IFF-type file consists of one or more chunks. The proposed usage of the Chunk class defined here is to instantiate an instance at the start of each chunk and read from the instance until it reaches the end, after which a new instance can be instantiated. At the end of the file, creating a new instance will fail with a EOFError exception.

class chunk.Chunk (file, align=True, bigendian=True, inclheader=False)

Class which represents a chunk. The file argument is expected to be a file-like object. An instance of this class is specifically allowed. The only method that is needed is read(). If the methods seek() and tell() are present and don’t raise an exception, they are also used. If these methods are present and raise an exception, they are expected to not have altered the object. If the optional argument align is true, chunks are assumed to be aligned on 2-byte boundaries. If align is false, no alignment is assumed. The default value is true. If the optional argument bigendian is false, the chunk size is assumed to be in little-endian order. This is needed for WAVE audio files. The default value is true. If the optional argument inclheader is true, the size given in the chunk header includes the size of the header. The default value is false.

A Chunk object supports the following methods:

- getname ()
  Returns the name (ID) of the chunk. This is the first 4 bytes of the chunk.

- getsize ()
  Returns the size of the chunk.

- close ()
  Close and skip to the end of the chunk. This does not close the underlying file.

The remaining methods will raise OSError if called after the close() method has been called. Before Python 3.3, they used to raise IOError, now an alias of OSError.

- isatty ()
  Returns False.

- seek (pos, whence=0)
  Set the chunk’s current position. The whence argument is optional and defaults to 0 (absolute file positioning); other values are 1 (seek relative to the current position) and 2 (seek relative to the file’s end). There is no return value. If the underlying file does not allow seek, only forward seeks are allowed.

- tell ()
  Return the current position into the chunk.

- read (size=-1)
  Read at most size bytes from the chunk (less if the read hits the end of the chunk before obtaining size bytes). If the size argument is negative or omitted, read all data until the end of the chunk. The bytes are returned as a string object. An empty string is returned when the end of the chunk is encountered immediately.

- skip ()
  Skip to the end of the chunk. All further calls to read() for the chunk will return "". If you are not interested in the contents of the chunk, this method should be called so that the file points to the start of the next chunk.

22.6 colorsys — Conversions between color systems

Source code: Lib/colors.py

The colorsys module defines bidirectional conversions of color values between colors expressed in the RGB (Red Green Blue) color space used in computer monitors and three other coordinate systems: YIQ, HLS (Hue

22.6. colorsys — Conversions between color systems 975
Lightness Saturation) and HSV (Hue Saturation Value). Coordinates in all of these color spaces are floating point values. In the YIQ space, the Y coordinate is between 0 and 1, but the I and Q coordinates can be positive or negative. In all other spaces, the coordinates are all between 0 and 1.

See Also:

The `colorsys` module defines the following functions:

- `colorsys.rgb_to_yiq(r, g, b)`: Convert the color from RGB coordinates to YIQ coordinates.
- `colorsys.yiq_to_rgb(y, i, q)`: Convert the color from YIQ coordinates to RGB coordinates.
- `colorsys.rgb_to_hls(r, g, b)`: Convert the color from RGB coordinates to HLS coordinates.
- `colorsys.hls_to_rgb(h, l, s)`: Convert the color from HLS coordinates to RGB coordinates.
- `colorsys.rgb_to_hsv(r, g, b)`: Convert the color from RGB coordinates to HSV coordinates.
- `colorsys.hsv_to_rgb(h, s, v)`: Convert the color from HSV coordinates to RGB coordinates.

Example:
```python
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

22.7 imghdr — Determine the type of an image

Source code: Lib/imghdr.py

The `imghdr` module determines the type of image contained in a file or byte stream.

The `imghdr` module defines the following function:

- `imghdr.what(filename, h=None)`: Tests the image data contained in the file named by `filename`, and returns a string describing the image type. If optional `h` is provided, the `filename` is ignored and `h` is assumed to contain the byte stream to test.

The following image types are recognized, as listed below with the return value from `what()`:

<table>
<thead>
<tr>
<th>Value</th>
<th>Image format</th>
</tr>
</thead>
<tbody>
<tr>
<td>rgb</td>
<td>SGI ImgLib Files</td>
</tr>
<tr>
<td>gif</td>
<td>GIF 87a and 89a Files</td>
</tr>
<tr>
<td>pbm</td>
<td>Portable Bitmap Files</td>
</tr>
<tr>
<td>pgm</td>
<td>Portable Graymap Files</td>
</tr>
<tr>
<td>ppm</td>
<td>Portable Pixmap Files</td>
</tr>
<tr>
<td>tiff</td>
<td>TIFF Files</td>
</tr>
<tr>
<td>rast</td>
<td>Sun Raster Files</td>
</tr>
<tr>
<td>xbm</td>
<td>X Bitmap Files</td>
</tr>
<tr>
<td>jpeg</td>
<td>JPEG data in JFIF or Exif formats</td>
</tr>
<tr>
<td>bmp</td>
<td>BMP files</td>
</tr>
<tr>
<td>png</td>
<td>Portable Network Graphics</td>
</tr>
</tbody>
</table>
You can extend the list of file types `imghdr` can recognize by appending to this variable:

```python
imghdr.tests
A list of functions performing the individual tests. Each function takes two arguments: the byte-stream and an open file-like object. When `what()` is called with a byte-stream, the file-like object will be `None`.

The test function should return a string describing the image type if the test succeeded, or `None` if it failed.

Example:

```python
>>> import imghdr
>>> imghdr.what('bass.gif')
'gif'
```

## 22.8 sndhdr — Determine type of sound file

**Source code:** Lib/sndhdr.py

The `sndhdr` provides utility functions which attempt to determine the type of sound data which is in a file. When these functions are able to determine what type of sound data is stored in a file, they return a tuple `(type, sampling_rate, channels, frames, bits_per_sample)`. The value for `type` indicates the data type and will be one of the strings `'aifc'`, `'aiff'`, `'au'`, `'hcom'`, `'snd'`, `'sndt'`, `'voc'`, `'wav'`, `'8svx'`, `'sb'`, `'ub'`, or `'ul'`. The `sampling_rate` will be either the actual value or 0 if unknown or difficult to decode. Similarly, `channels` will be either the number of channels or 0 if it cannot be determined or if the value is difficult to decode. The value for `frames` will be either the number of frames or `-1`. The last item in the tuple, `bits_per_sample`, will either be the sample size in bits or `'A'` for A-LAW or `'U'` for u-LAW.

```python
sndhdr.what(filename)
```

Determines the type of sound data stored in the file `filename` using `whathdr()`. If it succeeds, returns a tuple as described above, otherwise `None` is returned.

```python
sndhdr.whathdr(filename)
```

Determines the type of sound data stored in a file based on the file header. The name of the file is given by `filename`. This function returns a tuple as described above on success, or `None`.

## 22.9 ossaudiodev — Access to OSS-compatible audio devices

**Platforms:** Linux, FreeBSD

This module allows you to access the OSS (Open Sound System) audio interface. OSS is available for a wide range of open-source and commercial Unices, and is the standard audio interface for Linux and recent versions of FreeBSD. Changed in version 3.3: Operations in this module now raise `OSError` where `IOError` was raised.

**See Also:**

- Open Sound System Programmer’s Guide: the official documentation for the OSS C API

The module defines a large number of constants supplied by the OSS device driver; see `<sys/soundcard.h>` on either Linux or FreeBSD for a listing.

```python
ossaudiodev
```

defines the following variables and functions:

```python
exception ossaudiodev.OSSAudioError
```

This exception is raised on certain errors. The argument is a string describing what went wrong.

(If `ossaudiodev` receives an error from a system call such as `open()`, `write()`, or `ioctl()`, it raises `OSError`. Errors detected directly by `ossaudiodev` result in `OSSAudioError`.)

(For backwards compatibility, the exception class is also available as `ossaudiodev.error`.)

```python
ossaudiodev.open(mode)
```
ossaudiodev.open(device, mode)

Open an audio device and return an OSS audio device object. This object supports many file-like methods, such as read(), write(), and fileno() (although there are subtle differences between conventional Unix read/write semantics and those of OSS audio devices). It also supports a number of audio-specific methods; see below for the complete list of methods.

device is the audio device filename to use. If it is not specified, this module first looks in the environment variable AUDIODEV for a device to use. If not found, it falls back to /dev/dsp.

mode is one of ‘r’ for read-only (record) access, ‘w’ for write-only (playback) access and ‘rw’ for both. Since many sound cards only allow one process to have the recorder or player open at a time, it is a good idea to open the device only for the activity needed. Further, some sound cards are half-duplex: they can be opened for reading or writing, but not both at once.

Note the unusual calling syntax: the first argument is optional, and the second is required. This is a historical artifact for compatibility with the older linuxaudiodev module which ossaudiodev supersedes.

ossaudiodev.openmixer([device])

Open a mixer device and return an OSS mixer device object. device is the mixer device filename to use. If it is not specified, this module first looks in the environment variable MIXERDEV for a device to use. If not found, it falls back to /dev/mixer.

22.9.1 Audio Device Objects

Before you can write to or read from an audio device, you must call three methods in the correct order:

1. setfmt() to set the output format
2. channels() to set the number of channels
3. speed() to set the sample rate

Alternately, you can use the setparameters() method to set all three audio parameters at once. This is more convenient, but may not be as flexible in all cases.

The audio device objects returned by open() define the following methods and (read-only) attributes:

oss_audio_device.close()

Explicitly close the audio device. When you are done writing to or reading from an audio device, you should explicitly close it. A closed device cannot be used again.

oss_audio_device.fileno()

Return the file descriptor associated with the device.

oss_audio_device.read(size)

Read size bytes from the audio input and return them as a Python string. Unlike most Unix device drivers, OSS audio devices in blocking mode (the default) will block read() until the entire requested amount of data is available.

oss_audio_device.write(data)

Write the Python string data to the audio device and return the number of bytes written. If the audio device is in blocking mode (the default), the entire string is always written (again, this is different from usual Unix device semantics). If the device is in non-blocking mode, some data may not be written —see writeall().

oss_audio_device.writeall(data)

Write the entire Python string data to the audio device: waits until the audio device is able to accept data, writes as much data as it will accept, and repeats until data has been completely written. If the device is in blocking mode (the default), this has the same effect as write(); writeall() is only useful in non-blocking mode. Has no return value, since the amount of data written is always equal to the amount of data supplied.

Changed in version 3.2: Audio device objects also support the context manager protocol, i.e. they can be used in a with statement. The following methods each map to exactly one ioctl() system call. The correspondence is obvious: for example, setfmt() corresponds to the SNDCTL_DSP_SETFMT ioctl, and sync() to
SNDCTL_DSP_SYNC (this can be useful when consulting the OSS documentation). If the underlying ioctl() fails, they all raise OSError.

oss_audio_device.nonblock()
Put the device into non-blocking mode. Once in non-blocking mode, there is no way to return it to blocking mode.

oss_audio_device.getfms()
Return a bitmask of the audio output formats supported by the soundcard. Some of the formats supported by OSS are:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFMT_MU_LAW</td>
<td>a logarithmic encoding (used by Sun .au files and /dev/audio)</td>
</tr>
<tr>
<td>AFMT_A_LAW</td>
<td>a logarithmic encoding</td>
</tr>
<tr>
<td>AFMT_IMA_ADPCM</td>
<td>a 4:1 compressed format defined by the Interactive Multimedia Association</td>
</tr>
<tr>
<td>AFMT_U8</td>
<td>Unsigned, 8-bit audio</td>
</tr>
<tr>
<td>AFMT_S16_LE</td>
<td>Signed, 16-bit audio, little-endian byte order (as used by Intel processors)</td>
</tr>
<tr>
<td>AFMT_S16_BE</td>
<td>Signed, 16-bit audio, big-endian byte order (as used by 68k, PowerPC, Sparc)</td>
</tr>
<tr>
<td>AFMT_S8</td>
<td>Signed, 8-bit audio</td>
</tr>
<tr>
<td>AFMT_U16_LE</td>
<td>Unsigned, 16-bit little-endian audio</td>
</tr>
<tr>
<td>AFMT_U16_BE</td>
<td>Unsigned, 16-bit big-endian audio</td>
</tr>
</tbody>
</table>

Consult the OSS documentation for a full list of audio formats, and note that most devices support only a subset of these formats. Some older devices only support AFMT_U8; the most common format used today is AFMT_S16_LE.

oss_audio_device.setfmt(format)
Try to set the current audio format to format—see getfmts() for a list. Returns the audio format that the device was set to, which may not be the requested format. May also be used to return the current audio format—do this by passing an “audio format” of AFMT_QUERY.

oss_audio_device.channels(nchannels)
Set the number of output channels to nchannels. A value of 1 indicates monophonic sound, 2 stereophonic. Some devices may have more than 2 channels, and some high-end devices may not support mono. Returns the number of channels the device was set to.

oss_audio_device.speed(samplerate)
Try to set the audio sampling rate to samplerate samples per second. Returns the rate actually set. Most sound devices don’t support arbitrary sampling rates. Common rates are:

<table>
<thead>
<tr>
<th>Rate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8000</td>
<td>default rate for /dev/audio</td>
</tr>
<tr>
<td>11025</td>
<td>speech recording</td>
</tr>
<tr>
<td>22050</td>
<td></td>
</tr>
<tr>
<td>44100</td>
<td>CD quality audio (at 16 bits/sample and 2 channels)</td>
</tr>
<tr>
<td>96000</td>
<td>DVD quality audio (at 24 bits/sample)</td>
</tr>
</tbody>
</table>

oss_audio_device.sync()
Wait until the sound device has played every byte in its buffer. (This happens implicitly when the device is closed.) The OSS documentation recommends closing and re-opening the device rather than using sync().

oss_audio_device.reset()
Immediately stop playing or recording and return the device to a state where it can accept commands. The OSS documentation recommends closing and re-opening the device after calling reset().

oss_audio_device.post()
Tell the driver that there is likely to be a pause in the output, making it possible for the device to handle the pause more intelligently. You might use this after playing a spot sound effect, before waiting for user input, or before doing disk I/O.

The following convenience methods combine several ioctls, or one ioctl and some simple calculations.

oss_audio_device.setparameters(format, nchannels, samplerate[, strict=False])
Set the key audio sampling parameters—sample format, number of channels, and sampling rate—in one method call. format, nchannels, and samplerate should be as specified in the setfmt(), channels(),
and `speed()` methods. If `strict` is true, `setparameters()` checks to see if each parameter was actually set to the requested value, and raises `OSSAudioError` if not. Returns a tuple `(format, nchannels, samplerate)` indicating the parameter values that were actually set by the device driver (i.e., the same as the return values of `setfmt()`, `channels()`, and `speed()`).

For example,

```python
(fmt, channels, rate) = dsp.setparameters(fmt, channels, rate)
```

is equivalent to

```python
fmt = dsp.setfmt(fmt)
channels = dsp.channels(channels)
rate = dsp.rate(rate)
```

`oss_audio_device.bufsize()`

Returns the size of the hardware buffer, in samples.

`oss_audio_device.obufcount()`

Returns the number of samples that are in the hardware buffer yet to be played.

`oss_audio_device.obuffree()`

Returns the number of samples that could be queued into the hardware buffer to be played without blocking.

Audio device objects also support several read-only attributes:

- `oss_audio_device.closed`
  - Boolean indicating whether the device has been closed.

- `oss_audio_device.name`
  - String containing the name of the device file.

- `oss_audio_device.mode`
  - The I/O mode for the file, either "r", "rw", or "w".

## 22.9.2 Mixer Device Objects

The mixer object provides two file-like methods:

- `oss_mixer_device.close()`
  - This method closes the open mixer device file. Any further attempts to use the mixer after this file is closed will raise an `OSError`.

- `oss_mixer_device.fileno()`
  - Returns the file handle number of the open mixer device file.

Changed in version 3.2: Mixer objects also support the context manager protocol. The remaining methods are specific to audio mixing:

- `oss_mixer_device.controls()`
  - This method returns a bitmask specifying the available mixer controls ("Control" being a specific mixable "channel", such as `SOUND_MIXER_PCM` or `SOUND_MIXER_SYNTH`). This bitmask indicates a subset of all available mixer controls—the `SOUND_MIXER_*` constants defined at module level. To determine if, for example, the current mixer object supports a PCM mixer, use the following Python code:

```python
mixer=ossaudiodev.openmixer()
if mixer.controls() & (1 << ossaudiodev.SOUND_MIXER_PCM):
    # PCM is supported
    ... code ...
```

For most purposes, the `SOUND_MIXER_VOLUME` (master volume) and `SOUND_MIXER_PCM` controls should suffice—but code that uses the mixer should be flexible when it comes to choosing mixer controls. On the Gravis Ultrasound, for example, `SOUND_MIXER_VOLUME` does not exist.
oss_mixer_device.stereocontrols()

Returns a bitmask indicating stereo mixer controls. If a bit is set, the corresponding control is stereo; if it is unset, the control is either monophonic or not supported by the mixer (use in combination with controls() to determine which).

See the code example for the controls() function for an example of getting data from a bitmask.

oss_mixer_device.reccontrols()

Returns a bitmask specifying the mixer controls that may be used to record. See the code example for controls() for an example of reading from a bitmask.

oss_mixer_device.get(control)

Returns the volume of a given mixer control. The returned volume is a 2-tuple (left_volume, right_volume). Volumes are specified as numbers from 0 (silent) to 100 (full volume). If the control is monophonic, a 2-tuple is still returned, but both volumes are the same.

 Raises OSSAudioError if an invalid control was specified, or OSError if an unsupported control is specified.

oss_mixer_device.set(control, (left, right))

Sets the volume for a given mixer control to (left, right). left and right must be ints and between 0 (silent) and 100 (full volume). On success, the new volume is returned as a 2-tuple. Note that this may not be exactly the same as the volume specified, because of the limited resolution of some soundcard’s mixers.

 Raises OSSAudioError if an invalid mixer control was specified, or if the specified volumes were out-of-range.

oss_mixer_device.get_recsrc()

This method returns a bitmask indicating which control(s) are currently being used as a recording source.

oss_mixer_device.set_recsrc(bitmask)

Call this function to specify a recording source. Returns a bitmask indicating the new recording source (or sources) if successful; raises OSError if an invalid source was specified. To set the current recording source to the microphone input:

    mixer.setrecesrc(1 << ossaudiodev.SOUND_MIXER_MIC)
The modules described in this chapter help you write software that is independent of language and locale by providing mechanisms for selecting a language to be used in program messages or by tailoring output to match local conventions.

The list of modules described in this chapter is:

### 23.1 gettext — Multilingual internationalization services

Source code: Lib/gettext.py

The `gettext` module provides internationalization (I18N) and localization (L10N) services for your Python modules and applications. It supports both the GNU `gettext` message catalog API and a higher level, class-based API that may be more appropriate for Python files. The interface described below allows you to write your module and application messages in one natural language, and provide a catalog of translated messages for running under different natural languages.

Some hints on localizing your Python modules and applications are also given.

#### 23.1.1 GNU gettext API

The `gettext` module defines the following API, which is very similar to the GNU `gettext` API. If you use this API you will affect the translation of your entire application globally. Often this is what you want if your application is monolingual, with the choice of language dependent on the locale of your user. If you are localizing a Python module, or if your application needs to switch languages on the fly, you probably want to use the class-based API instead.

- `gettext.bindtextdomain(domain, localedir=None)`
  
  Bind the `domain` to the locale directory `localedir`. More concretely, `gettext` will look for binary `.mo` files for the given domain using the path (on Unix): `localedir/language/LC_MESSAGES/domain.mo`, where `languages` is searched for in the environment variables `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, and `LANG` respectively.

  If `localedir` is omitted or `None`, then the current binding for `domain` is returned.  

- `gettext.bind_textdomain_codeset(domain, codeset=None)`

  Bind the `domain` to `codeset`, changing the encoding of strings returned by the `gettext()` family of functions. If `codeset` is omitted, then the current binding is returned.

  The default locale directory is system dependent: for example, on RedHat Linux it is `/usr/share/locale`, but on Solaris it is `/usr/lib/locale`. The `gettext` module does not try to support these system dependent defaults; instead its default is `sys.prefix/share/locale`. For this reason, it is always best to call `bindtextdomain()` with an explicit absolute path at the start of your application.
gettext.textdomain(domain=None)

Change or query the current global domain. If domain is None, then the current global domain is returned, otherwise the global domain is set to domain, which is returned.

ggettext(message)

Return the localized translation of message, based on the current global domain, language, and locale directory. This function is usually aliased as _() in the local namespace (see examples below).

ggettext.lgettext(message)

Equivalent to gettext(), but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with bind_textdomain_codeset().

ggettext.dgettext(domain, message)

Like gettext(), but look the message up in the specified domain.

ggettext.ldgettext(domain, message)

Equivalent to dgettext(), but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with bind_textdomain_codeset().

ggettext.ngettext(singular, plural, n)

Like gettext(), but consider plural forms. If a translation is found, apply the plural formula to n, and return the resulting message (some languages have more than two plural forms). If no translation is found, return singular if n is 1; return plural otherwise.

The Plural formula is taken from the catalog header. It is a C or Python expression that has a free variable n; the expression evaluates to the index of the plural in the catalog. See the GNU gettext documentation for the precise syntax to be used in .po files and the formulas for a variety of languages.

ggettext.lngettext(singular, plural, n)

Equivalent to ngettext(), but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with bind_textdomain_codeset().

ggettext.dngettext(domain, singular, plural, n)

Like ngettext(), but look the message up in the specified domain.

ggettext.ldngettext(domain, singular, plural, n)

Equivalent to dngettext(), but the translation is returned in the preferred system encoding, if no other encoding was explicitly set with bind_textdomain_codeset().

Note that GNU gettext also defines a dcgettext() method, but this was deemed not useful and so it is currently unimplemented.

Here’s an example of typical usage for this API:

```python
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print(_('This is a translatable string.'))
```

### 23.1.2 Class-based API

The class-based API of the gettext module gives you more flexibility and greater convenience than the GNU gettext API. It is the recommended way of localizing your Python applications and modules. gettext defines a “translations” class which implements the parsing of GNU .mo format files, and has methods for returning strings. Instances of this “translations” class can also install themselves in the built-in namespace as the function _().

ggettext.find(domain, localedir=None, languages=None, all=False)

This function implements the standard .mo file search algorithm. It takes a domain, identical to what textdomain() takes. Optional localedir is as in bindtextdomain() Optional languages is a list of strings, where each string is a language code.
If `localedir` is not given, then the default system locale directory is used. ² If `languages` is not given, then the following environment variables are searched: `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, and `LANG`. The first one returning a non-empty value is used for the `languages` variable. The environment variables should contain a colon separated list of languages, which will be split on the colon to produce the expected list of language code strings.

`find()` then expands and normalizes the languages, and then iterates through them, searching for an existing file built of these components:

```
localedir/language/LC_MESSAGES/domain.mo
```

The first such file name that exists is returned by `find()`. If no such file is found, then `None` is returned. If `all` is given, it returns a list of all file names, in the order in which they appear in the languages list or the environment variables.

```
gettext.translation(domain, localedir=None, languages=None, class_=None, fallback=False, codeset=None)
```

Return a `Translations` instance based on the `domain`, `localedir`, and `languages`, which are first passed to `find()` to get a list of the associated `.mo` file paths. Instances with identical `.mo` file names are cached. The actual class instantiated is either `class_` if provided, otherwise `GNUTranslations`. The class’s constructor must take a single `file object` argument. If provided, `codeset` will change the charset used to encode translated strings in the `lgettext()` and `lngettext()` methods.

If multiple files are found, later files are used as fallbacks for earlier ones. To allow setting the fallback, `copy.copy()` is used to clone each translation object from the cache; the actual instance data is still shared with the cache.

If no `.mo` file is found, this function raises `OSError` if `fallback` is false (which is the default), and returns a `NullTranslations` instance if `fallback` is true. Changed in version 3.3: `IOSError` used to be raised instead of `OSError`.

```
gettext.install(domain, localedir=None, codeset=None, names=None)
```

This installs the function `_()` in Python’s builtins namespace, based on `domain`, `localedir`, and `codeset` which are passed to the function `translation()`. For the `names` parameter, please see the description of the translation object’s `install()` method.

As seen below, you usually mark the strings in your application that are candidates for translation, by wrapping them in a call to the `_()` function, like this:

```
print(_(‘This string will be translated.’))
```

For convenience, you want the `_()` function to be installed in Python’s builtins namespace, so it is easily accessible in all modules of your application.

### The `NullTranslations` class

Translation classes are what actually implement the translation of original source file message strings to translated message strings. The base class used by all translation classes is `NullTranslations`; this provides the basic interface you can use to write your own specialized translation classes. Here are the methods of `NullTranslations`:

```
class gettext.NullTranslations(fp=None)
```

Takes an optional `file object` `fp`, which is ignored by the base class. Initializes “protected” instance variables `__info` and `__charset` which are set by derived classes, as well as `__fallback`, which is set through `add_fallback()`. It then calls `self.__parse(fp)` if `fp` is not `None`.

```
__parse(fp)
```

No-op’d in the base class, this method takes file object `fp`, and reads the data from the file, initializing its message catalog. If you have an unsupported message catalog file format, you should override this method to parse your format.

² See the footnote for `bindtextdomain()` above.
add_fallback (fallback)
Add "fallback" as the fallback object for the current translation object. A translation object should consult the fallback if it cannot provide a translation for a given message.

gettext (message)
If a fallback has been set, forward gettext () to the fallback. Otherwise, return the translated message. Overridden in derived classes.

lgettext (message)
If a fallback has been set, forward lgettext () to the fallback. Otherwise, return the translated message. Overridden in derived classes.

ngettext (singular, plural, n)
If a fallback has been set, forward ngettext () to the fallback. Otherwise, return the translated message. Overridden in derived classes.

lngettext (singular, plural, n)
If a fallback has been set, forward lngettext () to the fallback. Otherwise, return the translated message. Overridden in derived classes.

info ()
Return the “protected” _info variable.

charset ()
Return the “protected” _charset variable, which is the encoding of the message catalog file.

output_charset ()
Return the “protected” _output_charset variable, which defines the encoding used to return translated messages in lgettext () and lngettext ()

set_output_charset (charset)
Change the “protected” _output_charset variable, which defines the encoding used to return translated messages.

install (names=None)
This method installs self.gettext () into the built-in namespace, binding it to _.

If the names parameter is given, it must be a sequence containing the names of functions you want to install in the builtins namespace in addition to _(). Supported names are ‘gettext’ (bound to self.gettext ()), ‘ngettext’ (bound to self.ngettext ()), ‘lgettext’ and ‘lngettext’.

Note that this is only one way, albeit the most convenient way, to make the _() function available to your application. Because it affects the entire application globally, and specifically the built-in namespace, localized modules should never install _(). Instead, they should use this code to make _() available to their module:

import gettext
t = gettext.translation(‘mymodule’, ...)
_ = t.gettext

This puts _() only in the module’s global namespace and so only affects calls within this module.

The GNUTranslations class

The gettext module provides one additional class derived from NullTranslations: GNUTranslations. This class overrides _parse() to enable reading GNU gettext format .mo files in both big-endian and little-endian format.

GNUTranslations parses optional meta-data out of the translation catalog. It is convention with GNU gettext to include meta-data as the translation for the empty string. This meta-data is in RFC 822-style key: value pairs, and should contain the Project-Id-Version key. If the key Content-Type is found, then the charset property is used to initialize the “protected” _charset instance variable, defaulting to None if not
found. If the charset encoding is specified, then all message ids and message strings read from the catalog are converted to Unicode using this encoding, else ASCII encoding is assumed.

Since message ids are read as Unicode strings too, all *gettext() methods will assume message ids as Unicode strings, not byte strings.

The entire set of key/value pairs are placed into a dictionary and set as the “protected” _info instance variable.

If the .mo file’s magic number is invalid, or if other problems occur while reading the file, instantiating a GNUTranslations class can raise OSError.

The following methods are overridden from the base class implementation:

**GNUTranslations.gettext**(message)

Look up the message id in the catalog and return the corresponding message string, as a Unicode string. If there is no entry in the catalog for the message id, and a fallback has been set, the look up is forwarded to the fallback’s gettext() method. Otherwise, the message id is returned.

**GNUTranslations.lgettext**(message)

Equivalent to gettext(), but the translation is returned as a bytestring encoded in the selected output charset, or in the preferred system encoding if no encoding was explicitly set with set_output_charset().

**GNUTranslations.ngettext**(singular, plural, n)

Do a plural-forms lookup of a message id. singular is used as the message id for purposes of lookup in the catalog, while n is used to determine which plural form to use. The returned message string is a Unicode string.

If the message id is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback’s ngettext() method. Otherwise, when $n$ is 1 singular is returned, and plural is returned in all other cases.

Here is an example:

```python
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ngettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

**GNUTranslations.lngettext**(singular, plural, n)

Equivalent to gettext(), but the translation is returned as a bytestring encoded in the selected output charset, or in the preferred system encoding if no encoding was explicitly set with set_output_charset().

### Solaris message catalog support

The Solaris operating system defines its own binary .mo file format, but since no documentation can be found on this format, it is not supported at this time.

### The Catalog constructor

GNOME uses a version of the gettext module by James Henstridge, but this version has a slightly different API. Its documented usage was:

```python
import gettext
cat = gettext.Catalog(domain, localedir)
_ = cat.gettext
print(_('hello world'))
```
For compatibility with this older module, the function `Catalog()` is an alias for the `translation()` function described above.

One difference between this module and Henstridge’s: his catalog objects supported access through a mapping API, but this appears to be unused and so is not currently supported.

### 23.1.3 Internationalizing your programs and modules

Internationalization (I18N) refers to the operation by which a program is made aware of multiple languages. Localization (L10N) refers to the adaptation of your program, once internationalized, to the local language and cultural habits. In order to provide multilingual messages for your Python programs, you need to take the following steps:

1. prepare your program or module by specially marking translatable strings
2. run a suite of tools over your marked files to generate raw messages catalogs
3. create language specific translations of the message catalogs
4. use the `gettext` module so that message strings are properly translated

In order to prepare your code for I18N, you need to look at all the strings in your files. Any string that needs to be translated should be marked by wrapping it in `_('...')` — that is, a call to the function `_()`. For example:

```python
catalog = Catalog
message = _('writing a log message')
fp = open(filename, 'w')
fp.write(message)
fp.close()
```

In this example, the string `‘writing a log message’` is marked as a candidate for translation, while the strings `‘mylog.txt’` and `‘w’` are not.

The Python distribution comes with two tools which help you generate the message catalogs once you’ve prepared your source code. These may or may not be available from a binary distribution, but they can be found in a source distribution, in the `Tools/i18n` directory.

The `pygettext`³ program scans all your Python source code looking for the strings you previously marked as translatable. It is similar to the GNU `gettext` program except that it understands all the intricacies of Python source code, but knows nothing about C or C++ source code. You don’t need GNU `gettext` unless you’re also going to be translating C code (such as C extension modules).

`pygettext` generates textual Uniforum-style human readable message catalog `.pot` files, essentially structured human readable files which contain every marked string in the source code, along with a placeholder for the translation strings. `pygettext` is a command line script that supports a similar command line interface as `xgettext`; for details on its use, run:

```bash
pygettext --help
```

Copies of these `.pot` files are then handed over to the individual human translators who write language-specific versions for every supported natural language. They send you back the filled in language-specific versions as a `.po` file. Using the `msgfmt.py`⁴ program (in the `Tools/i18n` directory), you take the `.po` files from your translators and generate the machine-readable `.mo` binary catalog files. The `.mo` files are what the `gettext` module uses for the actual translation processing during run-time.

How you use the `gettext` module in your code depends on whether you are internationalizing a single module or your entire application. The next two sections will discuss each case.

---

³ François Pinard has written a program called `xpot` which does a similar job. It is available as part of his `po-utils` package.

⁴ `msgfmt.py` is binary compatible with GNU `msgfmt` except that it provides a simpler, all-Python implementation. With this and `pygettext.py`, you generally won’t need to install the GNU `gettext` package to internationalize your Python applications.
Localizing your module

If you are localizing your module, you must take care not to make global changes, e.g. to the built-in namespace. You should not use the GNU gettext API but instead the class-based API.

Let’s say your module is called “spam” and the module’s various natural language translation .mo files reside in /usr/share/locale in GNU gettext format. Here’s what you would put at the top of your module:

```python
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.lgettext
```

Localizing your application

If you are localizing your application, you can install the `_()` function globally into the built-in namespace, usually in the main driver file of your application. This will let all your application-specific files just use `_('...')` without having to explicitly install it in each file.

In the simple case then, you need only add the following bit of code to the main driver file of your application:

```python
import gettext
ggettext.install('myapplication')
```

If you need to set the locale directory, you can pass these into the `install()` function:

```python
import gettext
gettext.install('myapplication', '/usr/share/locale')
```

Changing languages on the fly

If your program needs to support many languages at the same time, you may want to create multiple translation instances and then switch between them explicitly, like so:

```python
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

Deferred translations

In most coding situations, strings are translated where they are coded. Occasionally however, you need to mark strings for translation, but defer actual translation until later. A classic example is:

```python
animals = ['mollusk',
          'albatross',
          'rat',
          'penguin',
          'python',]
```

```python
# ...
```
for a in animals:
    print(a)

Here, you want to mark the strings in the animals list as being translatable, but you don’t actually want to translate them until they are printed.

Here is one way you can handle this situation:

def _(message): return message

animals = [_('mollusk'),
           _('albatross'),
           _('rat'),
           _('penguin'),
           _('python'), ]

del _

# ...
for a in animals:
    print(_(a))

This works because the dummy definition of _() simply returns the string unchanged. And this dummy definition will temporarily override any definition of _() in the built-in namespace (until the del command). Take care, though if you have a previous definition of _() in the local namespace.

Note that the second use of _() will not identify “a” as being translatable to the pygettext program, since it is not a string.

Another way to handle this is with the following example:

def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print(N_(a))

In this case, you are marking translatable strings with the function N_(),\(^5\) which won’t conflict with any definition of _(). However, you will need to teach your message extraction program to look for translatable strings marked with N_. pygettext and xpot both support this through the use of command line switches.

### 23.1.4 Acknowledgements

The following people contributed code, feedback, design suggestions, previous implementations, and valuable experience to the creation of this module:

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard

\(^5\) The choice of N_() here is totally arbitrary; it could have just as easily been MarkThisStringForTranslation().
23.2 locale — Internationalization services

The `locale` module opens access to the POSIX locale database and functionality. The POSIX locale mechanism allows programmers to deal with certain cultural issues in an application, without requiring the programmer to know all the specifics of each country where the software is executed.

The `locale` module is implemented on top of the `_locale` module, which in turn uses an ANSI C locale implementation if available.

The `locale` module defines the following exception and functions:

**exception** `locale.Error`

Exception raised when the locale passed to `setlocale()` is not recognized.

`locale.setlocale(category, locale=None)`

If `locale` is given and not `None`, `setlocale()` modifies the locale setting for the `category`. The available categories are listed in the data description below. `locale` may be a string, or an iterable of two strings (language code and encoding). If it’s an iterable, it’s converted to a locale name using the locale aliasing engine. An empty string specifies the user’s default settings. If the modification of the locale fails, the exception `Error` is raised. If successful, the new locale setting is returned.

If `locale` is omitted or `None`, the current setting for `category` is returned.

`setlocale()` is not thread-safe on most systems. Applications typically start with a call of

```python
import locale
locale.setlocale(locale.LC_ALL, '')
```

This sets the locale for all categories to the user’s default setting (typically specified in the `LANG` environment variable). If the locale is not changed thereafter, using multithreading should not cause problems.

`locale.localeconv()`

Returns the database of the local conventions as a dictionary. This dictionary has the following strings as keys:
## LC_NUMERIC

- `'decimal_point'` (Key): Decimal point character.
- `'grouping'` (Key): Sequence of numbers specifying which relative positions the `'thousands_sep'` is expected. If the sequence is terminated with `CHAR_MAX`, no further grouping is performed. If the sequence terminates with a 0, the last group size is repeatedly used.

## LC_MONETARY

- `'thousands_sep'` (Key): Character used between groups.
- `'int_curr_symbol'` (Key): International currency symbol.
- `'currency_symbol'` (Key): Local currency symbol.
- `'p_cs_precedes/n_cs_precedes'` (Key): Whether the currency symbol precedes the value (for positive resp. negative values).
- `'p_sep_by_space/n_sep_by_space'` (Key): Whether the currency symbol is separated from the value by a space (for positive resp. negative values).
- `'mon_decimal_point'` (Key): Decimal point used for monetary values.
- `'frac_digits'` (Key): Number of fractional digits used in local formatting of monetary values.
- `'int_frac_digits'` (Key): Number of fractional digits used in international formatting of monetary values.
- `'mon_thousands_sep'` (Key): Group separator used for monetary values.
- `'mon_grouping'` (Key): Equivalent to `'grouping'`, used for monetary values.
- `'positive_sign'` (Key): Symbol used to annotate a positive monetary value.
- `'negative_sign'` (Key): Symbol used to annotate a negative monetary value.
- `'p_sign_posn/n_sign_posn'` (Key): The position of the sign (for positive resp. negative values), see below.

All numeric values can be set to `CHAR_MAX` to indicate that there is no value specified in this locale.

The possible values for `'p_sign_posn'` and `'n_sign_posn'` are given below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Currency and value are surrounded by parentheses.</td>
</tr>
<tr>
<td>1</td>
<td>The sign should precede the value and currency symbol.</td>
</tr>
<tr>
<td>2</td>
<td>The sign should follow the value and currency symbol.</td>
</tr>
<tr>
<td>3</td>
<td>The sign should immediately precede the value.</td>
</tr>
<tr>
<td>4</td>
<td>The sign should immediately follow the value.</td>
</tr>
<tr>
<td><code>CHAR_MAX</code></td>
<td>Nothing is specified in this locale.</td>
</tr>
</tbody>
</table>

### locale.nl_langinfo(option)

Return some locale-specific information as a string. This function is not available on all systems, and the set of possible options might also vary across platforms. The possible argument values are numbers, for which symbolic constants are available in the locale module.

The `nl_langinfo()` function accepts one of the following keys. Most descriptions are taken from the corresponding description in the GNU C library.

---

992 Chapter 23. Internationalization
locale.CODESET
Get a string with the name of the character encoding used in the selected locale.

locale.D_T_FMT
Get a string that can be used as a format string for time.strftime() to represent date and time in a locale-specific way.

locale.D_FMT
Get a string that can be used as a format string for time.strftime() to represent a date in a locale-specific way.

locale.T_FMT
Get a string that can be used as a format string for time.strftime() to represent a time in a locale-specific way.

locale.T_FMT_AMPM
Get a format string for time.strftime() to represent time in the am/pm format.

DAY_1 ... DAY_7
Get the name of the n-th day of the week.

Note: This follows the US convention of DAY_1 being Sunday, not the international convention (ISO 8601) that Monday is the first day of the week.

ABDAY_1 ... ABDAY_7
Get the abbreviated name of the n-th day of the week.

MON_1 ... MON_12
Get the name of the n-th month.

ABMON_1 ... ABMON_12
Get the abbreviated name of the n-th month.

locale.RADIXCHAR
Get the radix character (decimal dot, decimal comma, etc.)

locale.THOUSEP
Get the separator character for thousands (groups of three digits).

locale.YESEXPR
Get a regular expression that can be used with the regex function to recognize a positive response to a yes/no question.

Note: The expression is in the syntax suitable for the regex() function from the C library, which might differ from the syntax used in re.

locale.NOEXPR
Get a regular expression that can be used with the regex(3) function to recognize a negative response to a yes/no question.

locale.CRNCYSTR
Get the currency symbol, preceded by “_” if the symbol should appear before the value, “+” if the symbol should appear after the value, or “.” if the symbol should replace the radix character.

locale.ERA
Get a string that represents the era used in the current locale.

Most locales do not define this value. An example of a locale which does define this value is the Japanese one. In Japan, the traditional representation of dates includes the name of the era corresponding to the then-emperor’s reign.

Normally it should not be necessary to use this value directly. Specifying the E modifier in their format strings causes the time.strftime() function to use this information. The format of the returned string is not specified, and therefore you should not assume knowledge of it on different systems.
locale.ERA_D_T_FMT
Get a format string for time.strftime() to represent date and time in a locale-specific era-based way.

locale.ERA_D_FMT
Get a format string for time.strftime() to represent a date in a locale-specific era-based way.

locale.ERA_T_FMT
Get a format string for time.strftime() to represent a time in a locale-specific era-based way.

locale.ALT_DIGITS
Get a representation of up to 100 values used to represent the values 0 to 99.

custom_function() # This could be a user-defined function

locale.getdefaultlocale([envvars])
Tries to determine the default locale settings and returns them as a tuple of the form (language code, encoding).

According to POSIX, a program which has not called setlocale(LC_ALL, "") runs using the portable 'C' locale. Calling setlocale(LC_ALL, "") lets it use the default locale as defined by the LANG variable. Since we do not want to interfere with the current locale setting we thus emulate the behavior in the way described above.

To maintain compatibility with other platforms, not only the LANG variable is tested, but a list of variables given as envvars parameter. The first found to be defined will be used. envvars defaults to the search path used in GNU gettext; it must always contain the variable name 'LANG'. The GNU gettext search path contains 'LC_ALL', 'LC_CTYPE', 'LANG' and 'LANGUAGE', in that order.

Except for the code 'C', the language code corresponds to RFC 1766. language code and encoding may be None if their values cannot be determined.

locale.getlocale(category=LC_CTYPE)
Returns the current setting for the given locale category as sequence containing language code, encoding.

category may be one of the LC_* values except LC_ALL. It defaults to LC_CTYPE.

Except for the code 'C', the language code corresponds to RFC 1766. language code and encoding may be None if their values cannot be determined.

locale.getpreferredencoding(do_setlocale=True)
Return the encoding used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess.

On some systems, it is necessary to invoke setlocale() to obtain the user preferences, so this function is not thread-safe. If invoking setlocale is not necessary or desired, do_setlocale should be set to False.

locale.normalize(localename)
Returns a normalized locale code for the given locale name. The returned locale code is formatted for use with setlocale(). If normalization fails, the original name is returned unchanged.

If the given encoding is not known, the function defaults to the default encoding for the locale code just like setlocale().

locale.resetlocale(category=LC_ALL)
Sets the locale for category to the default setting.

The default setting is determined by calling getdefaultlocale(). category defaults to LC_ALL.

locale.strcoll(string1, string2)
Compares two strings according to the current LC_COLLATE setting. As any other compare function, returns a negative, or a positive value, or 0, depending on whether string1 collates before or after string2 or is equal to it.

locale.strxfrm(string)
Transforms a string to one that can be used in locale-aware comparisons. For example, strxfrm(s1) < strxfrm(s2) is equivalent to strcoll(s1, s2) < 0. This function can be used when the same string is compared repeatedly, e.g. when collating a sequence of strings.
locale.format (format, val, grouping=False, monetary=False)
   Formats a number val according to the current LC_NUMERIC setting. The format follows the conventions of the % operator. For floating point values, the decimal point is modified if appropriate. If grouping is true, also takes the grouping into account.

   If monetary is true, the conversion uses monetary thousands separator and grouping strings.

   Please note that this function will only work for exactly one %char specifier. For whole format strings, use format_string().

locale.format_string (format, val, grouping=False)
   Processes formatting specifiers as in format % val, but takes the current locale settings into account.

locale.currency (val, symbol=True, grouping=False, international=False)
   Formats a number val according to the current LC_MONETARY settings.

   The returned string includes the currency symbol if symbol is true, which is the default. If grouping is true (which is not the default), grouping is done with the value. If international is true (which is not the default), the international currency symbol is used.

   Note that this function will not work with the ‘C’ locale, so you have to set a locale via setlocale() first.

locale.str (float)
   Formats a floating point number using the same format as the built-in function str (float), but takes the decimal point into account.

locale.atof (string)
   Converts a string to a floating point number, following the LC_NUMERIC settings.

locale.atoi (string)
   Converts a string to an integer, following the LC_NUMERIC conventions.

locale.LC_CTYPE
   Locale category for the character type functions. Depending on the settings of this category, the functions of module string dealing with case change their behaviour.

locale.LC_COLLATE
   Locale category for sorting strings. The functions strcoll() and strxfrm() of the locale module are affected.

locale.LC_TIME
   Locale category for the formatting of time. The function time.strftime() follows these conventions.

locale.LC_MONETARY
   Locale category for formatting of monetary values. The available options are available from the localeconv() function.

locale.LC_MESSAGES
   Locale category for message display. Python currently does not support application specific locale-aware messages. Messages displayed by the operating system, like those returned by os.strerror() might be affected by this category.

locale.LC_NUMERIC
   Locale category for formatting numbers. The functions format(), atoi(), atof() and str() of the locale module are affected by that category. All other numeric formatting operations are not affected.

locale.LC_ALL
   Combination of all locale settings. If this flag is used when the locale is changed, setting the locale for all categories is attempted. If that fails for any category, no category is changed at all. When the locale is retrieved using this flag, a string indicating the setting for all categories is returned. This string can be later used to restore the settings.

locale.CHAR_MAX
   This is a symbolic constant used for different values returned by localeconv().

Example:
>>> import locale
>>> loc = locale.getlocale() # get current locale
# use German locale; name might vary with platform
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
>>> # use German locale
>>> locale.setlocale(locale.LC_ALL, '') # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C') # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc) # restore saved locale

23.2.1 Background, details, hints, tips and caveats

The C standard defines the locale as a program-wide property that may be relatively expensive to change. On top of that, some implementation are broken in such a way that frequent locale changes may cause core dumps. This makes the locale somewhat painful to use correctly.

Initially, when a program is started, the locale is the C locale, no matter what the user’s preferred locale is. There is one exception: the LC_CTYPE category is changed at startup to set the current locale encoding to the user’s preferred locale encoding. The program must explicitly say that it wants the user’s preferred locale settings for other categories by calling setlocale(LC_ALL, ").

It is generally a bad idea to call setlocale() in some library routine, since as a side effect it affects the entire program. Saving and restoring it is also as bad: it is expensive and affects other threads that happen to run before the settings have been restored.

If, when coding a module for general use, you need a locale independent version of an operation that is affected by the locale (such as certain formats used with time.strftime()), you will have to find a way to do it without using the standard library routine. Even better is convincing yourself that using locale settings is okay. Only as a last resort should you document that your module is not compatible with non-C locale settings.

The only way to perform numeric operations according to the locale is to use the special functions defined by this module: atof(), atoi(), format(), str().

There is no way to perform case conversions and character classifications according to the locale. For (Unicode) text strings these are done according to the character value only, while for byte strings, the conversions and classifications are done according to the ASCII value of the byte, and bytes whose high bit is set (i.e., non-ASCII bytes) are never converted or considered part of a character class such as letter or whitespace.

23.2.2 For extension writers and programs that embed Python

Extension modules should never call setlocale(), except to find out what the current locale is. But since the return value can only be used portably to restore it, that is not very useful (except perhaps to find out whether or not the locale is C).

When Python code uses the locale module to change the locale, this also affects the embedding application. If the embedding application doesn’t want this to happen, it should remove the _locale extension module (which does all the work) from the table of built-in modules in the config.c file, and make sure that the _locale module is not accessible as a shared library.

23.2.3 Access to message catalogs

The locale module exposes the C library’s gettext interface on systems that provide this interface. It consists of the functions gettext(), dgettext(), dcgettext(), textdomain(), bindtextdomain(), and bind_textdomain_codeset(). These are similar to the same functions in the gettext module, but use the C library’s binary format for message catalogs, and the C library’s search algorithms for locating message catalogs.

Python applications should normally find no need to invoke these functions, and should use gettext instead. A known exception to this rule are applications that link with additional C libraries which internally invoke...
gettext() or dcgettext(). For these applications, it may be necessary to bind the text domain, so that the libraries can properly locate their message catalogs.
The modules described in this chapter are frameworks that will largely dictate the structure of your program. Currently the modules described here are all oriented toward writing command-line interfaces.

The full list of modules described in this chapter is:

24.1 turtle — Turtle graphics

24.1.1 Introduction

Turtle graphics is a popular way for introducing programming to kids. It was part of the original Logo programming language developed by Wally Feurzig and Seymour Papert in 1966.

Imagine a robotic turtle starting at (0, 0) in the x-y plane. After an import turtle, give it the command turtle.forward(15), and it moves (on-screen!) 15 pixels in the direction it is facing, drawing a line as it moves. Give it the command turtle.right(25), and it rotates in-place 25 degrees clockwise.
Turtle star
Turtle can draw intricate shapes using programs that repeat simple moves.

```python
from turtle import *
color('red', 'yellow')
begn_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

By combining together these and similar commands, intricate shapes and pictures can easily be drawn.

The turtle module is an extended reimplementation of the same-named module from the Python standard distribution up to version Python 2.5.

It tries to keep the merits of the old turtle module and to be (nearly) 100% compatible with it. This means in the first place to enable the learning programmer to use all the commands, classes and methods interactively when using the module from within IDLE run with the -n switch.

The turtle module provides turtle graphics primitives, in both object-oriented and procedure-oriented ways. Because it uses tkinter for the underlying graphics, it needs a version of Python installed with Tk support.

The object-oriented interface uses essentially two+two classes:

1. The TurtleScreen class defines graphics windows as a playground for the drawing turtles. Its constructor needs a tkinter.Canvas or a ScrolledCanvas as argument. It should be used when turtle is used as part of some application.

   The function Screen() returns a singleton object of a TurtleScreen subclass. This function should be used when turtle is used as a standalone tool for doing graphics. As a singleton object, inheriting from its class is not possible.

   All methods of TurtleScreen/Screen also exist as functions, i.e. as part of the procedure-oriented interface.

2. RawTurtle (alias: RawPen) defines Turtle objects which draw on a TurtleScreen. Its constructor needs a Canvas, ScrolledCanvas or TurtleScreen as argument, so the RawTurtle objects know where to draw.

   Derived from RawTurtle is the subclass Turtle (alias: Pen), which draws on “the” Screen instance which is automatically created, if not already present.
All methods of RawTurtle/Turtle also exist as functions, i.e. part of the procedure-oriented interface.

The procedural interface provides functions which are derived from the methods of the classes Screen and Turtle. They have the same names as the corresponding methods. A screen object is automatically created whenever a function derived from a Screen method is called. An (unnamed) turtle object is automatically created whenever any of the functions derived from a Turtle method is called.

To use multiple turtles on a screen one has to use the object-oriented interface.

---

**Note:** In the following documentation the argument list for functions is given. Methods, of course, have the additional first argument *self* which is omitted here.

### 24.1.2 Overview of available Turtle and Screen methods

#### Turtle methods

**Turtle motion**

**Move and draw**

```python
forward() | fd()
backward() | bk() | back()
right()   | rt()
left()    | lt()
goto()    | setpos() | setposition()
setx()
sety()
setheading() | seth()
home()
circle()
dot()
stamp()
clearstamp()
clearstamps()
undo()
speed()
```

**Tell Turtle’s state**

```python
position() | pos()
towards()
xcor()
ycor()
heading()
distance()
```

**Setting and measurement**

```python
degrees()
radians()
```

**Pen control**

**Drawing state**

```python
pendown() | pd() | down()
penup()   | pu() | up()
pensize() | width()
```
pen()
isdown()

**Color control**

color()
pencolor()
fillcolor()

**Filling**

filling()
begin_fill()
end_fill()

**More drawing control**

reset()
clear()
write()

**Turtle state**

**Visibility**

showturtle() | st()
hideturtle() | ht()
isvisible()

**Appearance**

shape()
resizemode()
shapesize() | turtlesize()
shearfactor()
settiltangle()
tiltangle()
tilt()
shapetransform()
get_shapepoly()

**Using events**

onclick()
onrelease()
ondrag()

**Special Turtle methods**

begin_poly()
end_poly()
get_poly()
close()
getturtle() | getpen()
getscreenshot()
setundobuffer()
undobufferentries()

**Methods of TurtleScreen/Screen**

**Window control**
bgcolor()
bgpic()
clear() | clearscren() 
reset() | resetscreen() 
screensize() 
setworldcoordinates()

Animation control

delay()
tracer()
update()

Using screen events

listen()
onkey() | onkeyrelease()
onkeypress()
onclick() | onscreenonclick()
ontimer()
mainloop() | done()

Settings and special methods

mode()
colormode()
getcanvas()
getshapes()
register_shape() | addshape()
turtles()
window_height()
window_width()

Input methods

textinput()
numinput()

Methods specific to Screen

bye()
exitonclick()
setup()
title()

24.1.3 Methods of RawTurtle/Turtle and corresponding functions

Most of the examples in this section refer to a Turtle instance called turtle.

Turtle motion

turtle.forward(distance)
turtle.fd(distance)

Parameters distance – a number (integer or float)

Move the turtle forward by the specified distance, in the direction the turtle is headed.
turtle.position()
(0.00,0.00)
turtle.forward(25)
turtle.position()
(25.00,0.00)
turtle.forward(-75)
turtle.position()
(-50.00,0.00)

turtle.back(distance)
turtle.bk(distance)
turtle.backward(distance)

Parameters distance – a number
Move the turtle backward by distance, opposite to the direction the turtle is headed. Do not change the turtle’s heading.

turtle.position()
(0.00,0.00)
turtle.backward(30)
turtle.position()
(-30.00,0.00)

turtle.right(angle)
turtle.rt(angle)

Parameters angle – a number (integer or float)
Turn turtle right by angle units. (Units are by default degrees, but can be set via the degrees() and radians() functions.) Angle orientation depends on the turtle mode, see mode().

turtle.heading()
22.0
turtle.right(45)
turtle.heading()
337.0

turtle.left(angle)
turtle.lt(angle)

Parameters angle – a number (integer or float)
Turn turtle left by angle units. (Units are by default degrees, but can be set via the degrees() and radians() functions.) Angle orientation depends on the turtle mode, see mode().

turtle.heading()
22.0
turtle.left(45)
turtle.heading()
67.0

turtle.goto(x, y=None)
turtle.setpos(x, y=None)
turtle.setposition(x, y=None)

Parameters
- x – a number or a pair/vector of numbers
- y – a number or None
If `y` is `None`, `x` must be a pair of coordinates or a `Vec2D` (e.g. as returned by `pos()`).

Move turtle to an absolute position. If the pen is down, draw line. Do not change the turtle’s orientation.

```python
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.setpos(60,30)
>>> turtle.pos()
(60.00,30.00)
>>> turtle.setpos((20,80))
>>> turtle.pos()
(20.00,80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00,0.00)
```

`turtle.setx(x)`

**Parameters** `x` – a number (integer or float)

Set the turtle’s first coordinate to `x`, leave second coordinate unchanged.

```python
>>> turtle.position()
(0.00,240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00,240.00)
```

`turtle.sety(y)`

**Parameters** `y` – a number (integer or float)

Set the turtle’s second coordinate to `y`, leave first coordinate unchanged.

```python
>>> turtle.position()
(0.00,40.00)
>>> turtle.sety(-10)
>>> turtle.position()
(0.00,-10.00)
```

`turtle.setheading(to_angle)`
`turtle.seth(to_angle)`

**Parameters** `to_angle` – a number (integer or float)

Set the orientation of the turtle to `to_angle`. Here are some common directions in degrees:

<table>
<thead>
<tr>
<th>standard mode</th>
<th>logo mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - east</td>
<td>0 - north</td>
</tr>
<tr>
<td>90 - north</td>
<td>90 - east</td>
</tr>
<tr>
<td>180 - west</td>
<td>180 - south</td>
</tr>
<tr>
<td>270 - south</td>
<td>270 - west</td>
</tr>
</tbody>
</table>

```python
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

`turtle.home()`

Move turtle to the origin – coordinates (0,0) – and set its heading to its start-orientation (which depends on the mode, see `mode()`).

24.1. **turtle** — Turtle graphics
turtle.
circle

Parameters

• radius – a number
• extent – a number (or None)
• steps – an integer (or None)

Draw a circle with given radius. The center is radius units left of the turtle; extent – an angle – determines which part of the circle is drawn. If extent is not given, draw the entire circle. If extent is not a full circle, one endpoint of the arc is the current pen position. Draw the arc in counterclockwise direction if radius is positive, otherwise in clockwise direction. Finally the direction of the turtle is changed by the amount of extent.

As the circle is approximated by an inscribed regular polygon, steps determines the number of steps to use. If not given, it will be calculated automatically. May be used to draw regular polygons.

turtle.
dot

Parameters

• size – an integer >= 1 (if given)
• color – a colorstring or a numeric color tuple

Draw a circular dot with diameter size, using color. If size is not given, the maximum of pensize+4 and 2*pensize is used.
turtle.\texttt{stamp}()

   Stamp a copy of the turtle shape onto the canvas at the current turtle position. Return a stamp_id for that stamp, which can be used to delete it by calling \texttt{clearstamp(stamp_id)}.

   >>> turtle.color("blue")
   >>> turtle.stamp()
   11
   >>> turtle.fd(50)

turtle.\texttt{clearstamp}(\texttt{stampid})

   Parameters \texttt{stampid} – an integer, must be return value of previous \texttt{stamp()} call

   Delete stamp with given \texttt{stampid}.

   >>> turtle.position()
   (150.00,-0.00)
   >>> turtle.color("blue")
   >>> astamp = turtle.stamp()
   >>> turtle.fd(50)
   >>> turtle.position()
   (200.00,-0.00)
   >>> turtle.clearstamp(astamp)
   >>> turtle.position()
   (200.00,-0.00)

turtle.\texttt{clearstamps}(\texttt{n=None})

   Parameters \texttt{n} – an integer (or \texttt{None})

   Delete all or first/last \texttt{n} of turtle’s stamps. If \texttt{n} is None, delete all stamps, if \texttt{n > 0} delete first \texttt{n} stamps, else if \texttt{n < 0} delete last \texttt{n} stamps.

   >>> for i in range(8):
   ...    turtle.stamp(); turtle.fd(30)
   13
   14
   15
   16
   17
   18
   19
   20
   >>> turtle.clearstamps(2)
   >>> turtle.clearstamps(-2)
   >>> turtle.clearstamps()

turtle.\texttt{undo}()

   Undo (repeatedly) the last turtle action(s). Number of available undo actions is determined by the size of the undobuffer.

   >>> for i in range(4):
   ...    turtle.fd(50); turtle.lt(80)
   ...    turtle.undo()

   turtle.\texttt{speed}(\texttt{speed=None})

   Parameters \texttt{speed} – an integer in the range 0..10 or a speedstring (see below)
Set the turtle’s speed to an integer value in the range 0..10. If no argument is given, return current speed. If input is a number greater than 10 or smaller than 0.5, speed is set to 0. Speedstrings are mapped to speedvalues as follows:

- “fastest”: 0
- “fast”: 10
- “normal”: 6
- “slow”: 3
- “slowest”: 1

Speeds from 1 to 10 enforce increasingly faster animation of line drawing and turtle turning.

Attention: speed = 0 means that no animation takes place. forward/back makes turtle jump and likewise left/right make the turtle turn instantly.

```python
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
```

**Tell Turtle’s state**

- `turtle.position()`
- `turtle.pos()`
  - Return the turtle’s current location (x,y) (as a `Vec2D` vector).
  ```python
  >>> turtle.pos()
  (440.00,-0.00)
  ```
- `turtle.towards(x,y=None)`
  - **Parameters**
    - x – a number or a pair/vector of numbers or a turtle instance
    - y – a number if x is a number, else `None`
  - Return the angle between the line from turtle position to position specified by (x,y), the vector or the other turtle. This depends on the turtle’s start orientation which depends on the mode - “standard”/“world” or “logo”).
  ```python
  >>> turtle.goto(10,10)
  >>> turtle.towards(0,0)
  225.0
  ```
- `turtle.xcor()`
  - Return the turtle’s x coordinate.
  ```python
  >>> turtle.home()
  >>> turtle.left(50)
  >>> turtle.forward(100)
  >>> turtle.pos()
  (64.28, 76.60)
  ```
>>> print(round(turtle.xcor(), 5))
64.27876

turtle.ycor()
Return the turtle’s y coordinate.

>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print(turtle.pos())
(50.00, 86.60)
>>> print(round(turtle.ycor(), 5))
86.60254

turtle.heading()
Return the turtle’s current heading (value depends on the turtle mode, see mode()).

>>> turtle.home()
>>> turtle.left(67)
67.0

turtle.distance(x, y=None)
Parameters
• x – a number or a pair/vector of numbers or a turtle instance
• y – a number if x is a number, else None
Return the distance from the turtle to (x,y), the given vector, or the given other turtle, in turtle step units.

>>> turtle.home()
>>> turtle.distance(30, 40)
50.0
>>> turtle.distance((30, 40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0

Settings for measurement

turtle.degrees(fullcircle=360.0)
Parameters fullcircle – a number
Set angle measurement units, i.e. set number of “degrees” for a full circle. Default value is 360 degrees.

>>> turtle.home()
>>> turtle.left(90)
90.0
Change angle measurement unit to grad (also known as gon, grade, or gradian and equals 1/100-th of the right angle.)

>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
```python
>>> turtle.degrees(360)
90.0

turtle.radians()
Set the angle measurement units to radians. Equivalent to degrees(2*math.pi).

>>> turtle.home()
>>> turtle.left(90)
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966
```

**Pen control**

**Drawing state**

turtle.pendown()
turtle.pd()
turtle.down()
Pull the pen down – drawing when moving.
turtle.penup()
turtle.pu()
turtle.up()
Pull the pen up – no drawing when moving.
turtle.pensize(width=None)
turtle.width(width=None)

**Parameters**

- **width** – a positive number
Set the line thickness to width or return it. If resizemode is set to “auto” and turtleshape is a polygon, that polygon is drawn with the same line thickness. If no argument is given, the current pensize is returned.

```python
>>> turtle.pensize()
1
>>> turtle.pensize(10)  # from here on lines of width 10 are drawn
```

turtle.pen(pen=None, **pendict)

**Parameters**

- **pen** – a dictionary with some or all of the below listed keys
- **pendict** – one or more keyword-arguments with the below listed keys as keywords

Return or set the pen’s attributes in a “pen-dictionary” with the following key/value pairs:

- **‘shown’**: True/False
- **‘pendown’**: True/False
- **‘pencolor’**: color-string or color-tuple
- **‘fillcolor’**: color-string or color-tuple
- **‘pensize’**: positive number
- **‘speed’**: number in range 0..10
- **‘resizemode’**: “auto” or “user” or “noresize”
• "stretchfactor": (positive number, positive number)
• "outline": positive number
• "tilt": number

This dictionary can be used as argument for a subsequent call to `pen()` to restore the former pen-state. Moreover one or more of these attributes can be provided as keyword-arguments. This can be used to set several pen attributes in one statement.

```python
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
 ('shearfactor', 0.0), ('shown', True), ('speed', 9),
 ('stretchfactor', (1.0, 1.0)), ('tilt', 0.0)]
```

```python
tenstate=turtle.pen()
```

```python
>>> turtle.color("yellow", "")
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow')]
```

```python
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red')]
```

**turtle.isdown()**

Return True if pen is down, False if it's up.

```python
>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
>>> turtle.isdown()
True
```

### Color control

**turtle.pencolor(*args)**

Return or set the pencolor.

Four input formats are allowed:

- **pencolor()** Return the current pencolor as color specification string or as a tuple (see example). May be used as input to another color/pencolor/fillcolor call.

- **pencolor(colorstring)** Set pencolor to `colorstring`, which is a Tk color specification string, such as "red", "yellow", or "#33cc8c".

- **pencolor((r, g, b))** Set pencolor to the RGB color represented by the tuple of `r`, `g`, and `b`. Each of `r`, `g`, and `b` must be in the range 0..colormode, where colormode is either 1.0 or 255 (see `colormode()`).

- **pencolor(r, g, b)** Set pencolor to the RGB color represented by `r`, `g`, and `b`. Each of `r`, `g`, and `b` must be in the range 0..colormode.

If turtleshape is a polygon, the outline of that polygon is drawn with the newly set pencolor.

```python
>>> colormode()
1.0
>>> turtle.pencolor()
```
"red"
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
"brown"
>>> tup = (0.2, 0.8, 0.55)
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)
>>> colormode(255)
>>> turtle.pencolor()
(51.0, 204.0, 140.0)
>>> turtle.pencolor(’#32c18f’)
>>> turtle.pencolor()
(50.0, 193.0, 143.0)

turtle.fillcolor(*args)

Return or set the fillcolor.

Four input formats are allowed:

fillcolor() Return the current fillcolor as color specification string, possibly in tuple format (see example). May be used as input to another color/pencolor/fillcolor call.

fillcolor(colorstring) Set fillcolor to colorstring, which is a Tk color specification string, such as "red", "yellow", or "#33cc8c".

fillcolor((r, g, b)) Set fillcolor to the RGB color represented by the tuple of r, g, and b. Each of r, g, and b must be in the range 0..colormode, where colormode is either 1.0 or 255 (see colormode()).

fillcolor(r, g, b) Set fillcolor to the RGB color represented by r, g, and b. Each of r, g, and b must be in the range 0..colormode.

If turtleshape is a polygon, the interior of that polygon is drawn with the newly set fillcolor.

>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
’violet’
>>> col = turtle.pencolor()
>>> col
(50.0, 193.0, 143.0)
>>> turtle.fillcolor(col)
>>> turtle.fillcolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor(’#ffffff’)
>>> turtle.fillcolor()
(255.0, 255.0, 255.0)

turtle.color(*args)

Return or set pencolor and fillcolor.

Several input formats are allowed. They use 0 to 3 arguments as follows:

color() Return the current pencolor and the current fillcolor as a pair of color specification strings or tuples as returned by pencolor() and fillcolor().

color(colorstring), color((r,g,b)), color(r,g,b) Inputs as in pencolor(), set both, fillcolor and pencolor, to the given value.

color(colorstring1, colorstring2), color((r1,g1,b1), (r2,g2,b2))
Equivalent to `pencolor(colorstring1)` and `fillcolor(colorstring2)` and analogously if the other input format is used.

If `turtleshape` is a polygon, outline and interior of that polygon is drawn with the newly set colors.

```python
gturtle.color("red", "green")
gturtle.color()
('red', 'green')
gturtle.color("#285078", "#a0c8f0")
gturtle.color()
((40.0, 80.0, 120.0), (160.0, 200.0, 240.0))
```

See also: Screen method `colormode()`.

**Filling**

```python
turtle.filling()
```

Return fillstate (True if filling, False else).

```python
gturtle.begin_fill()  
```n
To be called just before drawing a shape to be filled.

```python
gturtle.end_fill()
```

Fill the shape drawn after the last call to `begin_fill()`.

```python
gturtle.color("black", "red")
gturtle.begin_fill()  
gturtle.circle(80)
gturtle.end_fill()  
```

**More drawing control**

```python
turtle.reset()
```

Delete the turtle’s drawings from the screen, re-center the turtle and set variables to the default values.

```python
gturtle.goto(0,-22)
gturtle.left(100)
gturtle.position()
(0.00,-22.00)
gturtle.heading()
100.0
gturtle.reset()
gturtle.position()
(0.00,0.00)
gturtle.heading()
0.0
```

```python
turtle.clear()
```

Delete the turtle’s drawings from the screen. Do not move turtle. State and position of the turtle as well as drawings of other turtles are not affected.

```python
turtle.write(arg, move=False, align="left", font=("Arial", 8, "normal"))
```
Parameters

- **arg** – object to be written to the TurtleScreen
- **move** – True/False
- **align** – one of the strings “left”, “center” or right”
- **font** – a triple (fontname, fontsize, fonttype)

Write text - the string representation of `arg` - at the current turtle position according to `align` (“left”, “center” or right”) and with the given font. If `move` is True, the pen is moved to the bottom-right corner of the text. By default, `move` is False.

```python
>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)
```

Turtle state

Visibility

```python
turtle.hideturtle()
turtle.ht()
```

Make the turtle invisible. It’s a good idea to do this while you’re in the middle of doing some complex drawing, because hiding the turtle speeds up the drawing observably.

```python
>>> turtle.hideturtle()
```

```python
turtle.showturtle()
turtle.st()
```

Make the turtle visible.

```python
>>> turtle.showturtle()
```

```python
turtle.isvisible()
```

Return True if the Turtle is shown, False if it’s hidden.

```python
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

Appearance

```python
turtle.shape(name=None)
```

Parameters name – a string which is a valid shapename

Set turtle shape to shape with given name or, if name is not given, return name of current shape. Shape with `name` must exist in the TurtleScreen’s shape dictionary. Initially there are the following polygon shapes: “arrow”, “turtle”, “circle”, “square”, “triangle”, “classic”. To learn about how to deal with shapes see Screen method `register_shape()`.

```python
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```
turtle.resizemode(rmode=None)

Parameters rmode – one of the strings “auto”, “user”, “noresize”

Set resizemode to one of the values: “auto”, “user”, “noresize”. If rmode is not given, return current resizemode. Different resizemodes have the following effects:

• “auto”: adapts the appearance of the turtle corresponding to the value of pensize.
• “user”: adapts the appearance of the turtle according to the values of stretchfactor and outlinewidth
  (outline), which are set by shapesize().
• “noresize”: no adaption of the turtle’s appearance takes place.

resizemode(“user”) is called by shapesize() when used with arguments.

```python
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

turtle.shapesize(stretch_wid=None, stretch_len=None, outline=None)
turtle.turtlesize(stretch_wid=None, stretch_len=None, outline=None)

Parameters

• stretch_wid – positive number
• stretch_len – positive number
• outline – positive number

Return or set the pen’s attributes x/y-stretchfactors and/or outline. Set resizemode to “user”. If and only if resizemode is set to “user”, the turtle will be displayed stretched according to its stretchfactors: stretch_wid is stretchfactor perpendicular to its orientation, stretch_len is stretchfactor in direction of its orientation, outline determines the width of the shapes’s outline.

```python
>>> turtle.shapesize()
(1.0, 1.0, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

turtle.shearfactor(shear=None)

Parameters shear – number (optional)

Set or return the current shearfactor. Shear the turtleshape according to the given shearfactor shear, which is the tangent of the shear angle. Do not change the turtle’s heading (direction of movement). If shear is not given: return the current shearfactor, i.e. the tangent of the shear angle, by which lines parallel to the heading of the turtle are sheared.

```python
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.shearfactor(0.5)
>>> turtle.shearfactor()
0.5
```

turtle.tilt(angle)
Parameters **angle** – a number

Rotate the turtleshape by *angle* from its current tilt-angle, but do *not* change the turtle’s heading (direction of movement).

```python
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5, 2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

```
turtle.settiltangle(angle)
```

Parameters **angle** – a number

Rotate the turtleshape to point in the direction specified by *angle*, regardless of its current tilt-angle. *Do not* change the turtle’s heading (direction of movement).

```python
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5, 2)
>>> turtle.settiltangle(45)
>>> turtle.fd(50)
>>> turtle.settiltangle(-45)
>>> turtle.fd(50)
```

**Deprecated since version 3.1.**

```
turtle.tiltangle(angle=None)
```

Parameters **angle** – a number (optional)

Set or return the current tilt-angle. If *angle* is given, rotate the turtleshape to point in the direction specified by *angle*, regardless of its current tilt-angle. *Do not* change the turtle’s heading (direction of movement). If *angle* is not given: return the current tilt-angle, i. e. the angle between the orientation of the turtleshape and the heading of the turtle (its direction of movement).

```python
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5, 2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

```
turtle.shapetransform(t11=None, t12=None, t21=None, t22=None)
```

Parameters

- **t11** – a number (optional)
- **t12** – a number (optional)
- **t21** – a number (optional)
- **t22** – a number (optional)

Set or return the current transformation matrix of the turtle shape.

If none of the matrix elements are given, return the transformation matrix as a tuple of 4 elements. Otherwise set the given elements and transform the turtleshape according to the matrix consisting of first row t11, t12 and second row t21, 22. The determinant $t11 * t22 - t12 * t21$ must not be zero, otherwise an error is raised. Modify stretchfactor, shearfactor and tiltangle according to the given matrix.
```python
>>> turtle = Turtle()
>>> turtle.shape("square")
>>> turtle.shapesize(4,2)
>>> turtle.shearfactor(-0.5)
>>> turtle.shapetransform()
(4.0, -1.0, -0.0, 2.0)

turtle.get_shapepoly()
Return the current shape polygon as tuple of coordinate pairs. This can be used to define a new shape or components of a compound shape.

```  
```python
>>> turtle.shape("square")
>>> turtle.shapetransform(4, -1, 0, 2)
>>> turtle.get_shapepoly()
((50, -20), (30, 20), (-50, 20), (-30, -20))
```  
Using events

turtle.onclick (fun, btn=1, add=None)

Parameters

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **num** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – True or False – if True, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-click events on this turtle. If *fun* is None, existing bindings are removed. Example for the anonymous turtle, i.e. the procedural way:

```python
>>> def turn(x, y):
...    left(180)
...
>>> onclick(turn)  # Now clicking into the turtle will turn it.
>>> onclick(None)  # event-binding will be removed
```  
```python
turtle.onrelease (fun, btn=1, add=None)

Parameters

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **num** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – True or False – if True, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-button-release events on this turtle. If *fun* is None, existing bindings are removed.

```python
>>> class MyTurtle(Turtle):
...    def glow(self, x, y):
...        self.fillcolor("red")
...    def unglow(self, x, y):
...        self.fillcolor("")
...
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)  # clicking on turtle turns fillcolor red,
>>> turtle.onrelease(turtle.unglow)  # releasing turns it to transparent.
```
turtle.ondrag (fun, btn=1, add=None)

Parameters

• fun – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
• num – number of the mouse-button, defaults to 1 (left mouse button)
• add – True or False – if True, a new binding will be added, otherwise it will replace a former binding

Bind fun to mouse-move events on this turtle. If fun is None, existing bindings are removed.

Remark: Every sequence of mouse-move-events on a turtle is preceded by a mouse-click event on that turtle.

>>> turtle.ondrag(turtle.goto)

Subsequently, clicking and dragging the Turtle will move it across the screen thereby producing handdrawings (if pen is down).

Special Turtle methods

turtle.begin_poly ()
Start recording the vertices of a polygon. Current turtle position is first vertex of polygon.

turtle.end_poly ()
Stop recording the vertices of a polygon. Current turtle position is last vertex of polygon. This will be connected with the first vertex.

turtle.get_poly ()
Return the last recorded polygon.

>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
>>> turtle.left(20)
>>> turtle.fd(30)
>>> turtle.left(60)
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)

turtle.clone ()
Create and return a clone of the turtle with same position, heading and turtle properties.

>>> mick = Turtle()
>>> joe = mick.clone()

turtle.getturtle ()
turtle.getpen ()
Return the Turtle object itself. Only reasonable use: as a function to return the “anonymous turtle”:

>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
turtle.getscreen()

Return the TurtleScreen object the turtle is drawing on. TurtleScreen methods can then be called for that object.

```python
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

turtle.setundobuffer(size)

Parameters size – an integer or None

Set or disable undobuffer. If size is an integer an empty undobuffer of given size is installed. size gives the maximum number of turtle actions that can be undone by the undo() method/function. If size is None, the undobuffer is disabled.

```python
>>> turtle.setundobuffer(42)
```

turtle.undobufferentries()

Return number of entries in the undobuffer.

```python
>>> while undobufferentries():
...     undo()
```

Compound shapes

To use compound turtle shapes, which consist of several polygons of different color, you must use the helper class Shape explicitly as described below:

1. Create an empty Shape object of type “compound”.
2. Add as many components to this object as desired, using the addcomponent() method.

For example:

```python
>>> s = Shape("compound")
>>> poly1 = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s.addcomponent(poly1, "red", "blue")
>>> poly2 = ((0,0),(10,-5),(-10,-5))
>>> s.addcomponent(poly2, "blue", "red")
```
3. Now add the Shape to the Screen’s shapelist and use it:

```python
>>> register_shape("myshape", s)
>>> shape("myshape")
```

Note: The Shape class is used internally by the register_shape() method in different ways. The application programmer has to deal with the Shape class only when using compound shapes like shown above!

24.1.4 Methods of TurtleScreen/Screen and corresponding functions

Most of the examples in this section refer to a TurtleScreen instance called screen.

Window control

turtlebgcolor(*args)
Parameters `args` – a color string or three numbers in the range 0..colormode or a 3-tuple of such numbers

Set or return background color of the TurtleScreen.

```python
green = "green"
green = screen.bgcolor()
green = "green"
```

```python
green = screen.bgcolor()
green = green
```

```
>>> screen.bgcolor("orange")
'orange'
>>> screen.bgcolor("#800080")
(128.0, 0.0, 128.0)
```

turtle.bgpic(`picname=None`)

Parameters `picname` – a string, name of a gif-file or "nopic", or None

Set background image or return name of current background image. If `picname` is a filename, set the corresponding image as background. If `picname` is "nopic", delete background image, if present. If `picname` is None, return the filename of the current background image.

```python
green = screen.bgpic()
green = "nopic"
green = screen.bgpic("landscape.gif")
green = screen.bgpic()
```

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
"landscape.gif"
```

turtle.clear()
turtle.clearscreen()

Delete all drawings and all turtles from the TurtleScreen. Reset the now empty TurtleScreen to its initial state: white background, no background image, no event bindings and tracing on.

Note: This TurtleScreen method is available as a global function only under the name `clearscreen`. The global function `clear` is a different one derived from the Turtle method `clear`.

```python
green = screen.clear()
```

```
turtle.reset()
turtle.resetscreen()

Reset all Turtles on the Screen to their initial state.

Note: This TurtleScreen method is available as a global function only under the name `resetscreen`. The global function `reset` is another one derived from the Turtle method `reset`.

```python
green = screen.reset()
```

turtle.screensize(`canvwidth=None, canvheight=None, bg=None`)

Parameters

- `canvwidth` – positive integer, new width of canvas in pixels
- `canvheight` – positive integer, new height of canvas in pixels
- `bg` – colorstring or color-tuple, new background color

If no arguments are given, return current (canvwidth, canvheight). Else resize the canvas the turtles are drawing on. Do not alter the drawing window. To observe hidden parts of the canvas, use the scrollbars. With this method, one can make visible those parts of a drawing which were outside the canvas before.

```python
green = screen.screensize()
```

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000,1500)
>>> screen.screensize()
(2000, 1500)
```
e.g. to search for an erroneously escaped turtle ;-)  

turtle.setworldcoordinates (llx, lly, urx, ury)  

Parameters

- **llx** – a number, x-coordinate of lower left corner of canvas  
- **lly** – a number, y-coordinate of lower left corner of canvas  
- **urx** – a number, x-coordinate of upper right corner of canvas  
- **ury** – a number, y-coordinate of upper right corner of canvas

Set up user-defined coordinate system and switch to mode "world" if necessary. This performs a screen.reset(). If mode “world” is already active, all drawings are redrawn according to the new coordinates.

**ATTENTION:** in user-defined coordinate systems angles may appear distorted.

```python
>>> screen.reset()
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...    left(10)
...  ...
>>> for _ in range(8):
...    left(45); fd(2)  # a regular octagon
```

**Animation control**

turtle.delay (delay=None)  

Parameters **delay** – positive integer

Set or return the drawing delay in milliseconds. (This is approximately the time interval between two consecutive canvas updates.) The longer the drawing delay, the slower the animation.

Optional argument:

```python
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

turtle.tracer (n=None, delay=None)  

Parameters

- **n** – nonnegative integer  
- **delay** – nonnegative integer

Turn turtle animation on/off and set delay for update drawings. If **n** is given, only each n-th regular screen update is really performed. (Can be used to accelerate the drawing of complex graphics.) When called without arguments, returns the currently stored value of **n**. Second argument sets delay value (see delay()).

```python
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...    fd(dist)
...    rt(90)
...    dist += 2
```
turtle.update()

Perform a TurtleScreen update. To be used when tracer is turned off.

See also the RawTurtle/Turtle method speed().

Using screen events

turtle.listen(xdummy=None, ydummy=None)

Set focus on TurtleScreen (in order to collect key-events). Dummy arguments are provided in order to be able to pass listen() to the onclick method.

turtle.onkey(fun, key)
turtle.onkeyrelease(fun, key)

Parameters

- fun – a function with no arguments or None
- key – a string: key (e.g. “a”) or key-symbol (e.g. “space”)

Bind fun to key-release event of key. If fun is None, event bindings are removed. Remark: in order to be able to register key-events, TurtleScreen must have the focus. (See method listen().)

>>> def f():
...   fd(50)
...   lt(60)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()

turtle.onkeypress(fun, key=None)

Parameters

- fun – a function with no arguments or None
- key – a string: key (e.g. “a”) or key-symbol (e.g. “space”)

Bind fun to key-press event of key if key is given, or to any key-press-event if no key is given. Remark: in order to be able to register key-events, TurtleScreen must have focus. (See method listen().)

>>> def f():
...   fd(50)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()

turtle.onclick(fun, btn=1, add=None)
turtle.onscreenclick(fun, btn=1, add=None)

Parameters

- fun – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- num – number of the mouse-button, defaults to 1 (left mouse button)
- add – True or False – if True, a new binding will be added, otherwise it will replace a former binding

Bind fun to mouse-click events on this screen. If fun is None, existing bindings are removed.

Example for a TurtleScreen instance named screen and a Turtle instance named turtle:
>>> screen.onclick(turtle.goto)  # Subsequently clicking into the TurtleScreen will
>>> screen.onclick(None)        # make the turtle move to the clicked point.
>>> screen.onclick(None)        # remove event binding again

Note: This TurtleScreen method is available as a global function only under the name onscreenclick. The global function onclick is another one derived from the Turtle method onclick.

turtle.ontimer(fun, t=0)

Parameters

• fun – a function with no arguments
• t – a number &ge; 0

Install a timer that calls fun after t milliseconds.

>>> running = True
>>> def f():
...    if running:
...        fd(50)
...        lt(60)
...        screen.ontimer(f, 250)
>>> f()  ### makes the turtle march around
>>> running = False

turtle.mainloop()
turtle.done()

Starts event loop - calling Tkinter’s mainloop function. Must be the last statement in a turtle graphics program. Must not be used if a script is run from within IDLE in -n mode (No subprocess) - for interactive use of turtle graphics.

>>> screen.mainloop()

Input methods

turtle.textinput(title, prompt)

Parameters

• title – string
• prompt – string

Pop up a dialog window for input of a string. Parameter title is the title of the dialog window, prompt is a text mostly describing what information to input. Return the string input. If the dialog is canceled, return None.

>>> screen.textinput("NIM", "Name of first player:")

turtle.numinput(title, prompt, default=None, minval=None, maxval=None)

Parameters

• title – string
• prompt – string
• default – number (optional)
• minval – number (optional)
• maxval – number (optional)
Pop up a dialog window for input of a number. title is the title of the dialog window, prompt is a text mostly describing what numerical information to input. default: default value, minval: minimum value for input, maxval: maximum value for input. The number input must be in the range minval .. maxval if these are given. If not, a hint is issued and the dialog remains open for correction. Return the number input. If the dialog is canceled, return None.

```python
>>> screen.numinput("Poker", "Your stakes:", 1000, minval=10, maxval=10000)
```

**Settings and special methods**

turtle.

mode (mode=None)

Parameters mode – one of the strings “standard”, “logo” or “world”

Set turtle mode (“standard”, “logo” or “world”) and perform reset. If mode is not given, current mode is returned.

Mode “standard” is compatible with old turtle. Mode “logo” is compatible with most Logo turtle graphics. Mode “world” uses user-defined “world coordinates”. Attention: in this mode angles appear distorted if x/y unit-ratio doesn’t equal 1.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Initial turtle heading</th>
<th>positive angles</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;standard&quot; &quot;logo&quot;</td>
<td>to the right (east) upward (north)</td>
<td>counterclockwise clockwise</td>
</tr>
</tbody>
</table>

```python
>>> mode("logo")  # resets turtle heading to north
>>> mode()
'logo'
```

turtle.

colormode (cmode=None)

Parameters cmode – one of the values 1.0 or 255

Return the colormode or set it to 1.0 or 255. Subsequently r, g, b values of color triples have to be in the range 0..cmode.

```python
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
...
TurtleGraphicsError: bad color sequence: (240, 160, 80)
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

turtle.

getcanvas

Return the Canvas of this TurtleScreen. Useful for insiders who know what to do with a Tkinter Canvas.

```python
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas object at ...>
```

turtle.

getshapes

Return a list of names of all currently available turtle shapes.

```python
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```
turtle.register_shape(name, shape=None)
turtle.addshape(name, shape=None)

There are three different ways to call this function:

1. **name** is the name of a gif-file and **shape** is **None**: Install the corresponding image shape.

   >>> screen.register_shape("turtle.gif")

   **Note:** Image shapes do not rotate when turning the turtle, so they do not display the heading of the turtle!

2. **name** is an arbitrary string and **shape** is a tuple of pairs of coordinates: Install the corresponding polygon shape.

   >>> screen.register_shape("triangle", ((5,-3), (0,5), (-5,-3)))

3. **name** is an arbitrary string and **shape** is a (compound) **Shape** object: Install the corresponding compound shape.

   Add a turtle shape to TurtleScreen’s shapelist. Only thusly registered shapes can be used by issuing the command `shape(shapename)`.

turtle.turtles()

Return the list of turtles on the screen.

   >>> for turtle in screen.turtles():
   ...    turtle.color("red")

turtle.window_height()

Return the height of the turtle window.

   >>> screen.window_height()
480

turtle.window_width()

Return the width of the turtle window.

   >>> screen.window_width()
640

**Methods specific to Screen, not inherited from TurtleScreen**

turtle.bye()

Shut the turtlegraphics window.

turtle.exitonclick()

Bind bye() method to mouse clicks on the Screen.

   If the value “using_IDLE” in the configuration dictionary is False (default value), also enter mainloop. Remark: If IDLE with the -n switch (no subprocess) is used, this value should be set to True in turtle.cfg. In this case IDLE’s own mainloop is active also for the client script.

turtle.setup(width=_CFG["width"], height=_CFG["height"], startx=_CFG["leftright"], starty=_CFG["topbottom"])

Set the size and position of the main window. Default values of arguments are stored in the configuration dictionary and can be changed via a turtle.cfg file.

**Parameters**
• **width** – if an integer, a size in pixels, if a float, a fraction of the screen; default is 50% of screen

• **height** – if an integer, the height in pixels, if a float, a fraction of the screen; default is 75% of screen

• **startx** – if positive, starting position in pixels from the left edge of the screen, if negative from the right edge, if None, center window horizontally

• **starty** – if positive, starting position in pixels from the top edge of the screen, if negative from the bottom edge, if None, center window vertically

```python
>>> screen.setup (width=200, height=200, startx=0, starty=0)
# sets window to 200x200 pixels, in upper left of screen
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
# sets window to 75% of screen by 50% of screen and centers
```

**turtle.title**(titlestring)

- **Parameters** titlestring – a string that is shown in the titlebar of the turtle graphics window

  Set title of turtle window to **titlestring**.

  ```python
  >>> screen.title("Welcome to the turtle zoo!"
  ```

### 24.1.5 Public classes

**class** `turtle.RawTurtle` *(canvas)*

**class** `turtle.RawPen` *(canvas)*

- **Parameters** canvas – a **tkinter.Canvas**, **ScrolledCanvas** or a **TurtleScreen**

Create a turtle. The turtle has all methods described above as "methods of Turtle/RawTurtle".

**class** `turtle.Turtle`

Subclass of RawTurtle, has the same interface but draws on a default **Screen** object created automatically when needed for the first time.

**class** `turtle.TurtleScreen` *(cv)*

- **Parameters** cv – a **tkinter.Canvas**

  Provides screen oriented methods like **setbg()** etc. that are described above.

**class** `turtle.Screen`

Subclass of TurtleScreen, with **four methods added**.

**class** `turtle.ScrolledCanvas` *(master)*

- **Parameters** master – some Tkinter widget to contain the ScrolledCanvas, i.e. a Tkinter-canvas with scrollbars added

  Used by class Screen, which thus automatically provides a ScrolledCanvas as playground for the turtles.

**class** `turtle.Shape` *(type_, data)*

- **Parameters** type_ – one of the strings “polygon”, “image”, “compound”

  Data structure modeling shapes. The pair (type_, data) must follow this specification:

<table>
<thead>
<tr>
<th>type_</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;polygon&quot;</td>
<td>a polygon-tuple, i.e. a tuple of pairs of coordinates</td>
</tr>
<tr>
<td>&quot;image&quot;</td>
<td>an image (in this form only used internally!)</td>
</tr>
<tr>
<td>&quot;compound&quot;</td>
<td>None (a compound shape has to be constructed using the <strong>addcomponent()</strong> method)</td>
</tr>
</tbody>
</table>

**addcomponent** *(poly, fill, outline=None)*

- **Parameters**
• **poly** – a polygon, i.e. a tuple of pairs of numbers

• **fill** – a color the *poly* will be filled with

• **outline** – a color for the *poly*’s outline (if given)

Example:

```python
>>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s = Shape("compound")
>>> s.add_component(poly, "red", "blue")
>>> # ... add more components and then use register_shape()
```

See [*Compound shapes*](#).

```python
class turtle.Turtle.Vec2D(x, y)
```

A two-dimensional vector class, used as a helper class for implementing turtle graphics. May be useful for turtle graphics programs too. Derived from tuple, so a vector is a tuple!

Provides (for *a*, *b* vectors, *k* number):

• *a* + *b* vector addition

• *a* - *b* vector subtraction

• *a* * b inner product

• *k* * a and a * k multiplication with scalar

• abs(*a*) absolute value of *a*

• *a*.rotate(*angle*) rotation

### 24.1.6 Help and configuration

**How to use help**

The public methods of the Screen and Turtle classes are documented extensively via docstrings. So these can be used as online-help via the Python help facilities:

- When using IDLE, tooltips show the signatures and first lines of the docstrings of typed in function-/method calls.

- Calling `help()` on methods or functions displays the docstrings:

  ```python
  >>> help(Screen.bgcolor)
  Help on method bgcolor in module turtle:
  bgcolor(self, *args) unbound turtle.Screen method
  Set or return backgroundcolor of the TurtleScreen.
  
  Arguments (if given): a color string or three numbers in the range 0..colormode or a 3-tuple of such numbers.
  
  >>> screen.bgcolor("orange")
  >>> screen.bgcolor()
  "orange"
  >>> screen.bgcolor(0.5,0,0.5)
  >>> screen.bgcolor()
  "#800080"
  
  >>> help(Turtle.penup)
  Help on method penup in module turtle:
  ```
penup(self) unbound turtle.Turtle method
    Pull the pen up -- no drawing when moving.
    
    Aliases: penup | pu | up
    
    No argument
    
    >>> turtle.penup()

• The docstrings of the functions which are derived from methods have a modified form:

    >>> help(bgcolor)
    Help on function bgcolor in module turtle:
    
    bgcolor(*args)
    Set or return backgroundcolor of the TurtleScreen.
    
    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.
    
    Example::
    
    >>> bgcolor("orange")
    >>> bgcolor()
    "orange"
    >>> bgcolor(0.5,0,0.5)
    >>> bgcolor()
    "#800080"

    >>> help(penup)
    Help on function penup in module turtle:
    
    penup()
    Pull the pen up -- no drawing when moving.
    
    Aliases: penup | pu | up
    
    No argument
    
    Example:
    
    >>> penup()

These modified docstrings are created automatically together with the function definitions that are derived from
the methods at import time.

Translation of docstrings into different languages

There is a utility to create a dictionary the keys of which are the method names and the values of which are the
docstrings of the public methods of the classes Screen and Turtle.

turtle.write_docstringdict (filename="turtle_docstringdict")

Parameters  filename – a string, used as filename

Create and write docstring-dictionary to a Python script with the given filename. This function has to be
called explicitly (it is not used by the turtle graphics classes). The docstring dictionary will be written to
the Python script filename.py. It is intended to serve as a template for translation of the docstrings into
different languages.
If you (or your students) want to use turtle with online help in your native language, you have to translate the
docstrings and save the resulting file as e.g. turtle_docstringdict_german.py.

If you have an appropriate entry in your turtle.cfg file this dictionary will be read in at import time and will replace
the original English docstrings.

At the time of this writing there are docstring dictionaries in German and in Italian. (Requests please to
blingl@aon.at.)

How to configure Screen and Turtles

The built-in default configuration mimics the appearance and behaviour of the old turtle module in order to retain
best possible compatibility with it.

If you want to use a different configuration which better reflects the features of this module or which better fits to
your needs, e.g. for use in a classroom, you can prepare a configuration file turtle.cfg which will be read at
import time and modify the configuration according to its settings.

The built in configuration would correspond to the following turtle.cfg:

```python
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = nosize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

Short explanation of selected entries:

- The first four lines correspond to the arguments of the Screen.setup() method.
- Line 5 and 6 correspond to the arguments of the method Screen.screensize().
- shape can be any of the built-in shapes, e.g: arrow, turtle, etc. For more info try help(shape).
- If you want to use no fillcolor (i.e. make the turtle transparent), you have to write fillcolor = "" (but
  all nonempty strings must not have quotes in the cfg-file).
- If you want to reflect the turtle its state, you have to use resizemode = auto.
- If you set e.g. language = italian the docstringdict turtle_docstringdict_italian.py
  will be loaded at import time (if present on the import path, e.g. in the same directory as turtle.
- The entries exampleturtle and examplescreen define the names of these objects as they occur in the doc-
  strings. The transformation of method-docstrings to function-docstrings will delete these names from the
docstrings.
- using_IDLE: Set this to True if you regularly work with IDLE and its -n switch (“no subprocess”). This
  will prevent exitonclick() to enter the mainloop.
There can be a `turtle.cfg` file in the directory where `turtle` is stored and an additional one in the current working directory. The latter will override the settings of the first one.

The `Lib/turtledemo` directory contains a `turtle.cfg` file. You can study it as an example and see its effects when running the demos (preferably not from within the demo-viewer).

### 24.1.7 Demo scripts

There is a set of demo scripts in the `turtledemo` package. These scripts can be run and viewed using the supplied demo viewer as follows:

```python
python -m turtledemo
```

Alternatively, you can run the demo scripts individually. For example,

```python
python -m turtledemo.bytedesign
```

The `turtledemo` package directory contains:

- a set of 15 demo scripts demonstrating different features of the new module `turtle`;
- a demo viewer `__main__.py` which can be used to view the source code of the scripts and run them at the same time. 14 of the examples can be accessed via the Examples menu; all of them can also be run standalone.
- The example `turtledemo.two_canvases` demonstrates the simultaneous use of two canvases with the turtle module. Therefore it only can be run standalone.
- There is a `turtle.cfg` file in this directory, which serves as an example for how to write and use such files.

The demo scripts are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytedesign</td>
<td>complex classical turtle graphics pattern</td>
<td><code>tracer()</code>, delay, <code>update()</code> world coordinates</td>
</tr>
<tr>
<td>chaos</td>
<td>graphs Verhulst dynamics, shows that computer's computations can generate results sometimes against the common sense expectations</td>
<td>turtles as clock's hands, <code>ontimer</code></td>
</tr>
<tr>
<td>clock</td>
<td>analog clock showing time of your computer</td>
<td><code>ondrag()</code></td>
</tr>
<tr>
<td>colormixer</td>
<td>experiment with r, g, b</td>
<td><code>recursion</code></td>
</tr>
<tr>
<td>fractalcurves</td>
<td>Hilbert &amp; Koch curves</td>
<td><code>L-System</code></td>
</tr>
<tr>
<td>lindemayer</td>
<td>ethnnomathematics (indian kolams)</td>
<td>Rectangular Turtles as Hanoi discs (shape, shapesize)</td>
</tr>
<tr>
<td>minimal_hanoi</td>
<td>Towers of Hanoi</td>
<td>turtles as nimsticks, event driven (mouse, keyboard)</td>
</tr>
<tr>
<td>nim</td>
<td>play the classical nim game with three heaps of sticks against the computer.</td>
<td><code>onclick()</code> turtle: appearance and animation</td>
</tr>
<tr>
<td>paint</td>
<td>super minimalistic drawing program</td>
<td><code>stamp()</code> compound shapes, <code>Vec2D</code> compound shapes, clone shapesize, tint, <code>get_shapepoly</code>, <code>update</code></td>
</tr>
<tr>
<td>peace</td>
<td>elementary</td>
<td><code>clone()</code></td>
</tr>
<tr>
<td>penrose</td>
<td>aperiodic tiling with kites and darts</td>
<td><code>clone()</code>, <code>undo()</code></td>
</tr>
<tr>
<td>planet_and_moon</td>
<td>simulation of gravitational system</td>
<td><code>circle()</code></td>
</tr>
<tr>
<td>round_dance</td>
<td>dancing turtles rotating pairwise in opposite direction</td>
<td></td>
</tr>
<tr>
<td>tree</td>
<td>a (graphical) breadth first tree (using generators)</td>
<td></td>
</tr>
<tr>
<td>wikipedia</td>
<td>a pattern from the wikipedia article on turtle graphics</td>
<td></td>
</tr>
<tr>
<td>yingyang</td>
<td>another elementary example</td>
<td></td>
</tr>
</tbody>
</table>

Have fun!
24.1.8 Changes since Python 2.6

- The methods `Turtle.tracer()`, `Turtle.window_width()` and `Turtle.window_height()` have been eliminated. Methods with these names and functionality are now available only as methods of `Screen`. The functions derived from these remain available. (In fact already in Python 2.6 these methods were merely duplications of the corresponding `TurtleScreen/Screen`-methods.)

- The method `Turtle.fill()` has been eliminated. The behaviour of `begin_fill()` and `end_fill()` have changed slightly: now every filling-process must be completed with an `end_fill()` call.

- A method `Turtle.filling()` has been added. It returns a boolean value: `True` if a filling process is under way, `False` otherwise. This behaviour corresponds to a `fill()` call without arguments in Python 2.6.

24.1.9 Changes since Python 3.0

- The methods `Turtle.shearfactor()`, `Turtle.shapetransform()` and `Turtle.get_shapepoly()` have been added. Thus the full range of regular linear transforms is now available for transforming turtle shapes. `Turtle.tiltangle()` has been enhanced in functionality: it now can be used to get or set the tiltangle. `Turtle.settiltangle()` has been deprecated.

- The method `Screen.onkeypress()` has been added as a complement to `Screen.onkey()` which in fact binds actions to the keyrelease event. Accordingly the latter has got an alias: `Screen.onkeyrelease()`.

- The method `Screen.mainloop()` has been added. So when working only with `Screen` and `Turtle` objects one must not additionally import `mainloop()` anymore.

- Two input methods has been added `Screen.textinput()` and `Screen.numinput()`. These popup input dialogs and return strings and numbers respectively.

- Two example scripts `tdemo_nim.py` and `tdemo_round_dance.py` have been added to the `Lib/turtledemo` directory.

24.2 cmd — Support for line-oriented command interpreters

Source code: `Lib/cmd.py`

The `Cmd` class provides a simple framework for writing line-oriented command interpreters. These are often useful for test harnesses, administrative tools, and prototypes that will later be wrapped in a more sophisticated interface.

```python
class Cmd (completekey='tab', stdin=None, stdout=None)
```

A `Cmd` instance or subclass instance is a line-oriented interpreter framework. There is no good reason to instantiate `Cmd` itself; rather, it’s useful as a superclass of an interpreter class you define yourself in order to inherit `Cmd`’s methods and encapsulate action methods.

The optional argument `completekey` is the `readline` name of a completion key; it defaults to `Tab`. If `completekey` is not `None` and `readline` is available, command completion is done automatically.

The optional arguments `stdin` and `stdout` specify the input and output file objects that the `Cmd` instance or subclass instance will use for input and output. If not specified, they will default to `sys.stdin` and `sys.stdout`.

If you want a given `stdin` to be used, make sure to set the instance’s `use_rawinput` attribute to `False`, otherwise `stdin` will be ignored.
24.2.1 Cmd Objects

A `Cmd` instance has the following methods:

`Cmd.cmdloop (intro=None)`
Repeatedly issue a prompt, accept input, parse an initial prefix off the received input, and dispatch to action methods, passing them the remainder of the line as argument.

The optional argument is a banner or intro string to be issued before the first prompt (this overrides the `intro` class attribute).

If the `readline` module is loaded, input will automatically inherit `bash`-like history-list editing (e.g. `Control-P` scrolls back to the last command, `Control-N` forward to the next one, `Control-F` moves the cursor to the right non-destructively, `Control-B` moves the cursor to the left non-destructively, etc.).

An end-of-file on input is passed back as the string ‘EOF’.

An interpreter instance will recognize a command name `foo` if and only if it has a method `do_foo()`. As a special case, a line beginning with the character ‘?’ is dispatched to the method `do_help()`. As another special case, a line beginning with the character ‘!’ is dispatched to the method `do_shell()` (if such a method is defined).

This method will return when the `postcmd()` method returns a true value. The `stop` argument to `postcmd()` is the return value from the command’s corresponding `do_*()` method.

If completion is enabled, completing commands will be done automatically, and completing of commands `args` is done by calling `complete_foo()` with arguments `text`, `line`, `begidx`, and `endidx`. `text` is the string prefix we are attempting to match: all returned matches must begin with it. `line` is the current input line with leading whitespace removed, `begidx` and `endidx` are the beginning and ending indexes of the prefix text, which could be used to provide different completion depending upon which position the argument is in.

All subclasses of `Cmd` inherit a predefined `do_help()`. This method, called with an argument ‘bar’, invokes the corresponding method `help_bar()`, and if that is not present, prints the docstring of `do_bar()`, if available. With no argument, `do_help()` lists all available help topics (that is, all commands with corresponding `help_*()` methods or commands that have docstrings), and also lists any undocumented commands.

`Cmd.onecmd (str)`
Interpret the argument as though it had been typed in response to the prompt. This may be overridden, but should not normally need to be; see the `precmd()` and `postcmd()` methods for useful execution hooks. The return value is a flag indicating whether interpretation of commands by the interpreter should stop. If there is a `do_*()` method for the command `str`, the return value of that method is returned, otherwise the return value from the `default()` method is returned.

`Cmd.emptyline ()`
Method called when an empty line is entered in response to the prompt. If this method is not overridden, it repeats the last nonempty command entered.

`Cmd.default (line)`
Method called on an input line when the command prefix is not recognized. If this method is not overridden, it prints an error message and returns.

`Cmd.completedefault (text, line, begidx, endidx)`
Method called to complete an input line when no command-specific `complete_*()` method is available. By default, it returns an empty list.

`Cmd.precmd (line)`
Hook method executed just before the command line `line` is interpreted, but after the input prompt is generated and issued. This method is a stub in `Cmd`; it exists to be overridden by subclasses. The return value is used as the command which will be executed by the `onecmd()` method; the `precmd()` implementation may re-write the command or simply return `line` unchanged.

`Cmd.postcmd (stop, line)`
Hook method executed just after a command dispatch is finished. This method is a stub in `Cmd`; it exists to be overridden by subclasses. `line` is the command line which was executed, and `stop` is a flag which
indicates whether execution will be terminated after the call to \texttt{postcmd()}; this will be the return value of the \texttt{onecmd()} method. The return value of this method will be used as the new value for the internal flag which corresponds to \texttt{stop}; returning false will cause interpretation to continue.

\texttt{Cmd.preloop()}
\begin{itemize}
  \item Hook method executed once when \texttt{cmdloop()} is called. This method is a stub in \texttt{Cmd}; it exists to be overridden by subclasses.
\end{itemize}

\texttt{Cmd.postloop()}
\begin{itemize}
  \item Hook method executed once when \texttt{cmdloop()} is about to return. This method is a stub in \texttt{Cmd}; it exists to be overridden by subclasses.
\end{itemize}

Instances of \texttt{Cmd} subclasses have some public instance variables:

\texttt{Cmd.prompt}
\begin{description}
  \item The prompt issued to solicit input.
\end{description}

\texttt{Cmd.identchars}
\begin{description}
  \item The string of characters accepted for the command prefix.
\end{description}

\texttt{Cmd.lastcmd}
\begin{description}
  \item The last nonempty command prefix seen.
\end{description}

\texttt{Cmd.intro}
\begin{description}
  \item A string to issue as an intro or banner. May be overridden by giving the \texttt{cmdloop()} method an argument.
\end{description}

\texttt{Cmd.doc_header}
\begin{description}
  \item The header to issue if the help output has a section for documented commands.
\end{description}

\texttt{Cmd.misc_header}
\begin{description}
  \item The header to issue if the help output has a section for miscellaneous help topics (that is, there are \texttt{help_*()} methods without corresponding \texttt{do_*()} methods).
\end{description}

\texttt{Cmd.undoc_header}
\begin{description}
  \item The header to issue if the help output has a section for undocumented commands (that is, there are \texttt{do_*()} methods without corresponding \texttt{help_*()} methods).
\end{description}

\texttt{Cmd.ruler}
\begin{description}
  \item The character used to draw separator lines under the help-message headers. If empty, no ruler line is drawn. It defaults to '"'.
\end{description}

\texttt{Cmd.use_rawinput}
\begin{description}
  \item A flag, defaulting to true. If true, \texttt{cmdloop()} uses \texttt{input()} to display a prompt and read the next command; if false, \texttt{sys.stdout.write()} and \texttt{sys.stdin.readline()} are used. (This means that by importing \texttt{readline}, on systems that support it, the interpreter will automatically support \texttt{Emacs}-like line editing and command-history keystrokes.)
\end{description}

\subsection{24.2.2 \texttt{Cmd} Example}

The \texttt{cmd} module is mainly useful for building custom shells that let a user work with a program interactively.

This section presents a simple example of how to build a shell around a few of the commands in the \texttt{turtle} module.

Basic turtle commands such as \texttt{forward()} are added to a \texttt{Cmd} subclass with method named \texttt{do_forward()}. The argument is converted to a number and dispatched to the turtle module. The docstring is used in the help utility provided by the shell.

The example also includes a basic record and playback facility implemented with the \texttt{precmd()} method which is responsible for converting the input to lowercase and writing the commands to a file. The \texttt{do_playback()} method reads the file and adds the recorded commands to the \texttt{cmdqueue} for immediate playback:

\begin{verbatim}
import cmd, sys
from turtle import *
\end{verbatim}
```python
class TurtleShell(cmd.Cmd):
    intro = 'Welcome to the turtle shell. Type help or ? to list commands.

    prompt = '(turtle) '
    file = None

    # ----- basic turtle commands -----
    def do_forward(self, arg):
        'Move the turtle forward by the specified distance: FORWARD 10'
        forward(*parse(arg))

    def do_right(self, arg):
        'Turn turtle right by given number of degrees: RIGHT 20'
        right(*parse(arg))

    def do_left(self, arg):
        'Turn turtle left by given number of degrees: LEFT 90'
        left(*parse(arg))

    def do_goto(self, arg):
        'Move turtle to an absolute position with changing orientation. GOTO 100 200'
        goto(*parse(arg))

    def do_home(self, arg):
        'Return turtle to the home position: HOME'
        home()

    def do_circle(self, arg):
        'Draw circle with given radius an options extent and steps: CIRCLE 50'
        circle(*parse(arg))

    def do_position(self, arg):
        'Print the current turle position: POSITION'
        print('Current position is %d %d' % position())

    def do_heading(self, arg):
        'Print the current turle heading in degrees: HEADING'
        print('Current heading is %d' % (heading(),))

    def do_color(self, arg):
        'Set the color: COLOR BLUE'
        color(arg.lower())

    def do_undo(self, arg):
        'Undo (repeatedly) the last turtle action(s): UNDO'

    def do_reset(self, arg):
        'Clear the screen and return turtle to center: RESET'
        reset()

    def do_bye(self, arg):
        'Stop recording, close the turtle window, and exit: BYE'
        print('Thank you for using Turtle')
        self.close()
        bye()
        return True

    # ----- record and playback -----
    def do_record(self, arg):
        'Save future commands to filename: RECORD rose.cmd'
        self.file = open(arg, 'w')

    def do_playback(self, arg):
        'Playback commands from a file: PLAYBACK rose.cmd'

    def precmd(self, line):
        line = line.lower()
        if self.file and 'playback' not in line:
            print(line, file=self.file)
```

Chapter 24. Program Frameworks
return line

def close(self):
    if self.file:
        self.file.close()
    self.file = None

def parse(arg):
    'Convert a series of zero or more numbers to an argument tuple'
    return tuple(map(int, arg.split()))

if __name__ == '__main__':
    TurtleShell().cmdloop()

Here is a sample session with the turtle shell showing the help functions, using blank lines to repeat commands, and the simple record and playback facility:

Welcome to the turtle shell. Type help or ? to list commands.

(turtle) ?

Documented commands (type help <topic>):
========================================
bye    color    goto    home    playback    record    right
circle forward heading left position reset undo

(turtle) help forward
Move the turtle forward by the specified distance: FORWARD 10
(turtle) record spiral.cmd
(turtle) position
Current position is 0 0

(turtle) heading
Current heading is 0

(turtle) reset
(turtle) circle 20
(turtle) right 30
(turtle) circle 40
(turtle) right 30
(turtle) circle 60
(turtle) right 30
(turtle) circle 80
(turtle) right 30
(turtle) circle 100
(turtle) right 30
(turtle) circle 120
(turtle) right 30
(turtle) circle 120
(turtle) heading
Current heading is 180

(turtle) forward 100
(turtle) right 90
(turtle) forward 100
(turtle) right 90
(turtle) forward 400
(turtle) right 90

24.2. cmd — Support for line-oriented command interpreters 1035
(turtle) forward 500
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 300
(turtle) playback spiral.cmd
Current position is 0 0
Current heading is 0
Current heading is 180
(turtle) bye
Thank you for using Turtle

24.3 shlex — Simple lexical analysis

Source code: Lib/shlex.py

The shlex class makes it easy to write lexical analyzers for simple syntaxes resembling that of the Unix shell. This will often be useful for writing minilanguages, (for example, in run control files for Python applications) or for parsing quoted strings.

The shlex module defines the following functions:

shlex.split(s, comments=False, posix=True)
Split the string s using shell-like syntax. If comments is False (the default), the parsing of comments in the given string will be disabled (setting the commenters attribute of the shlex instance to the empty string). This function operates in POSIX mode by default, but uses non-POSIX mode if the posix argument is false.

Note: Since the split() function instantiates a shlex instance, passing None for s will read the string to split from standard input.

shlex.quote(s)
Return a shell-escaped version of the string s. The returned value is a string that can safely be used as one token in a shell command line, for cases where you cannot use a list.

This idiom would be unsafe:

>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print(command)  # executed by a shell: boom!
ls -l somefile; rm -rf ~

quote() lets you plug the security hole:

>>> command = 'ls -l {}'.format(quote(filename))
>>> print(command)
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print(remote_command)
ssh home 'ls -l """"""somefile; rm -rf """"""

The quoting is compatible with UNIX shells and with split():
>>> remote_command = split(remote_command)
>>> remote_command
['ssh', 'home', "ls -l 'somefile; rm -rf ~'\"]
>>> command = split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']

New in version 3.3.

The `shlex` module defines the following class:

```python
class shlex (instream=None, infile=None, posix=False)
```

A `shlex` instance or subclass instance is a lexical analyzer object. The initialization argument, if present, specifies where to read characters from. It must be a file-/stream-like object with `read()` and `readline()` methods, or a string. If no argument is given, input will be taken from `sys.stdin`. The second optional argument is a filename string, which sets the initial value of the `infile` attribute. If the `instream` argument is omitted or equal to `sys.stdin`, this second argument defaults to “stdin”. The `posix` argument defines the operational mode: when `posix` is not true (default), the `shlex` instance will operate in compatibility mode. When operating in POSIX mode, `shlex` will try to be as close as possible to the POSIX shell parsing rules.

See Also:

- Module `configparser`: Parser for configuration files similar to the Windows `.ini` files.

### 24.3.1 `shlex` Objects

A `shlex` instance has the following methods:

- `shlex.get_token()`
  Return a token. If tokens have been stacked using `push_token()`, pop a token off the stack. Otherwise, read one from the input stream. If reading encounters an immediate end-of-file, `eof` is returned (the empty string (“”) in non-POSIX mode, and `None` in POSIX mode).

- `shlex.push_token(str)`
  Push the argument onto the token stack.

- `shlex.read_token()`
  Read a raw token. Ignore the pushback stack, and do not interpret source requests. (This is not ordinarily a useful entry point, and is documented here only for the sake of completeness.)

- `shlex.sourcehook(filename)`
  When `shlex` detects a source request (see `source` below) this method is given the following token as argument, and expected to return a tuple consisting of a filename and an open file-like object.

Normally, this method first strips any quotes off the argument. If the result is an absolute pathname, or there was no previous source request in effect, or the previous source was a stream (such as `sys.stdin`), the result is left alone. Otherwise, if the result is a relative pathname, the directory part of the name of the file immediately before it on the source inclusion stack is prepended (this behavior is like the way the C preprocessor handles `#include "file.h"`).

The result of the manipulations is treated as a filename, and returned as the first component of the tuple, with `open()` called on it to yield the second component. (Note: this is the reverse of the order of arguments in instance initialization!)

This hook is exposed so that you can use it to implement directory search paths, addition of file extensions, and other namespace hacks. There is no corresponding ‘close’ hook, but a `shlex` instance will call the `close()` method of the sourced input stream when it returns EOF.

For more explicit control of source stacking, use the `push_source()` and `pop_source()` methods.

- `shlex.push_source(newstream, newfile=None)`
  Push an input source stream onto the input stack. If the filename argument is specified it will later be
available for use in error messages. This is the same method used internally by the sourcehook() method.

```
shlex.pop_source()
```

Pop the last-pushed input source from the input stack. This is the same method used internally when the lexer reaches EOF on a stacked input stream.

```
shlex.error_leader(infile=None, lineno=None)
```

This method generates an error message leader in the format of a Unix C compiler error label; the format is "%%s", line %d: ", where the %s is replaced with the name of the current source file and the %d with the current input line number (the optional arguments can be used to override these).

This convenience is provided to encourage shlex users to generate error messages in the standard, parseable format understood by Emacs and other Unix tools.

Instances of shlex subclasses have some public instance variables which either control lexical analysis or can be used for debugging:

```
shlex.commenters
```

The string of characters that are recognized as comment beginners. All characters from the comment begin-
ner to end of line are ignored. Includes just ‘#’ by default.

```
shlex.wordchars
```

The string of characters that will accumulate into multi-character tokens. By default, includes all ASCII alphanumeric and underscore.

```
shlex.whitespace
```

Characters that will be considered whitespace and skipped. Whitespace bounds tokens. By default, includes space, tab, linefeed and carriage-return.

```
shlex.escape
```

Characters that will be considered as escape. This will be only used in POSIX mode, and includes just ‘\’ by default.

```
shlex.quotes
```

Characters that will be considered string quotes. The token accumulates until the same quote is encountered again (thus, different quote types protect each other as in the shell.) By default, includes ASCII single and double quotes.

```
shlex.escapedquotes
```

Characters in quotes that will interpret escape characters defined in escape. This is only used in POSIX mode, and includes just ‘”’ by default.

```
shlex.whitespace_split
```

If True, tokens will only be split in whitespaces. This is useful, for example, for parsing command lines with shlex, getting tokens in a similar way to shell arguments.

```
shlex.infile
```

The name of the current input file, as initially set at class instantiation time or stacked by later source requests. It may be useful to examine this when constructing error messages.

```
shlex.instream
```

The input stream from which this shlex instance is reading characters.

```
shlex.source
```

This attribute is None by default. If you assign a string to it, that string will be recognized as a lexical-level inclusion request similar to the source keyword in various shells. That is, the immediately following token will opened as a filename and input taken from that stream until EOF, at which point the close() method of that stream will be called and the input source will again become the original input stream. Source requests may be stacked any number of levels deep.

```
shlex.debug
```

If this attribute is numeric and 1 or more, a shlex instance will print verbose progress output on its behavior. If you need to use this, you can read the module source code to learn the details.
The Python Library Reference, Release 3.3.3

shlex.lineno
Source line number (count of newlines seen so far plus one).

shlex.token
The token buffer. It may be useful to examine this when catching exceptions.

shlex.eof
Token used to determine end of file. This will be set to the empty string ('"'), in non-POSIX mode, and to None in POSIX mode.

24.3.2 Parsing Rules

When operating in non-POSIX mode, shlex will try to obey to the following rules.

• Quote characters are not recognized within words ("Do"Not"Separate is parsed as the single word Do"Not"Separate);
  • Escape characters are not recognized;
  • Enclosing characters in quotes preserve the literal value of all characters within the quotes;
  • Closing quotes separate words ("Do"Separate is parsed as "Do" and Separate);
  • If whitespace_split is False, any character not declared to be a word character, whitespace, or a quote will be returned as a single-character token. If it is True, shlex will only split words in whitespaces;
  • EOF is signaled with an empty string (""");
  • It’s not possible to parse empty strings, even if quoted.

When operating in POSIX mode, shlex will try to obey to the following parsing rules.

• Quotes are stripped out, and do not separate words ("Do"Not"Separate" is parsed as the single word DoNotSeparate);
  • Non-quoted escape characters (e.g. ‘\’) preserve the literal value of the next character that follows;
  • Enclosing characters in quotes which are not part of escapedquotes (e.g. ‘”’) preserve the literal value of all characters within the quotes;
  • Enclosing characters in quotes which are part of escapedquotes (e.g. ‘”’) preserves the literal value of all characters within the quotes, with the exception of the characters mentioned in escape. The escape characters retain its special meaning only when followed by the quote in use, or the escape character itself. Otherwise the escape character will be considered a normal character.
  • EOF is signaled with a None value;
  • Quoted empty strings (""") are allowed.
Tk/Tcl has long been an integral part of Python. It provides a robust and platform independent windowing toolkit, that is available to Python programmers using the tkinter package, and its extension, the tkinter.tix and the tkinter.ttk modules.

The tkinter package is a thin object-oriented layer on top of Tcl/Tk. To use tkinter, you don’t need to write Tcl code, but you will need to consult the Tk documentation, and occasionally the Tcl documentation. tkinter is a set of wrappers that implement the Tk widgets as Python classes. In addition, the internal module _tkinter provides a threadsafe mechanism which allows Python and Tcl to interact.

tkinter’s chief virtues are that it is fast, and that it usually comes bundled with Python. Although its standard documentation is weak, good material is available, which includes: references, tutorials, a book and others. tkinter is also famous for having an outdated look and feel, which has been vastly improved in Tk 8.5. Nevertheless, there are many other GUI libraries that you could be interested in. For more information about alternatives, see the Other Graphical User Interface Packages section.

25.1 tkinter — Python interface to Tcl/Tk

The tkinter package (“Tk interface”) is the standard Python interface to the Tk GUI toolkit. Both Tk and tkinter are available on most Unix platforms, as well as on Windows systems. (Tk itself is not part of Python; it is maintained at ActiveState.) You can check that tkinter is properly installed on your system by running python -m tkinter from the command line; this should open a window demonstrating a simple Tk interface.

See Also:
Python Tkinter Resources The Python Tkinter Topic Guide provides a great deal of information on using Tk from Python and links to other sources of information on Tk.
TKDocs Extensive tutorial plus friendlier widget pages for some of the widgets.
Tkinter docs from effbot Online reference for tkinter supported by effbot.org.
Tcl/Tk manual Official manual for the latest tcl/tk version.
Programming Python Book by Mark Lutz, has excellent coverage of Tkinter.
Modern Tkinter for Busy Python Developers Book by Mark Rozerman about building attractive and modern graphical user interfaces with Python and Tkinter.
25.1.1 Tkinter Modules

Most of the time, `tkinter` is all you really need, but a number of additional modules are available as well. The Tk interface is located in a binary module named `_tkinter`. This module contains the low-level interface to Tk, and should never be used directly by application programmers. It is usually a shared library (or DLL), but might in some cases be statically linked with the Python interpreter.

In addition to the Tk interface module, `tkinter` includes a number of Python modules, `tkinter.constants` being one of the most important. Importing `tkinter` will automatically import `tkinter.constants`, so, usually, to use Tkinter all you need is a simple import statement:

```python
import tkinter
```

Or, more often:

```python
from tkinter import *
```

```python
class tkinter.Tk (screenName=None, baseName=None, className='Tk', useTk=1)
```

The `Tk` class is instantiated without arguments. This creates a toplevel widget of Tk which usually is the main window of an application. Each instance has its own associated Tcl interpreter.

```python
tkinter.Tcl (screenName=None, baseName=None, className='Tk', useTk=0)
```

The `Tcl()` function is a factory function which creates an object much like that created by the `Tk` class, except that it does not initialize the Tk subsystem. This is most often useful when driving the Tcl interpreter in an environment where one doesn’t want to create extraneous toplevel windows, or where one cannot (such as Unix/Linux systems without an X server). An object created by the `Tcl()` object can have a Toplevel window created (and the Tk subsystem initialized) by calling its `loadtk()` method.

Other modules that provide Tk support include:

- `tkinter.scrolledtext` Text widget with a vertical scroll bar built in.
- `tkinter.colorchooser` Dialog to let the user choose a color.
- `tkinter.commondialog` Base class for the dialogs defined in the other modules listed here.
- `tkinter.filedialog` Common dialogs to allow the user to specify a file to open or save.
- `tkinter.font` Utilities to help work with fonts.
- `tkinter.messagebox` Access to standard Tk dialog boxes.
- `tkinter.simpledialog` Basic dialogs and convenience functions.
- `tkinter.dnd` Drag-and-drop support for `tkinter`. This is experimental and should become deprecated when it is replaced with the Tk DND.
- `turtle` Turtle graphics in a Tk window.

25.1.2 Tkinter Life Preserver

This section is not designed to be an exhaustive tutorial on either Tk or Tkinter. Rather, it is intended as a stop gap, providing some introductory orientation on the system.

Credits:

- Tk was written by John Ousterhout while at Berkeley.
- Tkinter was written by Steen Lumholt and Guido van Rossum.
- This Life Preserver was written by Matt Conway at the University of Virginia.
- The HTML rendering, and some liberal editing, was produced from a FrameMaker version by Ken Manheimer.
- Fredrik Lundh elaborated and revised the class interface descriptions, to get them current with Tk 4.2.
- Mike Clarkson converted the documentation to LaTeX, and compiled the User Interface chapter of the reference manual.
How To Use This Section

This section is designed in two parts: the first half (roughly) covers background material, while the second half can be taken to the keyboard as a handy reference.

When trying to answer questions of the form “how do I do blah”, it is often best to find out how to do”blah” in straight Tk, and then convert this back into the corresponding `tkinter` call. Python programmers can often guess at the correct Python command by looking at the Tk documentation. This means that in order to use Tkinter, you will have to know a little bit about Tk. This document can’t fulfill that role, so the best we can do is point you to the best documentation that exists. Here are some hints:

- The authors strongly suggest getting a copy of the Tk man pages. Specifically, the man pages in the `manN` directory are most useful. The `man3` man pages describe the C interface to the Tk library and thus are not especially helpful for script writers.
- Addison-Wesley publishes a book called Tcl and the Tk Toolkit by John Ousterhout (ISBN 0-201-63337-X) which is a good introduction to Tcl and Tk for the novice. The book is not exhaustive, and for many details it defers to the man pages.
- `tkinter/__init__.py` is a last resort for most, but can be a good place to go when nothing else makes sense.

See Also:

- [Tcl/Tk 8.6 man pages](#) The Tcl/Tk manual on www.tcl.tk.
- [ActiveState Tcl Home Page](#) The Tk/Tcl development is largely taking place at ActiveState.
- [Tcl and the Tk Toolkit](#) The book by John Ousterhout, the inventor of Tcl.
- [Practical Programming in Tcl and Tk](#) Brent Welch’s encyclopedic book.

A Simple Hello World Program

```python
import tkinter as tk

class Application(tk.Frame):
    def __init__(self, master=None):
        tk.Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

    def createWidgets(self):
        self.hi_there = tk.Button(self)
        self.hi_there['text'] = 'Hello World
(click me)
        self.hi_there['command'] = self.say_hi
        self.hi_there.pack(side="top")

        self.QUIT = tk.Button(self, text="QUIT", fg="red",
            command=root.destroy)
        self.QUIT.pack(side="bottom")

    def say_hi(self):
        print("hi there, everyone!")

root = tk.Tk()
app = Application(master=root)
app.mainloop()
```

25.1. `tkinter` — Python interface to Tcl/Tk
25.1.3 A (Very) Quick Look at Tcl/Tk

The class hierarchy looks complicated, but in actual practice, application programmers almost always refer to the classes at the very bottom of the hierarchy.

Notes:

- These classes are provided for the purposes of organizing certain functions under one namespace. They aren’t meant to be instantiated independently.
- The Tk class is meant to be instantiated only once in an application. Application programmers need not instantiate one explicitly, the system creates one whenever any of the other classes are instantiated.
- The Widget class is not meant to be instantiated, it is meant only for subclassing to make “real” widgets (in C++, this is called an ‘abstract class’).

To make use of this reference material, there will be times when you will need to know how to read short passages of Tk and how to identify the various parts of a Tk command. (See section Mapping Basic Tk into Tkinter for the tkinter equivalents of what’s below.)

Tk scripts are Tcl programs. Like all Tcl programs, Tk scripts are just lists of tokens separated by spaces. A Tk widget is just its class, the options that help configure it, and the actions that make it do useful things.

To make a widget in Tk, the command is always of the form:

classCommand newPathname options

- **classCommand** denotes which kind of widget to make (a button, a label, a menu...)
- **newPathname** is the new name for this widget. All names in Tk must be unique. To help enforce this, widgets in Tk are named with *pathnames*, just like files in a file system. The top level widget, the *root*, is called *. (period) and children are delimited by more periods. For example, *.myApp.controlPanel.okButton* might be the name of a widget.
- **options** configure the widget’s appearance and in some cases, its behavior. The options come in the form of a list of flags and values. Flags are preceded by a '-', like Unix shell command flags, and values are put in quotes if they are more than one word.

For example:

```
button .fred -fg red -text "hi there"
```

Once created, the pathname to the widget becomes a new command. This new *widget command* is the programmer’s handle for getting the new widget to perform some *action*. In C, you’d express this as *someAction(fred, someOptions)*, in C++, you would express this as fred.someAction(someOptions), and in Tk, you say:

```
.fred someAction someOptions
```

Note that the object name, .fred, starts with a dot.

As you’d expect, the legal values for *someAction* will depend on the widget’s class: .fred disable works if fred is a button (fred gets greyed out), but does not work if fred is a label (disabling of labels is not supported in Tk).

The legal values of *someOptions* is action dependent. Some actions, like disable, require no arguments, others, like a text-entry box’s delete command, would need arguments to specify what range of text to delete.

25.1.4 Mapping Basic Tk into Tkinter

Class commands in Tk correspond to class constructors in Tkinter.

```
button .fred =====> fred = Button()
```
The master of an object is implicit in the new name given to it at creation time. In Tkinter, masters are specified explicitly.

```python
button .panel.fred =====> fred = Button(panel)
```

The configuration options in Tk are given in lists of hyphenated tags followed by values. In Tkinter, options are specified as keyword-arguments in the instance constructor, and keyword-args for configure calls or as instance indices, in dictionary style, for established instances. See section Setting Options on setting options.

```python
button .fred -fg red =====> fred = Button(panel, fg="red")
.fredef configure -fg red =====> fred["fg"] = red
OR ==> fred.config(fg="red")
```

In Tk, to perform an action on a widget, use the widget name as a command, and follow it with an action name, possibly with arguments (options). In Tkinter, you call methods on the class instance to invoke actions on the widget. The actions (methods) that a given widget can perform are listed in tkinter/__init__.py.

```python
.fred invoke =====> fred.invoke()
```

To give a widget to the packer (geometry manager), you call pack with optional arguments. In Tkinter, the Pack class holds all this functionality, and the various forms of the pack command are implemented as methods. All widgets in tkinter are subclassed from the Packer, and so inherit all the packing methods. See the tkinter.tix module documentation for additional information on the Form geometry manager.

```python
pack .fred -side left =====> fred.pack(side="left")
```

### 25.1.5 How Tk and Tkinter are Related

From the top down:

**Your App Here (Python)** A Python application makes a tkinter call.

**tkinter (Python Package)** This call (say, for example, creating a button widget), is implemented in the tkinter package, which is written in Python. This Python function will parse the commands and the arguments and convert them into a form that makes them look as if they had come from a Tk script instead of a Python script.

**_tkinter (C)** These commands and their arguments will be passed to a C function in the _tkinter - note the underscore - extension module.

**Tk Widgets (C and Tcl)** This C function is able to make calls into other C modules, including the C functions that make up the Tk library. Tk is implemented in C and some Tcl. The Tcl part of the Tk widgets is used to bind certain default behaviors to widgets, and is executed once at the point where the Python tkinter package is imported. (The user never sees this stage).

**Tk (C)** The Tk part of the Tk Widgets implement the final mapping to ...

**Xlib (C)** the Xlib library to draw graphics on the screen.

### 25.1.6 Handy Reference

**Setting Options**

Options control things like the color and border width of a widget. Options can be set in three ways:

**At object creation time, using keyword arguments**

```python
fred = Button(self, fg="red", bg="blue")
```

**After object creation, treating the option name like a dictionary index**

```python
fred["fg"] = "red"
fred["bg"] = "blue"
```

Use the config() method to update multiple attrs subsequent to object creation.

---

25.1. Tkinter — Python interface to Tcl/Tk  1045
fred.config(fg="red", bg="blue")

For a complete explanation of a given option and its behavior, see the Tk man pages for the widget in question.

Note that the man pages list “STANDARD OPTIONS” and “WIDGET SPECIFIC OPTIONS” for each widget. The former is a list of options that are common to many widgets, the latter are the options that are idiosyncratic to that particular widget. The Standard Options are documented on the options(3) man page.

No distinction between standard and widget-specific options is made in this document. Some options don’t apply to some kinds of widgets. Whether a given widget responds to a particular option depends on the class of the widget; buttons have a `command` option, labels do not.

The options supported by a given widget are listed in that widget’s man page, or can be queried at runtime by calling the `config()` method without arguments, or by calling the `keys()` method on that widget. The return value of these calls is a dictionary whose key is the name of the option as a string (for example, `'relief'`) and whose values are 5-tuples.

Some options, like `bg` are synonyms for common options with long names (`bg` is shorthand for “background”). Passing the `config()` method the name of a shorthand option will return a 2-tuple, not 5-tuple. The 2-tuple passed back will contain the name of the synonym and the “real” option (such as (‘bg’, ‘background’)).

<table>
<thead>
<tr>
<th>Index</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>option name</td>
<td>‘relief’</td>
</tr>
<tr>
<td>1</td>
<td>option name for database lookup</td>
<td>‘relief’</td>
</tr>
<tr>
<td>2</td>
<td>option class for database lookup</td>
<td>‘Relief’</td>
</tr>
<tr>
<td>3</td>
<td>default value</td>
<td>‘raised’</td>
</tr>
<tr>
<td>4</td>
<td>current value</td>
<td>‘groove’</td>
</tr>
</tbody>
</table>

Example:

```python
>>> print(fred.config())
```

Of course, the dictionary printed will include all the options available and their values. This is meant only as an example.

### The Packer

The packer is one of Tk’s geometry-management mechanisms. Geometry managers are used to specify the relative positioning of the positioning of widgets within their container - their mutual `master`. In contrast to the more cumbersome `placer` (which is used less commonly, and we do not cover here), the packer takes qualitative relationship specification - above, to the left of, filling, etc - and works everything out to determine the exact placement coordinates for you.

The size of any `master` widget is determined by the size of the “slave widgets” inside. The packer is used to control where slave widgets appear inside the master into which they are packed. You can pack widgets into frames, and frames into other frames, in order to achieve the kind of layout you desire. Additionally, the arrangement is dynamically adjusted to accommodate incremental changes to the configuration, once it is packed.

Note that widgets do not appear until they have had their geometry specified with a geometry manager. It’s a common early mistake to leave out the geometry specification, and then be surprised when the widget is created but nothing appears. A widget will appear only after it has had, for example, the packer’s `pack()` method applied to it.

The `pack()` method can be called with keyword-option/value pairs that control where the widget is to appear within its container, and how it is to behave when the main application window is resized. Here are some examples:

```python
fred.pack()  # defaults to side = "top"
fred.pack(side="left")
fred.pack(expand=1)
```
Packer Options

For more extensive information on the packer and the options that it can take, see the man pages and page 183 of John Ousterhout’s book.

anchor Anchor type. Denotes where the packer is to place each slave in its parcel.

expand Boolean, 0 or 1.


ipadx and ipady A distance - designating internal padding on each side of the slave widget.

padx and pady A distance - designating external padding on each side of the slave widget.

side Legal values are: ‘left’, ‘right’, ‘top’, ‘bottom’.

Coupling Widget Variables

The current-value setting of some widgets (like text entry widgets) can be connected directly to application variables by using special options. These options are variable, textvariable, onvalue, offvalue, and value. This connection works both ways: if the variable changes for any reason, the widget it’s connected to will be updated to reflect the new value.

Unfortunately, in the current implementation of tkinter it is not possible to hand over an arbitrary Python variable to a widget through a variable or textvariable option. The only kinds of variables for which this works are variables that are subclassed from a class called Variable, defined in tkinter.

There are many useful subclasses of Variable already defined: StringVar, IntVar, DoubleVar, and BooleanVar. To read the current value of such a variable, call the get() method on it, and to change its value you call the set() method. If you follow this protocol, the widget will always track the value of the variable, with no further intervention on your part.

For example:

class App(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()

        self.entrythingy = Entry()
        self.entrythingy.pack()

        # here is the application variable
        self.contents = StringVar()
        # set it to some value
        self.contents.set("this is a variable")
        # tell the entry widget to watch this variable
        self.entrythingy["textvariable"] = self.contents

        # and here we get a callback when the user hits return.
        # we will have the program print out the value of the
        # application variable when the user hits return
        self.entrythingy.bind('<Key-Return>',
            self.print_contents)

    def print_contents(self, event):
        print("hi. contents of entry is now ---->",
            self.contents.get())
The Window Manager

In Tk, there is a utility command, `wm`, for interacting with the window manager. Options to the `wm` command allow you to control things like titles, placement, icon bitmaps, and the like. In `tkinter`, these commands have been implemented as methods on the `Wm` class. Toplevel widgets are subclassed from the `Wm` class, and so can call the `Wm` methods directly.

To get at the toplevel window that contains a given widget, you can often just refer to the widget’s master. Of course if the widget has been packed inside of a frame, the master won’t represent a toplevel window. To get at the toplevel window that contains an arbitrary widget, you can call the `_root()` method. This method begins with an underscore to denote the fact that this function is part of the implementation, and not an interface to Tk functionality.

Here are some examples of typical usage:

```python
from tkinter import *

class App(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()

    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()

myapp = App()

# here are method calls to the window manager class
myapp.master.title("My Do-Nothing Application")
myapp.master.maxsize(1000, 400)

# start the program
myapp.mainloop()
```

Tk Option Data Types

- **anchor** Legal values are points of the compass: "n", "ne", "e", "se", "s", "sw", "w", "nw", and also "center".

- **bitmap** There are eight built-in, named bitmaps: ‘error’, ‘gray25’, ‘gray50’, ‘hourglass’, ‘info’, ‘questhead’, ‘question’, ‘warning’. To specify an X bitmap filename, give the full path to the file, preceded with an @, as in "@/usr/contrib/bitmap/gumby.bit".

- **boolean** You can pass integers 0 or 1 or the strings "yes" or "no".

- **callback** This is any Python function that takes no arguments. For example:

  ```python
def print_it():
    print("hi there")
fred["command"] = print_it
```

- **color** Colors can be given as the names of X colors in the rgb.txt file, or as strings representing RGB values in 4 bit: "#RGB", 8 bit: "#RRGGBB", 12 bit: "#RRRRGGBBBB", or 16 bit "#RRRRGRRRRGGGGGGBBBB" ranges, where R,G,B here represent any legal hex digit. See page 160 of Ousterhout’s book for details.

- **cursor** The standard X cursor names from cursorfont.h can be used, without the XC_ prefix. For example to get a hand cursor (XC_hand2), use the string "hand2". You can also specify a bitmap and mask file of your own. See page 179 of Ousterhout’s book.

- **distance** Screen distances can be specified in either pixels or absolute distances. Pixels are given as numbers and absolute distances as strings, with the trailing character denoting units: c for centimetres, i for inches, m for millimetres, p for printer’s points. For example, 3.5 inches is expressed as "3.5i".
font  Tk uses a list font name format, such as `{courier 10 bold}`. Font sizes with positive numbers are measured in points; sizes with negative numbers are measured in pixels.

geometry  This is a string of the form `widthxheight`, where width and height are measured in pixels for most widgets (in characters for widgets displaying text). For example: `fred["geometry"] = "200x100"`.

justify  Legal values are the strings: "left", "center", "right", and "fill".

region  This is a string with four space-delimited elements, each of which is a legal distance (see above). For example: "2 3 4 5" and "3i 2i 4.5i 2i" and "3c 2c 4c 10.43c" are all legal regions.

relief  Determines what the border style of a widget will be. Legal values are: "raised", "sunken", "flat", "groove", and "ridge".

scrollcommand  This is almost always the `set()` method of some scrollbar widget, but can be any widget method that takes a single argument.

wrap:  Must be one of: "none", "char", or "word".

## Bindings and Events

The bind method from the widget command allows you to watch for certain events and to have a callback function trigger when that event type occurs. The form of the bind method is:

```python
def bind(self, sequence, func, add=''):  
```

where:

- `sequence` is a string that denotes the target kind of event. (See the bind man page and page 201 of John Ousterhout’s book for details).
- `func` is a Python function, taking one argument, to be invoked when the event occurs. An Event instance will be passed as the argument. (Functions deployed this way are commonly known as callbacks.)
- `add` is optional, either "" or "+". Passing an empty string denotes that this binding is to replace any other bindings that this event is associated with. Passing a "+" means that this function is to be added to the list of functions bound to this event type.

For example:

```python
def turnRed(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turnRed)
```

Notice how the widget field of the event is being accessed in the `turnRed()` callback. This field contains the widget that caught the X event. The following table lists the other event fields you can access, and how they are denoted in Tk, which can be useful when referring to the Tk man pages.

<table>
<thead>
<tr>
<th>Tk</th>
<th>Tkinter Event Field</th>
<th>Tk</th>
<th>Tkinter Event Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>%f</td>
<td>focus</td>
<td>%A</td>
<td>char</td>
</tr>
<tr>
<td>%h</td>
<td>height</td>
<td>%E</td>
<td>send_event</td>
</tr>
<tr>
<td>%k</td>
<td>keycode</td>
<td>%K</td>
<td>keysym</td>
</tr>
<tr>
<td>%s</td>
<td>state</td>
<td>%N</td>
<td>keysym_num</td>
</tr>
<tr>
<td>%t</td>
<td>time</td>
<td>%T</td>
<td>type</td>
</tr>
<tr>
<td>%w</td>
<td>width</td>
<td>%W</td>
<td>widget</td>
</tr>
<tr>
<td>%x</td>
<td>x</td>
<td>%X</td>
<td>x_root</td>
</tr>
<tr>
<td>%y</td>
<td>y</td>
<td>%Y</td>
<td>y_root</td>
</tr>
</tbody>
</table>

## The index Parameter

A number of widgets require “index” parameters to be passed. These are used to point at a specific place in a Text widget, or to particular characters in an Entry widget, or to particular menu items in a Menu widget.
Entry widget indexes (index, view index, etc.) Entry widgets have options that refer to character positions in the text being displayed. You can use these tkinter functions to access these special points in text widgets:

```python
tkinter.AtEnd()
refers to the last position in the text
Deprecated since version 3.3.
```

```python
tkinter.AtInsert()
refers to the point where the text cursor is
Deprecated since version 3.3.
```

```python
tkinter.AtSelFirst()
indicates the beginning point of the selected text
Deprecated since version 3.3.
```

```python
tkinter.AtSelLast()
denotes the last point of the selected text and finally
Deprecated since version 3.3.
```

```python
tkinter.At(x, y)
refers to the character at pixel location *x*, *y* (with *y* not used in the case of a text entry widget, which contains a single line of text).
Deprecated since version 3.3.
```

Text widget indexes The index notation for Text widgets is very rich and is best described in the Tk man pages.

Menu indexes (menu.invoke(), menu.entryconfig(), etc.) Some options and methods for menus manipulate specific menu entries. Anytime a menu index is needed for an option or a parameter, you may pass in:

- an integer which refers to the numeric position of the entry in the widget, counted from the top, starting with 0;
- the string "active", which refers to the menu position that is currently under the cursor;
- the string "last" which refers to the last menu item;
- An integer preceded by @, as in @6, where the integer is interpreted as a y pixel coordinate in the menu’s coordinate system;
- the string "none", which indicates no menu entry at all, most often used with menu.activate() to deactivate all entries, and finally,
- a text string that is pattern matched against the label of the menu entry, as scanned from the top of the menu to the bottom. Note that this index type is considered after all the others, which means that matches for menu items labelled last, active, or none may be interpreted as the above literals, instead.

Images

Bitmap/Pixelmap images can be created through the subclasses of tkinter.Image:

- `BitmapImage` can be used for X11 bitmap data.
- `PhotoImage` can be used for GIF and PPM/PGM color bitmaps.

Either type of image is created through either the `file` or the `data` option (other options are available as well).

The image object can then be used wherever an `image` option is supported by some widget (e.g. labels, buttons, menus). In these cases, Tk will not keep a reference to the image. When the last Python reference to the image object is deleted, the image data is deleted as well, and Tk will display an empty box wherever the image was used.
25.2 tkinter.ttk — Tk themed widgets

The tkinter.ttk module provides access to the Tk themed widget set, introduced in Tk 8.5. If Python has not been compiled against Tk 8.5, this module can still be accessed if Tile has been installed. The former method using Tk 8.5 provides additional benefits including anti-aliased font rendering under X11 and window transparency (requiring a composition window manager on X11).

The basic idea for tkinter.ttk is to separate, to the extent possible, the code implementing a widget’s behavior from the code implementing its appearance.

See Also:

Tk Widget Styling Support  A document introducing theming support for Tk

25.2.1 Using Ttk

To start using Ttk, import its module:

```python
from tkinter import ttk
```

To override the basic Tk widgets, the import should follow the Tk import:

```python
from tkinter import *
from tkinter.ttk import *
```

That code causes several tkinter.ttk widgets (Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale and Scrollbar) to automatically replace the Tk widgets.

This has the direct benefit of using the new widgets which gives a better look and feel across platforms; however, the replacement widgets are not completely compatible. The main difference is that widget options such as “fg”, “bg” and others related to widget styling are no longer present in Ttk widgets. Instead, use the ttk.Style class for improved styling effects.

See Also:

Converting existing applications to use Tile widgets  A monograph (using Tcl terminology) about differences typically encountered when moving applications to use the new widgets.

25.2.2 Ttk Widgets

Ttk comes with 17 widgets, eleven of which already existed in tkinter: Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale and Scrollbar. The other six are new: Combobox, Notebook, Progressbar, Separator, Sizegrip and Treeview. And all them are subclasses of Widget.

Using the Ttk widgets gives the application an improved look and feel. As discussed above, there are differences in how the styling is coded.

Tk code:

```python
l1 = tkinter.Label(text="Test", fg="black", bg="white")
l2 = tkinter.Label(text="Test", fg="black", bg="white")
```

Ttk code:

```python
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")

l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

For more information about TtkStyling, see the Style class documentation.
25.2.3 Widget

ttk.Widget defines standard options and methods supported by Tk themed widgets and is not supposed to be directly instantiated.

Standard Options

All the ttk Widgets accepts the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>Specifies the window class. The class is used when querying the option database for the window’s other options, to determine the default bindtags for the window, and to select the widget’s default layout and style. This is a read-only which may only be specified when the window is created.</td>
</tr>
<tr>
<td>cursor</td>
<td>Specifies the mouse cursor to be used for the widget. If set to the empty string (the default), the cursor is inherited for the parent widget.</td>
</tr>
<tr>
<td>takefocus</td>
<td>Determines whether the window accepts the focus during keyboard traversal. 0, 1 or an empty string is returned. If 0 is returned, it means that the window should be skipped entirely during keyboard traversal. If 1, it means that the window should receive the input focus as long as it is viewable. And an empty string means that the traversal scripts make the decision about whether or not to focus on the window. May be used to specify a custom widget style.</td>
</tr>
<tr>
<td>style</td>
<td>May be used to specify a custom widget style.</td>
</tr>
</tbody>
</table>

Scrollable Widget Options

The following options are supported by widgets that are controlled by a scrollbar.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xscrollcommand</td>
<td>Used to communicate with horizontal scrollbars. When the view in the widget’s window change, the widget will generate a Tcl command based on the xscrollcommand. Usually this option consists of the method Scrollbar.set() of some scrollbar. This will cause the scrollbar to be updated whenever the view in the window changes.</td>
</tr>
<tr>
<td>yscrollcommand</td>
<td>Used to communicate with vertical scrollbars. For some more information, see above.</td>
</tr>
</tbody>
</table>

Label Options

The following options are supported by labels, buttons and other button-like widgets.
### The Python Library Reference, Release 3.3.3

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>Specifies a text string to be displayed inside the widget.</td>
</tr>
<tr>
<td>textvariable</td>
<td>Specifies a name whose value will be used in place of the text option resource.</td>
</tr>
<tr>
<td>underline</td>
<td>If set, specifies the index (0-based) of a character to underline in the text string. The underline character is used for mnemonic activation.</td>
</tr>
<tr>
<td>image</td>
<td>Specifies an image to display. This is a list of 1 or more elements. The first element is the default image name. The rest of the list is a sequence of statespec/value pairs as defined by <code>Style.map()</code>, specifying different images to use when the widget is in a particular state or a combination of states. All images in the list should have the same size.</td>
</tr>
<tr>
<td>compound</td>
<td>Specifies how to display the image relative to the text, in the case both text and images options are present. Valid values are: • text: display text only • image: display image only • top, bottom, left, right: display image above, below, left of, or right of the text, respectively. • none: the default. Display the image if present, otherwise the text.</td>
</tr>
<tr>
<td>width</td>
<td>If greater than zero, specifies how much space, in character widths, to allocate for the text label, if less than zero, specifies a minimum width. If zero or unspecified, the natural width of the text label is used.</td>
</tr>
</tbody>
</table>

#### Compatibility Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>May be set to “normal” or “disabled” to control the “disabled” state bit. This is a write-only option: setting it changes the widget state, but the <code>Widget.state()</code> method does not affect this option.</td>
</tr>
</tbody>
</table>

#### Widget States

The widget state is a bitmap of independent state flags.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>active</td>
<td>The mouse cursor is over the widget and pressing a mouse button will cause some action to occur</td>
</tr>
<tr>
<td>disabled</td>
<td>Widget is disabled under program control</td>
</tr>
<tr>
<td>focus</td>
<td>Widget has keyboard focus</td>
</tr>
<tr>
<td>pressed</td>
<td>Widget is being pressed</td>
</tr>
<tr>
<td>selected</td>
<td>“On”, “true”, or “current” for things like Checkbuttons and radiobuttons</td>
</tr>
<tr>
<td>background</td>
<td>Windows and Mac have a notion of an “active” or foreground window. The background state is set for widgets in a background window, and cleared for those in the foreground window</td>
</tr>
<tr>
<td>readonly</td>
<td>Widget should not allow user modification</td>
</tr>
<tr>
<td>alternate</td>
<td>A widget-specific alternate display format</td>
</tr>
<tr>
<td>invalid</td>
<td>The widget’s value is invalid</td>
</tr>
</tbody>
</table>

A state specification is a sequence of state names, optionally prefixed with an exclamation point indicating that the bit is off.

#### `ttk.Widget`

Besides the methods described below, the `ttk.Widget` supports the methods `tkinter.Widget.cget()` and `tkinter.Widget.configure()`. 

25.2. `ttk.ttk` — Tk themed widgets
class tkinter.ttk.Widget

    identify(x, y)
    Returns the name of the element at position x y, or the empty string if the point does not lie within any element.
    x and y are pixel coordinates relative to the widget.

instate(statespec, callback=None, *args, **kw)
    Test the widget’s state. If a callback is not specified, returns True if the widget state matches statespec and False otherwise. If callback is specified then it is called with args if widget state matches statespec.

state(statespec=None)
    Modify or inquire widget state. If statespec is specified, sets the widget state according to it and return a new statespec indicating which flags were changed. If statespec is not specified, returns the currently-enabled state flags.

    statespec will usually be a list or a tuple.

25.2.4 Combobox

The ttk.Combobox widget combines a text field with a pop-down list of values. This widget is a subclass of Entry.

Besides the methods inherited from Widget: Widget.cget(), Widget.configure(), Widget.identify(), Widget.instate() and Widget.state(), and the following inherited from Entry: Entry.bbox(), Entry.delete(), Entry.icursor(), Entry.index(), Entry.inset(), Entry.selection(), Entry.xview(), it has some other methods, described at ttk.Combobox.

Options

This widget accepts the following specific options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exportselection</td>
<td>Boolean value. If set, the widget selection is linked to the Window Manager selection (which can be returned by invoking Misc.selection_get, for example).</td>
</tr>
<tr>
<td>justify</td>
<td>Specifies how the text is aligned within the widget. One of “left”, “center”, or “right”.</td>
</tr>
<tr>
<td>height</td>
<td>Specifies the height of the pop-down listbox, in rows.</td>
</tr>
<tr>
<td>postcommand</td>
<td>A script (possibly registered with Misc.register) that is called immediately before displaying the values. It may specify which values to display.</td>
</tr>
<tr>
<td>state</td>
<td>One of “normal”, “readonly”, or “disabled”. In the “readonly” state, the value may not be edited directly, and the user can only selection of the values from the dropdown list. In the “normal” state, the text field is directly editable. In the “disabled” state, no interaction is possible.</td>
</tr>
<tr>
<td>textvariable</td>
<td>Specifies a name whose value is linked to the widget value. Whenever the value associated with that name changes, the widget value is updated, and vice versa. See tkinter.StringVar.</td>
</tr>
<tr>
<td>values</td>
<td>Specifies the list of values to display in the drop-down listbox.</td>
</tr>
<tr>
<td>width</td>
<td>Specifies an integer value indicating the desired width of the entry window, in average-size characters of the widget’s font.</td>
</tr>
</tbody>
</table>

Virtual events

The combobox widgets generates a <<ComboboxSelected>> virtual event when the user selects an element from the list of values.
tkinter.ttk module provides a Tk themed widget set, which is a sub-module of the tkinter module. It includes many of the same widgets as tkinter, but with a more modern, cleaner, and more consistent interface.

### 25.2.5 Notebook

Ttk Notebook widget manages a collection of windows and displays a single one at a time. Each child window is associated with a tab, which the user may select to change the currently-displayed window.

#### Options

This widget accepts the following specific options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>height</td>
<td>If present and greater than zero, specifies the desired height of the pane area (not including internal padding or tabs). Otherwise, the maximum height of all panes is used.</td>
</tr>
<tr>
<td>padding</td>
<td>Specifies the amount of extra space to add around the outside of the notebook. The padding is a list up to four length specifications left top right bottom. If fewer than four elements are specified, bottom defaults to top, right defaults to left, and top defaults to left.</td>
</tr>
<tr>
<td>width</td>
<td>If present and greater than zero, specified the desired width of the pane area (not including internal padding). Otherwise, the maximum width of all panes is used.</td>
</tr>
</tbody>
</table>

#### Tab Options

There are also specific options for tabs:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>Either “normal”, “disabled” or “hidden”. If “disabled”, then the tab is not selectable. If “hidden”, then the tab is not shown.</td>
</tr>
<tr>
<td>sticky</td>
<td>Specifies how the child window is positioned within the pane area. Value is a string containing zero or more of the characters “n”, “s”, “e” or “w”. Each letter refers to a side (north, south, east or west) that the child window will stick to, as per the grid() geometry manager.</td>
</tr>
<tr>
<td>padding</td>
<td>Specifies the amount of extra space to add between the notebook and this pane. Syntax is the same as for the option padding used by this widget.</td>
</tr>
<tr>
<td>text</td>
<td>Specifies a text to be displayed in the tab. Syntax is the same as for the option padding used by this widget.</td>
</tr>
<tr>
<td>image</td>
<td>Specifies an image to display in the tab. See the option image described in Widget.</td>
</tr>
<tr>
<td>compound</td>
<td>Specifies how to display the image relative to the text, in the case both options text and image are present. See Label Options for legal values.</td>
</tr>
<tr>
<td>underline</td>
<td>Specifies the index (0-based) of a character to underline in the text string. The underlined character is used for mnemonic activation if Notebook.enable_traversal() is called.</td>
</tr>
</tbody>
</table>
Tab Identifiers

The `tab_id` present in several methods of `ttk.Notebook` may take any of the following forms:

- An integer between zero and the number of tabs
- The name of a child window
- A positional specification of the form “@x,y”, which identifies the tab
- The literal string “current”, which identifies the currently-selected tab
- The literal string “end”, which returns the number of tabs (only valid for `Notebook.index()`)

Virtual Events

This widget generates a `<<NotebookTabChanged>>` virtual event after a new tab is selected.

ttk.Notebook

```python
class tkinter.ttk.Notebook
```

- `add(child, **kw)`
  Adds a new tab to the notebook.
  If window is currently managed by the notebook but hidden, it is restored to its previous position.
  See Tab Options for the list of available options.

- `forget(tab_id)`
  Removes the tab specified by `tab_id`, unmaps and unmanages the associated window.

- `hide(tab_id)`
  Hides the tab specified by `tab_id`.
  The tab will not be displayed, but the associated window remains managed by the notebook and its configuration remembered. Hidden tabs may be restored with the `add()` command.

- `identify(x, y)`
  Returns the name of the tab element at position `x, y`, or the empty string if none.

- `index(tab_id)`
  Returns the numeric index of the tab specified by `tab_id`, or the total number of tabs if `tab_id` is the string “end”.

- `insert(pos, child, **kw)`
  Inserts a pane at the specified position.
  `pos` is either the string “end”, an integer index, or the name of a managed child. If `child` is already managed by the notebook, moves it to the specified position.
  See Tab Options for the list of available options.

- `select(tab_id=None)`
  Selects the specified `tab_id`.
  The associated child window will be displayed, and the previously-selected window (if different) is unmapped. If `tab_id` is omitted, returns the widget name of the currently selected pane.

- `tab(tab_id, option=None, **kw)`
  Query or modify the options of the specific `tab_id`.
  If `kw` is not given, returns a dictionary of the tab option values. If `option` is specified, returns the value of that `option`. Otherwise, sets the options to the corresponding values.
tabs()  
Returns a list of windows managed by the notebook.

enable_traversal()  
Enable keyboard traversal for a toplevel window containing this notebook.

This will extend the bindings for the toplevel window containing the notebook as follows:

- Control-Tab: selects the tab following the currently selected one.
- Shift-Control-Tab: selects the tab preceding the currently selected one.
- Alt-K: where K is the mnemonic (underlined) character of any tab, will select that tab.

Multiple notebooks in a single toplevel may be enabled for traversal, including nested notebooks. However, notebook traversal only works properly if all panes have the notebook they are in as master.

25.2.6 Progressbar

The ttk.Progressbar widget shows the status of a long-running operation. It can operate in two modes: 1) the determinate mode which shows the amount completed relative to the total amount of work to be done and 2) the indeterminate mode which provides an animated display to let the user know that work is progressing.

Options

This widget accepts the following specific options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orient</td>
<td>One of “horizontal” or “vertical”. Specifies the orientation of the progress bar.</td>
</tr>
<tr>
<td>length</td>
<td>Specifies the length of the long axis of the progress bar (width if horizontal, height if vertical).</td>
</tr>
<tr>
<td>mode</td>
<td>One of “determinate” or “indeterminate”.</td>
</tr>
<tr>
<td>maximum</td>
<td>A number specifying the maximum value. Defaults to 100.</td>
</tr>
<tr>
<td>value</td>
<td>The current value of the progress bar. In “determinate” mode, this represents the amount of work completed. In “indeterminate” mode, it is interpreted as modulo maximum; that is, the progress bar completes one “cycle” when its value increases by maximum.</td>
</tr>
<tr>
<td>variable</td>
<td>A name which is linked to the option value. If specified, the value of the progress bar is automatically set to the value of this name whenever the latter is modified.</td>
</tr>
<tr>
<td>phase</td>
<td>Read-only option. The widget periodically increments the value of this option whenever its value is greater than 0 and, in determinate mode, less than maximum. This option may be used by the current theme to provide additional animation effects.</td>
</tr>
</tbody>
</table>

ttk.Progressbar

class tkinter.ttk.Progressbar

    start (interval=None)  
    Begin autoincrement mode: schedules a recurring timer event that calls Progressbar.step() every interval milliseconds. If omitted, interval defaults to 50 milliseconds.

    step (amount=None)  
    Increments the progress bar’s value by amount.  
    amount defaults to 1.0 if omitted.

    stop ()  
    Stop autoincrement mode: cancels any recurring timer event initiated by Progressbar.start() for this progress bar.
25.2.7 Separator

The `ttk.Separator` widget displays a horizontal or vertical separator bar. It has no other methods besides the ones inherited from `ttk.Widget`.

Options

This widget accepts the following specific option:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orient</td>
<td>One of “horizontal” or “vertical”. Specifies the orientation of the separator.</td>
</tr>
</tbody>
</table>

25.2.8 Sizegrip

The `ttk.Sizegrip` widget (also known as a grow box) allows the user to resize the containing toplevel window by pressing and dragging the grip. This widget has neither specific options nor specific methods, besides the ones inherited from `ttk.Widget`.

Platform-specific notes

- On MacOS X, toplevel windows automatically include a built-in size grip by default. Adding a `Sizegrip` is harmless, since the built-in grip will just mask the widget.

Bugs

- If the containing toplevel’s position was specified relative to the right or bottom of the screen (e.g. ....), the `Sizegrip` widget will not resize the window.
- This widget supports only “southeast” resizing.

25.2.9 Treeview

The `ttk.Treeview` widget displays a hierarchical collection of items. Each item has a textual label, an optional image, and an optional list of data values. The data values are displayed in successive columns after the tree label. The order in which data values are displayed may be controlled by setting the widget option `displaycolumns`. The tree widget can also display column headings. Columns may be accessed by number or symbolic names listed in the widget option columns. See Column Identifiers.

Each item is identified by an unique name. The widget will generate item IDs if they are not supplied by the caller. There is a distinguished root item, named `{}`. The root item itself is not displayed; its children appear at the top level of the hierarchy.

Each item also has a list of tags, which can be used to associate event bindings with individual items and control the appearance of the item.

The `Treeview` widget supports horizontal and vertical scrolling, according to the options described in Scrollable Widget Options and the methods `Treeview.xview()` and `Treeview.yview()`.

Options

This widget accepts the following specific options:
### Option Description

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>columns</td>
<td>A list of column identifiers, specifying the number of columns and their names.</td>
</tr>
<tr>
<td>displaycolumns</td>
<td>A list of column identifiers (either symbolic or integer indices) specifying which data columns are displayed and the order in which they appear, or the string &quot;#all&quot;.</td>
</tr>
<tr>
<td>height</td>
<td>Specifies the number of rows which should be visible. Note: the requested width is determined from the sum of the column widths.</td>
</tr>
<tr>
<td>padding</td>
<td>Specifies the internal padding for the widget. The padding is a list of up to four length specifications.</td>
</tr>
<tr>
<td>selectmode</td>
<td>Controls how the built-in class bindings manage the selection. One of &quot;extended&quot;, &quot;browse&quot; or &quot;none&quot;. If set to &quot;extended&quot; (the default), multiple items may be selected. If &quot;browse&quot;, only a single item will be selected at a time. If &quot;none&quot;, the selection will not be changed. Note that the application code and tag bindings can set the selection however they wish, regardless of the value of this option.</td>
</tr>
</tbody>
</table>
| show           | A list containing zero or more of the following values, specifying which elements of the tree to display.  

  - tree: display tree labels in column #0.  
  - headings: display the heading row.  

  The default is "tree headings", i.e., show all elements.  

  **Note:** Column #0 always refers to the tree column, even if show="tree" is not specified. |

### Item Options

The following item options may be specified for items in the insert and item widget commands.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>The textual label to display for the item.</td>
</tr>
<tr>
<td>image</td>
<td>A Tk Image, displayed to the left of the label.</td>
</tr>
<tr>
<td>values</td>
<td>The list of values associated with the item. Each item should have the same number of values as the widget option columns. If there are fewer values than columns, the remaining values are assumed empty. If there are more values than columns, the extra values are ignored.</td>
</tr>
<tr>
<td>open</td>
<td>True/False value indicating whether the item’s children should be displayed or hidden.</td>
</tr>
<tr>
<td>tags</td>
<td>A list of tags associated with this item.</td>
</tr>
</tbody>
</table>

### Tag Options

The following options may be specified on tags:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>foreground</td>
<td>Specifies the text foreground color.</td>
</tr>
<tr>
<td>background</td>
<td>Specifies the cell or item background color.</td>
</tr>
<tr>
<td>font</td>
<td>Specifies the font to use when drawing text.</td>
</tr>
<tr>
<td>image</td>
<td>Specifies the item image, in case the item’s image option is empty.</td>
</tr>
</tbody>
</table>

### Column Identifiers

Column identifiers take any of the following forms:

- A symbolic name from the list of columns option.
- An integer n, specifying the nth data column.
- A string of the form #n, where n is an integer, specifying the nth display column.
Notes:

- Item’s option values may be displayed in a different order than the order in which they are stored.
- Column #0 always refers to the tree column, even if show=”tree” is not specified.

A data column number is an index into an item’s option values list; a display column number is the column number in the tree where the values are displayed. Tree labels are displayed in column #0. If option displaycolumns is not set, then data column n is displayed in column #n+1. Again, column #0 always refers to the tree column.

Virtual Events

The Treeview widget generates the following virtual events.

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;TreeviewSelect&gt;&gt;</td>
<td>Generated whenever the selection changes.</td>
</tr>
<tr>
<td>&lt;&lt;TreeviewOpen&gt;&gt;</td>
<td>Generated just before setting the focus item to open=True.</td>
</tr>
<tr>
<td>&lt;&lt;TreeviewClose&gt;&gt;</td>
<td>Generated just after setting the focus item to open=False.</td>
</tr>
</tbody>
</table>

The Treeview.focus() and Treeview.selection() methods can be used to determine the affected item or items.

ttk.Treeview

class tkinter.ttk.Treeview

bbox (item, column=None)  
Returns the bounding box (relative to the treeview widget’s window) of the specified item in the form (x, y, width, height).

If column is specified, returns the bounding box of that cell. If the item is not visible (i.e., if it is a descendant of a closed item or is scrolled offscreen), returns an empty string.

get_children (item=None)  
Returns the list of children belonging to item.

If item is not specified, returns root children.

set_children (item, *newchildren)  
Replaces item’s child with newchildren.

Children present in item that are not present in newchildren are detached from the tree. No items in newchildren may be an ancestor of item. Note that not specifying newchildren results in detaching item’s children.

column (column, option=None, **kw)  
Query or modify the options for the specified column.

If kw is not given, returns a dict of the column option values. If option is specified then the value for that option is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

- **id** Returns the column name. This is a read-only option.
- **anchor:** One of the standard Tk anchor values. Specifies how the text in this column should be aligned with respect to the cell.
- **minwidth:** width The minimum width of the column in pixels. The treeview widget will not make the column any smaller than specified by this option when the widget is resized or the user drags a column.
- **stretch:** True/False Specifies whether the column’s width should be adjusted when the widget is resized.
•**width**: width  The width of the column in pixels.

To configure the tree column, call this with column = “#0”

**delete** (*items*)
Delete all specified *items* and all their descendants.

The root item may not be deleted.

**detach** (*items*)
Unlinks all of the specified *items* from the tree.

The items and all of their descendants are still present, and may be reinserted at another point in the tree, but will not be displayed.

The root item may not be detached.

**exists** (*item*)
Returns True if the specified *item* is present in the tree.

**focus** (*item=None*)
If *item* is specified, sets the focus item to *item*. Otherwise, returns the current focus item, or ’’ if there is none.

**heading** (*column, option=None, **kw*)
Query or modify the heading options for the specified *column*.

If *kw* is not given, returns a dict of the heading option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

•**text**: text  The text to display in the column heading.

•**image**: imageName  Specifies an image to display to the right of the column heading.

•**anchor**: anchor  Specifies how the heading text should be aligned. One of the standard Tk anchor values.

•**command**: callback  A callback to be invoked when the heading label is pressed.

To configure the tree column heading, call this with column = “#0”.

**identify** (*component, x, y*)
Returns a description of the specified *component* under the point given by *x* and *y*, or the empty string if no such *component* is present at that position.

**identify_row** (*y*)
Returns the item ID of the item at position *y*.

**identify_column** (*x*)
Returns the data column identifier of the cell at position *x*.

The tree column has ID #0.

**identify_region** (*x, y*)
Returns one of:

<table>
<thead>
<tr>
<th>region</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>heading</td>
<td>Tree heading area.</td>
</tr>
<tr>
<td>separator</td>
<td>Space between two columns headings.</td>
</tr>
<tr>
<td>tree</td>
<td>The tree area.</td>
</tr>
<tr>
<td>cell</td>
<td>A data cell.</td>
</tr>
</tbody>
</table>

Availability: Tk 8.6.

**identify_element** (*x, y*)
Returns the element at position *x, y*.

Availability: Tk 8.6.
index (item)
Returns the integer index of item within its parent’s list of children.

insert (parent, index, iid=None, **kw)
Creates a new item and returns the item identifier of the newly created item.

parent is the item ID of the parent item, or the empty string to create a new top-level item. index is an integer, or the value “end”, specifying where in the list of parent’s children to insert the new item. If index is less than or equal to zero, the new node is inserted at the beginning; if index is greater than or equal to the current number of children, it is inserted at the end. If iid is specified, it is used as the item identifier; iid must not already exist in the tree. Otherwise, a new unique identifier is generated.

See Item Options for the list of available points.

item (item, option=None, **kw)
Query or modify the options for the specified item.

If no options are given, a dict with options/values for the item is returned. If option is specified then the value for that option is returned. Otherwise, sets the options to the corresponding values as given by kw.

move (item, parent, index)
Moves item to position index in parent’s list of children.

It is illegal to move an item under one of its descendants. If index is less than or equal to zero, item is moved to the beginning; if greater than or equal to the number of children, it is moved to the end. If item was detached it is reattached.

next (item)
Returns the identifier of item’s next sibling, or ‘’ if item is the last child of its parent.

parent (item)
Returns the ID of the parent of item, or ‘’ if item is at the top level of the hierarchy.

prev (item)
Returns the identifier of item’s previous sibling, or ‘’ if item is the first child of its parent.

reattach (item, parent, index)
An alias for Treeview.move().

see (item)
Ensure that item is visible.

Sets all of item’s ancestors open option to True, and scrolls the widget if necessary so that item is within the visible portion of the tree.

selection (selop=None, items=None)
If selop is not specified, returns selected items. Otherwise, it will act according to the following selection methods.

selection_set (items)
items becomes the new selection.

selection_add (items)
Add items to the selection.

selection_remove (items)
Remove items from the selection.

selection_toggle (items)
Toggle the selection state of each item in items.

set (item, column=None, value=None)
With one argument, returns a dictionary of column/value pairs for the specified item. With two arguments, returns the current value of the specified column. With three arguments, sets the value of given column in given item to the specified value.
tag_bind(tagname, sequence=None, callback=None)
Bind a callback for the given event sequence to the tag tagname. When an event is delivered to an item, the callbacks for each of the item’s tags option are called.

tag_configure(tagname, option=None, **kw)
Query or modify the options for the specified tagname.

If kw is not given, returns a dict of the option settings for tagname. If option is specified, returns the value for that option for the specified tagname. Otherwise, sets the options to the corresponding values for the given tagname.

tag_has(tagname, item=None)
If item is specified, returns 1 or 0 depending on whether the specified item has the given tagname. Otherwise, returns a list of all items that have the specified tag.

Availability: Tk 8.6

xview(*args)
Query or modify horizontal position of the treeview.

yview(*args)
Query or modify vertical position of the treeview.

25.2.10 Tk Styling

Each widget in ttk is assigned a style, which specifies the set of elements making up the widget and how they are arranged, along with dynamic and default settings for element options. By default the style name is the same as the widget’s class name, but it may be overridden by the widget’s style option. If you don’t know the class name of a widget, use the method Misc.winfo_class() (somewidget.winfo_class()).

See Also:
Tcl’2004 conference presentation This document explains how the theme engine works

class tkinter.ttk.Style
This class is used to manipulate the style database.

configure(style, query_opt=None, **kw)
Query or set the default value of the specified option(s) in style.

Each key in kw is an option and each value is a string identifying the value for that option.

For example, to change every default button to be a flat button with some padding and a different background color:

```python
from tkinter import ttk
import tkinter

root = tkinter.Tk()

ttk.Style().configure("TButton", padding=6, relief="flat", background="#ccc")

btn = ttk.Button(text="Sample")
btn.pack()

root.mainloop()
```

map(style, query_opt=None, **kw)
Query or sets dynamic values of the specified option(s) in style.

Each key in kw is an option and each value should be a list or a tuple (usually) containing statespecs grouped in tuples, lists, or some other preference. A statespec is a compound of one or more states and then a value.
import tkinter
from tkinter import ttk

root = tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=[('pressed', 'red'), ('active', 'blue')],
    background=[('pressed', '!disabled', 'black'), ('active', 'white')]
)

colored_btn = ttk.Button(text="Test", style="C.TButton").pack()

root.mainloop()

Note that the order of the (states, value) sequences for an option does matter, if the order is changed
to ["active", "blue"], ["pressed", "red"] in the foreground option, for example,
the result would be a blue foreground when the widget were in active or pressed states.

lookup(style, option, state=None, default=None)
Returns the value specified for option in style.
If state is specified, it is expected to be a sequence of one or more states. If the default argument is set,
it is used as a fallback value in case no specification for option is found.

To check what font a Button uses by default:

from tkinter import ttk

print(ttk.Style().lookup("TButton", "font"))

layout(style, layoutspec=None)
Define the widget layout for given style. If layoutspec is omitted, return the layout specification for
given style.
layoutspec, if specified, is expected to be a list or some other sequence type (excluding strings), where
each item should be a tuple and the first item is the layout name and the second item should have the
format described in Layouts.

To understand the format, see the following example (it is not intended to do anything useful):

from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.layout("TMenuButton", [
    ("Menubutton.background", None),
    ("Menubutton.button", {"children":
        ("Menubutton.focus", {"children":
            ("Menubutton.padding", {"children":
                ("Menubutton.label", {"side": "left", "expand": 1})
            )
        )
    )
})
])

mbtn = ttk.MenuButton(text='Text')
mbtn.pack()
root.mainloop()

**element_create**(elementname, etype, *args, **kw)

Create a new element in the current theme, of the given etype which is expected to be either “image”, “from” or “vsapi”. The latter is only available in Tk 8.6a for Windows XP and Vista and is not described here.

If “image” is used, args should contain the default image name followed by statespec/value pairs (this is the imagespec), and kw may have the following options:

- **border=padding**  padding is a list of up to four integers, specifying the left, top, right, and bottom borders, respectively.
- **height=height**  Specifies a minimum height for the element. If less than zero, the base image’s height is used as a default.
- **padding=padding**  Specifies the element’s interior padding. Defaults to border’s value if not specified.
- **sticky=spec**  Specifies how the image is placed within the final parcel. spec contains zero or more characters “n”, “s”, “w”, or “e”.
- **width=width**  Specifies a minimum width for the element. If less than zero, the base image’s width is used as a default.

If “from” is used as the value of etype, element_create() will clone an existing element. args is expected to contain a themename, from which the element will be cloned, and optionally an element to clone from. If this element to clone from is not specified, an empty element will be used. kw is discarded.

**element_names()**

Returns the list of elements defined in the current theme.

**element_options**(elementname)

Returns the list of elementname’s options.

**theme_create**(themename, parent=None, settings=None)

Create a new theme.

It is an error if themename already exists. If parent is specified, the new theme will inherit styles, elements and layouts from the parent theme. If settings are present they are expected to have the same syntax used for theme_settings().

**theme_settings**(themename, settings)

Temporarily sets the current theme to themename, apply specified settings and then restore the previous theme.

Each key in settings is a style and each value may contain the keys ‘configure’, ‘map’, ‘layout’ and ‘element create’ and they are expected to have the same format as specified by the methods Style.configure(), Style.map(), Style.layout() and Style.element_create() respectively.

As an example, let’s change the Combobox for the default theme a bit:

```python
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
    "TCombobox": {
        "configure": {"padding": 5},
        "map": {
```
```
"background": [{"active": "green2"},
              {"!disabled": "green4"}],
"fieldbackground": [{"!disabled": "green3"}],
"foreground": [{"focus": "OliveDrab1"},
               {"!disabled": "OliveDrab2"}]
}
```

```python
combo = ttk.Combobox().pack()

root.mainloop()
```

**theme_names()**

Returns a list of all known themes.

**theme_use(themename=None)**

If `themename` is not given, returns the theme in use. Otherwise, sets the current theme to `themename`, refreshes all widgets and emits a `<<ThemeChanged>>` event.

### Layouts

A layout can be just None, if it takes no options, or a dict of options specifying how to arrange the element. The layout mechanism uses a simplified version of the pack geometry manager: given an initial cavity, each element is allocated a parcel. Valid options/values are:

- **side**: `whichside` Specifies which side of the cavity to place the element; one of top, right, bottom or left. If omitted, the element occupies the entire cavity.
- **sticky**: `nswe` Specifies where the element is placed inside its allocated parcel.
- **unit**: 0 or 1 If set to 1, causes the element and all of its descendants to be treated as a single element for the purposes of `Widget.identify()` et al. It’s used for things like scrollbar thumbs with grips.
- **children**: `[sublayout... ]` Specifies a list of elements to place inside the element. Each element is a tuple (or other sequence type) where the first item is the layout name, and the other is a `Layout`.

### 25.3 tkinter.tix — Extension widgets for Tk

The `tkinter.tix` (Tk Interface Extension) module provides an additional rich set of widgets. Although the standard Tk library has many useful widgets, they are far from complete. The `tkinter.tix` library provides most of the commonly needed widgets that are missing from standard Tk: `HList, ComboBox, Control` (a.k.a. `SpinBox`) and an assortment of scrollable widgets. `tkinter.tix` also includes many more widgets that are generally useful in a wide range of applications: `NoteBook, FileEntry, PanedWindow`, etc; there are more than 40 of them.

With all these new widgets, you can introduce new interaction techniques into applications, creating more useful and more intuitive user interfaces. You can design your application by choosing the most appropriate widgets to match the special needs of your application and users.

**See Also:**

- **Tix Homepage** The home page for Tix. This includes links to additional documentation and downloads.
- **Tix Man Pages** On-line version of the man pages and reference material.
- **Tix Programming Guide** On-line version of the programmer’s reference material.
- **Tix Development Applications** Tix applications for development of Tix and Tkinter programs. Tide applications work under Tk or Tkinter, and include **TixInspect**, an inspector to remotely modify and debug Tix/Tk/Tkinter applications.
25.3.1 Using Tix

```python
class tkinter.tix.Tk(screenName=None, baseName=None, className='Tix')
    Toplevel widget of Tix which represents mostly the main window of an application. It has an associated Tcl interpreter.
```

Classes in the `tkinter.tix` module subclasses the classes in the `tkinter`. The former imports the latter, so to use `tkinter.tix` with Tkinter, all you need to do is to import one module. In general, you can just import `tkinter.tix`, and replace the toplevel call to `tkinter.Tk` with `tix.Tk`:

```python
from tkinter import tix
from tkinter.constants import *
root = tix.Tk()
```

To use `tkinter.tix`, you must have the Tix widgets installed, usually alongside your installation of the Tk widgets. To test your installation, try the following:

```python
from tkinter import tix
root = tix.Tk()
root.tk.eval('package require Tix')
```

If this fails, you have a Tk installation problem which must be resolved before proceeding. Use the environment variable `TIX_LIBRARY` to point to the installed Tix library directory, and make sure you have the dynamic object library (`tix8183.dll` or `libtix8183.so`) in the same directory that contains your Tk dynamic object library (`tk8183.dll` or `libtk8183.so`). The directory with the dynamic object library should also have a file called `pkgIndex.tcl` (case sensitive), which contains the line:

```bash
package ifneeded Tix 8.1 [list load "[file join $dir tix8183.dll]" Tix]
```

25.3.2 Tix Widgets

Tix introduces over 40 widget classes to the `tkinter` repertoire.

**Basic Widgets**

```python
class tkinter.tix.Balloon
    A Balloon that pops up over a widget to provide help. When the user moves the cursor inside a widget to which a Balloon widget has been bound, a small pop-up window with a descriptive message will be shown on the screen.

class tkinter.tix.ButtonBox
    The ButtonBox widget creates a box of buttons, such as is commonly used for Ok Cancel.

class tkinter.tix.ComboBox
    The ComboBox widget is similar to the combo box control in MS Windows. The user can select a choice by either typing in the entry subwidget or selecting from the listbox subwidget.

class tkinter.tix.Control
    The Control widget is also known as the SpinBox widget. The user can adjust the value by pressing the two arrow buttons or by entering the value directly into the entry. The new value will be checked against the user-defined upper and lower limits.

class tkinter.tix.LabelEntry
    The LabelEntry widget packages an entry widget and a label into one mega widget. It can be used be used to simplify the creation of “entry-form” type of interface.

class tkinter.tix.LabelFrame
    The LabelFrame widget packages a frame widget and a label into one mega widget. To create widgets inside a LabelFrame widget, one creates the new widgets relative to the frame subwidget and manage them inside the frame subwidget.
```
class `tkinter.tix.Meter`
   The `Meter` widget can be used to show the progress of a background job which may take a long time to execute.

class `tkinter.tix.OptionMenu`
   The `OptionMenu` creates a menu button of options.

class `tkinter.tix.PopupMenu`
   The `PopupMenu` widget can be used as a replacement of the `tk_popup` command. The advantage of the `Tix PopupMenu` widget is it requires less application code to manipulate.

class `tkinter.tix.Select`
   The `Select` widget is a container of button subwidgets. It can be used to provide radio-box or check-box style of selection options for the user.

class `tkinter.tix.StdButtonBox`
   The `StdButtonBox` widget is a group of standard buttons for Motif-like dialog boxes.

**File Selectors**

class `tkinter.tix.DirList`
   The `DirList` widget displays a list view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class `tkinter.tix.DirTree`
   The `DirTree` widget displays a tree view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class `tkinter.tix.DirSelectDialog`
   The `DirSelectDialog` widget presents the directories in the file system in a dialog window. The user can use this dialog window to navigate through the file system to select the desired directory.

class `tkinter.tix.DirSelectBox`
   The `DirSelectBox` is similar to the standard Motif(TM) directory-selection box. It is generally used for the user to choose a directory. `DirSelectBox` stores the directories mostly recently selected into a `ComboBox` widget so that they can be quickly selected again.

class `tkinter.tix.ExFileSelectBox`
   The `ExFileSelectBox` widget is usually embedded in a `tixExFileSelectDialog` widget. It provides an convenient method for the user to select files. The style of the `ExFileSelectBox` widget is very similar to the standard file dialog on MS Windows 3.1.

class `tkinter.tix.FileSelectBox`
   The `FileSelectBox` is similar to the standard Motif(TM) file-selection box. It is generally used for the user to choose a file. `FileSelectBox` stores the files mostly recently selected into a `ComboBox` widget so that they can be quickly selected again.

class `tkinter.tix.FileEntry`
   The `FileEntry` widget can be used to input a filename. The user can type in the filename manually. Alternatively, the user can press the button widget that sits next to the entry, which will bring up a file selection dialog.

**Hierarchical ListBox**

class `tkinter.tix.HList`
   The `HList` widget can be used to display any data that have a hierarchical structure, for example, file system directory trees. The list entries are indented and connected by branch lines according to their places in the hierarchy.

class `tkinter.tix.CheckList`
   The `CheckList` widget displays a list of items to be selected by the user. `CheckList` acts similarly to the Tk
checkbutton or radiobutton widgets, except it is capable of handling many more items than checkbuttons or radiobuttons.

```python
class tkinter.tix.Tree
    The Tree widget can be used to display hierarchical data in a tree form. The user can adjust the view of the tree by opening or closing parts of the tree.
```

Tabular ListBox

```python
class tkinter.tix.TList
    The TList widget can be used to display data in a tabular format. The list entries of a TList widget are similar to the entries in the Tk listbox widget. The main differences are (1) the TList widget can display the list entries in a two dimensional format and (2) you can use graphical images as well as multiple colors and fonts for the list entries.
```

Manager Widgets

```python
class tkinter.tix.PanedWindow
    The PanedWindow widget allows the user to interactively manipulate the sizes of several panes. The panes can be arranged either vertically or horizontally. The user changes the sizes of the panes by dragging the resize handle between two panes.

class tkinter.tix.ListNoteBook
    The ListNoteBook widget is very similar to the TixNoteBook widget: it can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages (windows). At one time only one of these pages can be shown. The user can navigate through these pages by choosing the name of the desired page in the hlist subwidget.

class tkinter.tix.NoteBook
    The NoteBook widget can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages. At one time only one of these pages can be shown. The user can navigate through these pages by choosing the visual “tabs” at the top of the NoteBook widget.
```

Image Types

The `tkinter.tix` module adds:

- **Pixmap** capabilities to all `tkinter.tix` and `tkinter` widgets to create color images from XPM files.
- **Compound** image types can be used to create images that consists of multiple horizontal lines; each line is composed of a series of items (texts, bitmaps, images or spaces) arranged from left to right. For example, a compound image can be used to display a bitmap and a text string simultaneously in a Tk Button widget.

Miscellaneous Widgets

```python
class tkinter.tix.InputOnly
    The InputOnly widgets are to accept inputs from the user, which can be done with the bind command (Unix only).
```

Form Geometry Manager

In addition, `tkinter.tix` augments `tkinter` by providing:

```python
class tkinter.tix.Form
    The Form geometry manager based on attachment rules for all Tk widgets.
```
25.3.3 Tix Commands

class tkinter.tix.tixCommand

The tix commands provide access to miscellaneous elements of Tix’s internal state and the Tix application context. Most of the information manipulated by these methods pertains to the application as a whole, or to a screen or display, rather than to a particular window.

To view the current settings, the common usage is:

```python
from tkinter import tix
root = tix.Tk()
print(root.tix_configure())
```

tixCommand.tix_configure(cnf=None, **kw)

Query or modify the configuration options of the Tix application context. If no option is specified, returns a dictionary all of the available options. If option is specified with no value, then the method returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the method modifies the given option(s) to have the given value(s); in this case the method returns an empty string. Option may be any of the configuration options.

tixCommand.tix_cget(option)

Returns the current value of the configuration option given by option. Option may be any of the configuration options.

tixCommand.tix_getbitmap(name)

Locates a bitmap file of the name name.xpm or name in one of the bitmap directories (see the tix_addbitmapdir() method). By using tix_getbitmap(), you can avoid hard coding the pathnames of the bitmap files in your application. When successful, it returns the complete pathname of the bitmap file, prefixed with the character @. The returned value can be used to configure the bitmap option of the Tk and Tix widgets.

tixCommand.tix_addbitmapdir(directory)

Tix maintains a list of directories under which the tix_getimage() and tix_getbitmap() methods will search for image files. The standard bitmap directory is $TIX_LIBRARY/bitmaps. The tix_addbitmapdir() method adds directory into this list. By using this method, the image files of an applications can also be located using the tix_getimage() or tix_getbitmap() method.

tixCommand.tix_filedialog([dlgclass])

Returns the file selection dialog that may be shared among different calls from this application. This method will create a file selection dialog widget when it is called the first time. This dialog will be returned by all subsequent calls to tix_filedialog(). An optional dlgclass parameter can be passed as a string to specified what type of file selection dialog widget is desired. Possible options are tix, FileSelectDialog or tixExFileSelectDialog.

tixCommand.tix_getimage(self, name)

Locates an image file of the name name.xpm, name.xbm or name.ppm in one of the bitmap directories (see the tix_addbitmapdir() method above). If more than one file with the same name (but different extensions) exist, then the image type is chosen according to the depth of the X display: xbm images are chosen on monochrome displays and color images are chosen on color displays. By using tix_getimage(), you can avoid hard coding the pathnames of the image files in your application. When successful, this method returns the name of the newly created image, which can be used to configure the image option of the Tk and Tix widgets.

tixCommand.tix_option_get(name)

Gets the options maintained by the Tix scheme mechanism.

tixCommand.tix_resetoptions(newScheme, newFontSet, newScmPrio)

Resets the scheme and fontset of the Tix application to newScheme and newFontSet, respectively. This affects only those widgets created after this call. Therefore, it is best to call the resetoptions method before the creation of any widgets in a Tix application.
The optional parameter newScmPrio can be given to reset the priority level of the Tk options set by the Tix schemes.

Because of the way Tk handles the X option database, after Tix has been has imported and initialized, it is not possible to reset the color schemes and font sets using the tix_config() method. Instead, the tix_resetoptions() method must be used.

25.4 tkinter.scrolledtext — Scrolled Text Widget

Platforms: Tk

The tkinter.scrolledtext module provides a class of the same name which implements a basic text widget which has a vertical scroll bar configured to do the “right thing.” Using the ScrolledText class is a lot easier than setting up a text widget and scroll bar directly. The constructor is the same as that of the tkinter.Text class.

The text widget and scrollbar are packed together in a Frame, and the methods of the Grid and Pack geometry managers are acquired from the Frame object. This allows the ScrolledText widget to be used directly to achieve most normal geometry management behavior.

Should more specific control be necessary, the following attributes are available:

ScrolledText.frame
The frame which surrounds the text and scroll bar widgets.

ScrolledText.vbar
The scroll bar widget.

25.5 IDLE

IDLE is the Python IDE built with the tkinter GUI toolkit.

IDLE has the following features:

• coded in 100% pure Python, using the tkinter GUI toolkit
• cross-platform: works on Windows and Unix
• multi-window text editor with multiple undo, Python colorizing and many other features, e.g. smart indent and call tips
• Python shell window (a.k.a. interactive interpreter)
• debugger (not complete, but you can set breakpoints, view and step)

25.5.1 Menus

File menu

New file create a new file editing window
Open... open an existing file
Open module... open an existing module (searches sys.path)
Class browser show classes and methods in current file
Path browser show sys.path directories, modules, classes and methods
Save save current window to the associated file (unsaved windows have a * before and after the window title)
Save As... save current window to new file, which becomes the associated file
Save Copy As...  save current window to different file without changing the associated file
Close  close current window (asks to save if unsaved)
Exit  close all windows and quit IDLE (asks to save if unsaved)

Edit menu

Undo  Undo last change to current window (max 1000 changes)
Redo  Redo last undone change to current window
Cut  Copy selection into system-wide clipboard; then delete selection
Copy  Copy selection into system-wide clipboard
Paste  Insert system-wide clipboard into window
Select All  Select the entire contents of the edit buffer
Find...  Open a search dialog box with many options
Find again  Repeat last search
Find selection  Search for the string in the selection
Find in Files...  Open a search dialog box for searching files
Replace...  Open a search-and-replace dialog box
Go to line  Ask for a line number and show that line
Indent region  Shift selected lines right 4 spaces
Dedent region  Shift selected lines left 4 spaces
Comment out region  Insert ## in front of selected lines
Uncomment region  Remove leading # or ## from selected lines
Tabify region  Turns leading stretches of spaces into tabs
Untabify region  Turn all tabs into the right number of spaces
Expand word  Expand the word you have typed to match another word in the same buffer; repeat to get a different expansion
Format Paragraph  Reformat the current blank-line-separated paragraph
Import module  Import or reload the current module
Run script  Execute the current file in the __main__ namespace

Windows menu

Zoom Height  toggles the window between normal size (24x80) and maximum height.
The rest of this menu lists the names of all open windows; select one to bring it to the foreground (deiconifying it if necessary).

Debug menu

• in the Python Shell window only

Go to file/line  Look around the insert point for a filename and line number, open the file, and show the line.
Useful to view the source lines referenced in an exception traceback.
Debugger  Run commands in the shell under the debugger.
Stack viewer  Show the stack traceback of the last exception.

Auto-open Stack Viewer  Open stack viewer on traceback.

**Edit context menu**

- Right-click in Edit window (Control-click on OS X)
- Cut  Copy selection into system-wide clipboard; then delete selection
- Copy  Copy selection into system-wide clipboard
- Paste  Insert system-wide clipboard into window
- Set Breakpoint  Sets a breakpoint. Breakpoints are only enabled when the debugger is open.
- Clear Breakpoint  Clears the breakpoint on that line.

**Shell context menu**

- Right-click in Python Shell window (Control-click on OS X)
- Cut  Copy selection into system-wide clipboard; then delete selection
- Copy  Copy selection into system-wide clipboard
- Paste  Insert system-wide clipboard into window
- Go to file/line  Same as in Debug menu.

**25.5.2 Basic editing and navigation**

- Backspace deletes to the left; Del deletes to the right
- Arrow keys and Page Up/Page Down to move around
- Home/End go to begin/end of line
- C-Home/C-End go to begin/end of file
- Some Emacs bindings may also work, including C-B, C-P, C-A, C-E, C-D, C-L

**Automatic indentation**

After a block-opening statement, the next line is indented by 4 spaces (in the Python Shell window by one tab). After certain keywords (break, return etc.) the next line is dedented. In leading indentation, Backspace deletes up to 4 spaces if they are there. Tab inserts 1-4 spaces (in the Python Shell window one tab). See also the indent/dedent region commands in the edit menu.

**Python Shell window**

- C-C interrupts executing command
- C-D sends end-of-file; closes window if typed at a >>> prompt
- Alt-p retrieves previous command matching what you have typed
- Alt-n retrieves next
- Return while on any previous command retrieves that command
- Alt-/ (Expand word) is also useful here.
25.5.3 Syntax colors

The coloring is applied in a background “thread,” so you may occasionally see uncolorized text. To change the color scheme, edit the [Colors] section in config.txt.

Python syntax colors:
- Keywords orange
- Strings green
- Comments red
- Definitions blue

Shell colors:
- Console output brown
- stdout blue
- stderr dark green
- stdin black

25.5.4 Startup

Upon startup with the -s option, IDLE will execute the file referenced by the environment variables IDLESTARTUP or PYTHONSTARTUP. Idle first checks for IDLESTARTUP; if IDLESTARTUP is present the file referenced is run. If IDLESTARTUP is not present, Idle checks for PYTHONSTARTUP. Files referenced by these environment variables are convenient places to store functions that are used frequently from the Idle shell, or for executing import statements to import common modules.

In addition, Tk also loads a startup file if it is present. Note that the Tk file is loaded unconditionally. This additional file is .Idle.py and is looked for in the user’s home directory. Statements in this file will be executed in the Tk namespace, so this file is not useful for importing functions to be used from Idle’s Python shell.

Command line usage

idle.py [-c command] [-d] [-e] [-s] [-t title] [arg] ...

-c command run this command
-d enable debugger
-e edit mode; arguments are files to be edited
-s run $IDLESTARTUP or $PYTHONSTARTUP first
-t title set title of shell window

If there are arguments:

1. If -e is used, arguments are files opened for editing and sys.argv reflects the arguments passed to IDLE itself.
2. Otherwise, if -c is used, all arguments are placed in sys.argv[1:], with sys.argv[0] set to ‘-c’.
3. Otherwise, if neither -e nor -c is used, the first argument is a script which is executed with the remaining arguments in sys.argv[1:] and sys.argv[0] set to the script name. If the script name is '-', no script is executed but an interactive Python session is started; the arguments are still available in sys.argv.

25.6 Other Graphical User Interface Packages

Major cross-platform (Windows, Mac OS X, Unix-like) GUI toolkits are available for Python:
See Also:

PyGObject provides introspection bindings for C libraries using GObject. One of these libraries is the GTK+ 3 widget set. GTK+ comes with many more widgets than Tkinter provides. An online Python GTK+ 3 Tutorial is available.

PyGTK provides bindings for an older version of the library, GTK+ 2. It provides an object oriented interface that is slightly higher level than the C one. There are also bindings to GNOME. An online tutorial is available.

PyQt PyQt is a sip-wrapped binding to the Qt toolkit. Qt is an extensive C++ GUI application development framework that is available for Unix, Windows and Mac OS X. sip is a tool for generating bindings for C++ libraries as Python classes, and is specifically designed for Python. The PyQt3 bindings have a book, GUI Programming with Python: QT Edition by Boudewijn Rempt. The PyQt4 bindings also have a book, Rapid GUI Programming with Python and Qt, by Mark Summerfield.

PySide is a newer binding to the Qt toolkit, provided by Nokia. Compared to PyQt, its licensing scheme is friendlier to non-open source applications.

wxPython wxPython is a cross-platform GUI toolkit for Python that is built around the popular wxWidgets (formerly wxWindows) C++ toolkit. It provides a native look and feel for applications on Windows, Mac OS X, and Unix systems by using each platform’s native widgets where ever possible, (GTK+ on Unix-like systems). In addition to an extensive set of widgets, wxPython provides classes for online documentation and context sensitive help, printing, HTML viewing, low-level device context drawing, drag and drop, system clipboard access, an XML-based resource format and more, including an ever growing library of user-contributed modules. wxPython has a book, wxPython in Action, by Noel Rappin and Robin Dunn.

PyGTK, PyQt, and wxPython, all have a modern look and feel and more widgets than Tkinter. In addition, there are many other GUI toolkits for Python, both cross-platform, and platform-specific. See the GUI Programming page in the Python Wiki for a much more complete list, and also for links to documents where the different GUI toolkits are compared.
The modules described in this chapter help you write software. For example, the `pydoc` module takes a module and generates documentation based on the module’s contents. The `doctest` and `unittest` modules contains frameworks for writing unit tests that automatically exercise code and verify that the expected output is produced. `2to3` can translate Python 2.x source code into valid Python 3.x code.

The list of modules described in this chapter is:

### 26.1 pydoc — Documentation generator and online help system

**Source code:** `Lib/pydoc.py`

The `pydoc` module automatically generates documentation from Python modules. The documentation can be presented as pages of text on the console, served to a Web browser, or saved to HTML files.

The built-in function `help()` invokes the online help system in the interactive interpreter, which uses `pydoc` to generate its documentation as text on the console. The same text documentation can also be viewed from outside the Python interpreter by running `pydoc` as a script at the operating system’s command prompt. For example, running

```
pydoc sys
```

at a shell prompt will display documentation on the `sys` module, in a style similar to the manual pages shown by the Unix `man` command. The argument to `pydoc` can be the name of a function, module, or package, or a dotted reference to a class, method, or function within a module or module in a package. If the argument to `pydoc` looks like a path (that is, it contains the path separator for your operating system, such as a slash in Unix), and refers to an existing Python source file, then documentation is produced for that file.

**Note:** In order to find objects and their documentation, `pydoc` imports the module(s) to be documented. Therefore, any code on module level will be executed on that occasion. Use an `if __name__ == '__main__':` guard to only execute code when a file is invoked as a script and not just imported.

Specifying a `-w` flag before the argument will cause HTML documentation to be written out to a file in the current directory, instead of displaying text on the console.

Specifying a `-k` flag before the argument will search the synopsis lines of all available modules for the keyword given as the argument, again in a manner similar to the Unix `man` command. The synopsis line of a module is the first line of its documentation string.

You can also use `pydoc` to start an HTTP server on the local machine that will serve documentation to visiting Web browsers. `pydoc -p 1234` will start a HTTP server on port 1234, allowing you to browse the documentation at [http://localhost:1234/](http://localhost:1234/) in your preferred Web browser. Specifying 0 as the port number will select an arbitrary unused port.
**26.2 doctest — Test interactive Python examples**

The `doctest` module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown. There are several common ways to use doctest:

- To check that a module’s docstrings are up-to-date by verifying that all interactive examples still work as documented.
- To perform regression testing by verifying that interactive examples from a test file or a test object work as expected.
- To write tutorial documentation for a package, liberally illustrated with input-output examples. Depending on whether the examples or the expository text are emphasized, this has the flavor of “literate testing” or “executable documentation”.

Here’s a complete but small example module:

```python
def factorial(n):
    """Return the factorial of n, an exact integer >= 0."

>>> [factorial(n) for n in range(6)]
[1, 1, 2, 6, 24, 120]
>>> factorial(30)
265252859812191058636308480000000
>>> factorial(-1)
Traceback (most recent call last):
  ... ValueError: n must be >= 0

Factorials of floats are OK, but the float must be an exact integer:
>>> factorial(30.1)
Traceback (most recent call last):
  ... ValueError: n must be exact integer
```

---

**The Python Library Reference, Release 3.3.3**

[72x798]The `pydoc -g` will start the server and additionally bring up a small `tkinter`-based graphical interface to help you search for documentation pages. The `-g` option is deprecated, since the server can now be controlled directly from HTTP clients.

`pydoc -b` will start the server and additionally open a web browser to a module index page. Each served page has a navigation bar at the top where you can Get help on an individual item, Search all modules with a keyword in their synopsis line, and go to the Module index, Topics and Keywords pages.

When `pydoc` generates documentation, it uses the current environment and path to locate modules. Thus, invoking `pydoc spam` documents precisely the version of the module you would get if you started the Python interpreter and typed `import spam`.

Module docs for core modules are assumed to reside in [http://docs.python.org/X.Y/library/](http://docs.python.org/X.Y/library/) where X and Y are the major and minor version numbers of the Python interpreter. This can be overridden by setting the PYTHONDOCS environment variable to a different URL or to a local directory containing the Library Reference Manual pages. Changed in version 3.2: Added the `-b` option, deprecated the `-g` option.

---

26.2 doctest — Test interactive Python examples

The `doctest` module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown. There are several common ways to use doctest:

- To check that a module’s docstrings are up-to-date by verifying that all interactive examples still work as documented.
- To perform regression testing by verifying that interactive examples from a test file or a test object work as expected.
- To write tutorial documentation for a package, liberally illustrated with input-output examples. Depending on whether the examples or the expository text are emphasized, this has the flavor of “literate testing” or “executable documentation”.

Here’s a complete but small example module:

```python
def factorial(n):
    """Return the factorial of n, an exact integer >= 0."

>>> [factorial(n) for n in range(6)]
[1, 1, 2, 6, 24, 120]
>>> factorial(30)
265252859812191058636308480000000
>>> factorial(-1)
Traceback (most recent call last):
  ... ValueError: n must be >= 0

Factorials of floats are OK, but the float must be an exact integer:
>>> factorial(30.1)
Traceback (most recent call last):
  ... ValueError: n must be exact integer
```
>>> factorial(30.0)
265252859812191058636308480000000

It must also not be ridiculously large:
>>> factorial(1e100)
Traceback (most recent call last):
... 
OverflowError: n too large

import math
if not n >= 0:
    raise ValueError("n must be >= 0")
if math.floor(n) != n:
    raise ValueError("n must be exact integer")
if n+1 == n:  # catch a value like 1e300
    raise OverflowError("n too large")
result = 1
factor = 2
while factor <= n:
    result *= factor
    factor += 1
return result

if __name__ == "__main__":
    import doctest
doctest.testmod()

If you run example.py directly from the command line, doctest works its magic:

$ python example.py
$

There’s no output! That’s normal, and it means all the examples worked. Pass -v to the script, and doctest prints a detailed log of what it’s trying, and prints a summary at the end:

$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
And so on, eventually ending with:

Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
    ... 
    OverflowError: n too large
ok
2 items passed all tests:
1 tests in __main__
8 tests in __main__.factorial

26.2. doctest — Test interactive Python examples
9 tests in 2 items.
9 passed and 0 failed.
Test passed.
$ 

That's all you need to know to start making productive use of doctest! Jump in. The following sections provide full details. Note that there are many examples of doctests in the standard Python test suite and libraries. Especially useful examples can be found in the standard test file Lib/test/test_doctest.py.

26.2.1 Simple Usage: Checking Examples in Docstrings

The simplest way to start using doctest (but not necessarily the way you’ll continue to do it) is to end each module \( M \) with:

```python
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

doctest then examines docstrings in module \( M \).

Running the module as a script causes the examples in the docstrings to get executed and verified:

```bash
python M.py
```

This won’t display anything unless an example fails, in which case the failing example(s) and the cause(s) of the failure(s) are printed to stdout, and the final line of output is ***Test Failed*** \( N \) failures., where \( N \) is the number of examples that failed.

Run it with the \(-v\) switch instead:

```bash
python M.py -v
```

and a detailed report of all examples tried is printed to standard output, along with assorted summaries at the end.

You can force verbose mode by passing \( \text{verbose=True} \) to \text{testmod()}\), or prohibit it by passing \( \text{verbose=False} \). In either of those cases, \text{sys.argv} is not examined by \text{testmod()} (so passing \(-v\) or \(\text{not has no effect} \).

There is also a command line shortcut for running \text{testmod()}. You can instruct the Python interpreter to run the doctest module directly from the standard library and pass the module name(s) on the command line:

```bash
python -m doctest -v example.py
```

This will import \text{example.py} as a standalone module and run \text{testmod()} on it. Note that this may not work correctly if the file is part of a package and imports other submodules from that package.

For more information on \text{testmod()}, see section Basic API.

26.2.2 Simple Usage: Checking Examples in a Text File

Another simple application of doctest is testing interactive examples in a text file. This can be done with the \text{testfile()} function:

```python
import doctest
doctest.testfile("example.txt")
```

That short script executes and verifies any interactive Python examples contained in the file \text{example.txt}. The file content is treated as if it were a single giant docstring; the file doesn’t need to contain a Python program! For example, perhaps \text{example.txt} contains this:

```
The ''example'' module
===============

Using ''factorial''
------------------
```

Chapter 26. Development Tools
This is an example text file in reStructuredText format. First import "factorial" from the "example" module:

```python
>>> from example import factorial
```

Now use it:

```python
>>> factorial(6)
120
```

Running `doctest.testfile("example.txt")` then finds the error in this documentation:

```plaintext
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

As with `testmod()`, `testfile()` won’t display anything unless an example fails. If an example does fail, then the failing example(s) and the cause(s) of the failure(s) are printed to stdout, using the same format as `testmod()`.

By default, `testfile()` looks for files in the calling module’s directory. See section Basic API for a description of the optional arguments that can be used to tell it to look for files in other locations.

Like `testmod()`, `testfile()`’s verbosity can be set with the `--verbose` command-line switch or with the optional keyword argument `verbose`.

There is also a command line shortcut for running `testfile()`. You can instruct the Python interpreter to run the doctest module directly from the standard library and pass the file name(s) on the command line:

```bash
python -m doctest --verbose example.txt
```

Because the file name does not end with `.py`, `doctest` infers that it must be run with `testfile()`, not `testmod()`.

For more information on `testfile()`, see section Basic API.

### 26.2.3 How It Works

This section examines in detail how doctest works: which docstrings it looks at, how it finds interactive examples, what execution context it uses, how it handles exceptions, and how option flags can be used to control its behavior. This is the information that you need to know to write doctest examples; for information about actually running doctest on these examples, see the following sections.

#### Which Docstrings Are Examined?

The module docstring, and all function, class and method docstrings are searched. Objects imported into the module are not searched.

In addition, if `M.__test__` exists and “is true”, it must be a dict, and each entry maps a (string) name to a function object, class object, or string. Function and class object docstrings found from `M.__test__` are searched, and strings are treated as if they were docstrings. In output, a key `K` in `M.__test__` appears with name `<name of M>.__test__.K`

Any classes found are recursively searched similarly, to test docstrings in their contained methods and nested classes.
How are Docstring Examples Recognized?

In most cases a copy-and-paste of an interactive console session works fine, but doctest isn’t trying to do an exact emulation of any specific Python shell.

```python
# comments are ignored
x = 12
x
if x == 13:
    print("yes")
else:
    print("no")
    print("NO")
    print("NO!!!")
```

Any expected output must immediately follow the final ‘``` ``` or ‘``` ``` line containing the code, and the expected output (if any) extends to the next ‘``` ``` or all-whitespace line.

The fine print:

- Expected output cannot contain an all-whitespace line, since such a line is taken to signal the end of expected output. If expected output does contain a blank line, put `<BLANKLINE>` in your doctest example each place a blank line is expected.
- All hard tab characters are expanded to spaces, using 8-column tab stops. Tabs in output generated by the tested code are not modified. Because any hard tabs in the sample output are expanded, this means that if the code output includes hard tabs, the only way the doctest can pass is if the `NORMALIZE_WHITESPACE` option or `directive` is in effect. Alternatively, the test can be rewritten to capture the output and compare it to an expected value as part of the test. This handling of tabs in the source was arrived at through trial and error, and has proven to be the least error prone way of handling them. It is possible to use a different algorithm for handling tabs by writing a custom `DocTestParser` class.
- Output to stdout is captured, but not output to stderr (exception tracebacks are captured via a different means).
- If you continue a line via backslashing in an interactive session, or for any other reason use a backslash, you should use a raw docstring, which will preserve your backslashes exactly as you type them:

```python
>>> def f(x):
...     r'''Backslashes in a raw docstring: m
'''
>>> print(f.__doc__)
Backslashes in a raw docstring: m
```

Otherwise, the backslash will be interpreted as part of the string. For example, the \n above would be interpreted as a newline character. Alternatively, you can double each backslash in the doctest version (and not use a raw string):

```python
>>> def f(x):
...     '''Backslashes in a raw docstring: m\n'''
>>> print(f.__doc__)
Backslashes in a raw docstring: m
```

- The starting column doesn’t matter:

```python
assert "Easy!"
>>> import math
>>> math.floor(1.9)
1
```
and as many leading whitespace characters are stripped from the expected output as appeared in the initial
' >>> ' line that started the example.

What’s the Execution Context?

By default, each time doctest finds a docstring to test, it uses a shallow copy of $M$’s globals, so that running
tests doesn’t change the module’s real globals, and so that one test in $M$ can’t leave behind crumbs that accidentally
allow another test to work. This means examples can freely use any names defined at top-level in $M$, and names
defined earlier in the docstring being run. Examples cannot see names defined in other docstrings.

You can force use of your own dict as the execution context by passing globs=your_dict to testmod() or
testfile() instead.

What About Exceptions?

No problem, provided that the traceback is the only output produced by the example: just paste in the traceback.¹
Since tracebacks contain details that are likely to change rapidly (for example, exact file paths and line numbers),
this is one case where doctest works hard to be flexible in what it accepts.

Simple example:

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
```

That doctest succeeds if ValueError is raised, with the list.remove(x): x not in list detail as shown.

The expected output for an exception must start with a traceback header, which may be either of the following two
lines, indented the same as the first line of the example:

```
Traceback (most recent call last):
Traceback (innermost last):
```

The traceback header is followed by an optional traceback stack, whose contents are ignored by doctest. The
traceback stack is typically omitted, or copied verbatim from an interactive session.

The traceback stack is followed by the most interesting part: the line(s) containing the exception type and detail.
This is usually the last line of a traceback, but can extend across multiple lines if the exception has a multi-line
detail:

```
>>> raise ValueError('multi
  line
detail')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: multi
  line
detail
```

The last three lines (starting with ValueError) are compared against the exception’s type and detail, and the
rest are ignored.

Best practice is to omit the traceback stack, unless it adds significant documentation value to the example. So the
last example is probably better as:

```
>>> raise ValueError('multi
  line
detail')
Traceback (most recent call last):
...
ValueError: multi
  line
detail
```

¹ Examples containing both expected output and an exception are not supported. Trying to guess where one ends and the other begins is
too error-prone, and that also makes for a confusing test.
Note that tracebacks are treated very specially. In particular, in the rewritten example, the use of ... is independent of doctest’s ELLIPSIS option. The ellipsis in that example could be left out, or could just as well be three (or three hundred) commas or digits, or an indented transcript of a Monty Python skit.

Some details you should read once, but won’t need to remember:

- Doctest can’t guess whether your expected output came from an exception traceback or from ordinary printing. So, e.g., an example that expects ValueError: 42 is prime will pass whether ValueError is actually raised or if the example merely prints that traceback text. In practice, ordinary output rarely begins with a traceback header line, so this doesn’t create real problems.

- Each line of the traceback stack (if present) must be indented further than the first line of the example, or start with a non-alphanumeric character. The first line following the traceback header indented the same and starting with an alphanumeric is taken to be the start of the exception detail. Of course this does the right thing for genuine tracebacks.

- When the IGNORE_EXCEPTION_DETAIL doctest option is specified, everything following the leftmost colon and any module information in the exception name is ignored.

- The interactive shell omits the traceback header line for some SyntaxErrors. But doctest uses the traceback header line to distinguish exceptions from non-exceptions. So in the rare case where you need to test a SyntaxError that omits the traceback header, you will need to manually add the traceback header line to your test example.

- For some SyntaxErrors, Python displays the character position of the syntax error, using a ^ marker:

  >>> 1 1
     File "<stdin>", line 1
     1 1
     ^
  SyntaxError: invalid syntax

  Since the lines showing the position of the error come before the exception type and detail, they are not checked by doctest. For example, the following test would pass, even though it puts the ^ marker in the wrong location:

  >>> 1 1
     Traceback (most recent call last):
     File "<stdin>", line 1
     1 1
     ^
     SyntaxError: invalid syntax

Option Flags

A number of option flags control various aspects of doctest’s behavior. Symbolic names for the flags are supplied as module constants, which can be or’ed together and passed to various functions. The names can also be used in doctest directives.

The first group of options define test semantics, controlling aspects of how doctest decides whether actual output matches an example’s expected output:

doctest.DONT_ACCEPT_TRUE_FOR_1
  By default, if an expected output block contains just 1, an actual output block containing just 1 or just True is considered to be a match, and similarly for 0 versus False. When DONT_ACCEPT_TRUE_FOR_1 is specified, neither substitution is allowed. The default behavior caters to that Python changed the return type of many functions from integer to boolean; doctests expecting “little integer” output still work in these cases. This option will probably go away, but not for several years.

doctest.DONT_ACCEPT_BLANKLINE
  By default, if an expected output block contains a line containing only the string <BLANKLINE>, then that line will match a blank line in the actual output. Because a genuinely blank line delimits the expected output,
this is the only way to communicate that a blank line is expected. When `DONT_ACCEPT_BLANKLINE` is specified, this substitution is not allowed.

**doctest.NORMALIZE_WHITESPACE**

When specified, all sequences of whitespace (blanks and newlines) are treated as equal. Any sequence of whitespace within the expected output will match any sequence of whitespace within the actual output. By default, whitespace must match exactly. `NORMALIZE_WHITESPACE` is especially useful when a line of expected output is very long, and you want to wrap it across multiple lines in your source.

**doctest.ELLIPSIS**

When specified, an ellipsis marker (…) in the expected output can match any substring in the actual output. This includes substrings that span line boundaries, and empty substrings, so it’s best to keep usage of this simple. Complicated uses can lead to the same kinds of “oops, it matched too much!” surprises that `.*` is prone to in regular expressions.

**doctest.IGNORE_EXCEPTION_DETAIL**

When specified, an example that expects an exception passes if an exception of the expected type is raised, even if the exception detail does not match. For example, an example expecting `ValueError: 42` will pass if the actual exception raised is `ValueError: 3+14`, but will fail, e.g., if `TypeError` is raised.

It will also ignore the module name used in Python 3 doctest reports. Hence both of these variations will work with the flag specified, regardless of whether the test is run under Python 2.7 or Python 3.2 (or later versions):

```python
>>> raise CustomError('message')
CustomError: message

>>> raise CustomError('message')
my_module.CustomError: message
```

Note that `ELLIPSIS` can also be used to ignore the details of the exception message, but such a test may still fail based on whether or not the module details are printed as part of the exception name. Using `IGNORE_EXCEPTION_DETAIL` and the details from Python 2.3 is also the only clear way to write a doctest that doesn’t care about the exception detail yet continues to pass under Python 2.3 or earlier (those releases do not support doctest directives and ignore them as irrelevant comments). For example:

```python
>>> (1, 2)[3] = 'moo'
TypeError: object doesn’t support item assignment
```

passes under Python 2.3 and later Python versions with the flag specified, even though the detail changed in Python 2.4 to say “does not” instead of “doesn’t”. Changed in version 3.2: `IGNORE_EXCEPTION_DETAIL` now also ignores any information relating to the module containing the exception under test.

**doctest.SKIP**

When specified, do not run the example at all. This can be useful in contexts where doctest examples serve as both documentation and test cases, and an example should be included for documentation purposes, but should not be checked. E.g., the example’s output might be random; or the example might depend on resources which would be unavailable to the test driver.

The SKIP flag can also be used for temporarily “commenting out” examples.

**doctest.COMPARISON_FLAGS**

A bitmask or’ing together all the comparison flags above.

The second group of options controls how test failures are reported:
doctest.REPORT_UDIFF
   When specified, failures that involve multi-line expected and actual outputs are displayed using a unified diff.

doctest.REPORT_CDIFF
   When specified, failures that involve multi-line expected and actual outputs will be displayed using a context diff.

doctest.REPORT_NDIFF
   When specified, differences are computed by difflib.Differ, using the same algorithm as the popular ndiff.py utility. This is the only method that marks differences within lines as well as across lines. For example, if a line of expected output contains digit 1 where actual output contains letter l, a line is inserted with a caret marking the mismatching column positions.

doctest.REPORT_ONLY_FIRST_FAILURE
   When specified, display the first failing example in each doctest, but suppress output for all remaining examples. This will prevent doctest from reporting correct examples that break because of earlier failures; but it might also hide incorrect examples that fail independently of the first failure. When REPORT_ONLY_FIRST_FAILURE is specified, the remaining examples are still run, and still count towards the total number of failures reported; only the output is suppressed.

doctest.REPORTING_FLAGS
   A bitmask or’ing together all the reporting flags above.

There is also a way to register new option flag names, though this isn’t useful unless you intend to extend doctest internals via subclassing:

doctest.register_optionflag(name)
   Create a new option flag with a given name, and return the new flag’s integer value. register_optionflag() can be used when subclassing OutputChecker or DocTestRunner to create new options that are supported by your subclasses. register_optionflag() should always be called using the following idiom:

   MY_FLAG = register_optionflag(‘MY_FLAG’)

Directives

Doctest directives may be used to modify the option flags for an individual example. Doctest directives are special Python comments following an example’s source code:

   directive ::= "#" "doctest:" directive_options
   directive_options ::= directive_option ("," directive_option)\*
   directive_option ::= on_or_off directive_option_name
   on_or_off ::= "+" | "-"
   directive_option_name ::= "DONT_ACCEPT_BLANKLINE" | "NORMALIZE_WHITESPACE" | ...

Whitespace is not allowed between the + or - and the directive option name. The directive option name can be any of the option flag names explained above.

An example’s doctest directives modify doctest’s behavior for that single example. Use + to enable the named behavior, or - to disable it.

For example, this test passes:

   >>> print(list(range(20))) # doctest: +NORMALIZE_WHITESPACE
   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

Without the directive it would fail, both because the actual output doesn’t have two blanks before the single-digit list elements, and because the actual output is on a single line. This test also passes, and also requires a directive to do so:
Multiple directives can be used on a single physical line, separated by commas:

```python
>>> print(list(range(20)))  # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
[0, 1, ..., 18, 19]
```

If multiple directive comments are used for a single example, then they are combined:

```python
>>> print(list(range(20)))  # doctest: +ELLIPSIS
...
# doctest: +NORMALIZE_WHITESPACE
[0, 1, ..., 18, 19]
```

As the previous example shows, you can add ... lines to your example containing only directives. This can be useful when an example is too long for a directive to comfortably fit on the same line:

```python
>>> print(list(range(5)) + list(range(10, 20)) + list(range(30, 40)))
...
# doctest: +ELLIPSIS
[0, ..., 4, 10, ..., 19, 30, ..., 39]
```

Note that since all options are disabled by default, and directives apply only to the example they appear in, enabling options (via + in a directive) is usually the only meaningful choice. However, option flags can also be passed to functions that run doctests, establishing different defaults. In such cases, disabling an option via − in a directive can be useful.

### Warnings

doctest is serious about requiring exact matches in expected output. If even a single character doesn’t match, the test fails. This will probably surprise you a few times, as you learn exactly what Python does and doesn’t guarantee about output. For example, when printing a dict, Python doesn’t guarantee that the key-value pairs will be printed in any particular order, so a test like

```python
>>> foo()
{"Hermione": "hippogryph", "Harry": "broomstick"}
```

is vulnerable! One workaround is to do

```python
>>> foo() == {"Hermione": "hippogryph", "Harry": "broomstick"}
True
```

instead. Another is to do

```python
d = sorted(foo().items())
>>> d
[('Harry', 'broomstick'), ('Hermione', 'hippogryph')]
```

There are others, but you get the idea.

Another bad idea is to print things that embed an object address, like

```python
>>> id(1.0)  # certain to fail some of the time
7948648
```

```python
>>> class C: pass
>>> C()  # the default repr() for instances embeds an address
<__main__.C instance at 0x00AC18F0>
```

The ELLIPSIS directive gives a nice approach for the last example:

```python
>>> C()  # doctest: +ELLIPSIS
<__main__.C instance at 0x...>
```

Floating-point numbers are also subject to small output variations across platforms, because Python defers to the platform C library for float formatting, and C libraries vary widely in quality here.
Numbers of the form \( \frac{I}{2^J} \) are safe across all platforms, and I often contrive doctest examples to produce numbers of that form:

```python
>>> 3./4  # utterly safe
0.75
```

Simple fractions are also easier for people to understand, and that makes for better documentation.

### 26.2.4 Basic API

The functions `testmod()` and `testfile()` provide a simple interface to doctest that should be sufficient for most basic uses. For a less formal introduction to these two functions, see sections *Simple Usage: Checking Examples in Docstrings* and *Simple Usage: Checking Examples in a Text File*.  

```python
doctest.testfile(filename, module_relative=True, name=None, package=None, globs=None, verbose=None, report=True, optionflags=0, extraglobs=None, raise_on_error=False, parser=DocTestParser(), encoding=None)
```

All arguments except `filename` are optional, and should be specified in keyword form.

Test examples in the file named `filename`. Return `(failure_count, test_count)`.

Optional argument `module_relative` specifies how the filename should be interpreted:

- If `module_relative` is `True` (the default), then `filename` specifies an OS-independent module-relative path. By default, this path is relative to the calling module’s directory; but if the `package` argument is specified, then it is relative to that package. To ensure OS-independence, `filename` should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).

- If `module_relative` is `False`, then `filename` specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument `name` gives the name of the test; by default, or if `None`, `os.path.basename(filename)` is used.

Optional argument `package` is a Python package or the name of a Python package whose directory should be used as the base directory for a module-relative filename. If no package is specified, then the calling module’s directory is used as the base directory for module-relative filenames. It is an error to specify `package` if `module_relative` is `False`.

Optional argument `globs` gives a dict to be used as the globals when executing examples. A new shallow copy of this dict is created for the doctest, so its examples start with a clean slate. By default, or if `None`, a new empty dict is used.

Optional argument `extraglobs` gives a dict merged into the globals used to execute examples. This works like `dict.update()`: if `globs` and `extraglobs` have a common key, the associated value in `extraglobs` appears in the combined dict. By default, or if `None`, no extra globals are used. This is an advanced feature that allows parameterization of doctests. For example, a doctest can be written for a base class, using a generic name for the class, then reused to test any number of subclasses by passing an `extraglobs` dict mapping the generic name to the subclass to be tested.

Optional argument `verbose` prints lots of stuff if true, and prints only failures if false; by default, or if `None`, it’s true if and only if ‘-v’ is in `sys.argv`.

Optional argument `report` prints a summary at the end true, else prints nothing at the end. In verbose mode, the summary is detailed, else the summary is very brief (in fact, empty if all tests passed).

Optional argument `optionflags` or ‘s together option flags. See section *Option Flags*. 
Optional argument `raise_on_error` defaults to false. If true, an exception is raised upon the first failure or unexpected exception in an example. This allows failures to be post-mortem debugged. Default behavior is to continue running examples.

Optional argument `parser` specifies a `DocTestParser` (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument `encoding` specifies an encoding that should be used to convert the file to unicode.

```
doctest.testmod(m=None, name=None, globs=None, verbose=None, report=True, optionflags=0, extraglobs=None, raise_on_error=False, exclude_empty=False)
```

All arguments are optional, and all except for `m` should be specified in keyword form.

Test examples in docstrings in functions and classes reachable from module `m` (or module `__main__` if `m` is not supplied or is `None`), starting with `m.__doc__`.

Also test examples reachable from dict `m.__test__`, if it exists and is not `None`. `m.__test__` maps names (strings) to functions, classes and strings; function and class docstrings are searched for examples; strings are searched directly, as if they were docstrings.

Only doctests attached to objects belonging to module `m` are searched.

Return `(failure_count, test_count)`.

Optional argument `name` gives the name of the module; by default, or if `None`, `m.__name__` is used.

Optional argument `exclude_empty` defaults to false. If true, objects for which no doctests are found are excluded from consideration. The default is a backward compatibility hack, so that code still using `doctest.master.summarize()` in conjunction with `testmod()` continues to get output for objects with no tests. The `exclude_empty` argument to the newer `DocTestFinder` constructor defaults to true.

Optional arguments `extraglobs`, `verbose`, `report`, `optionflags`, `raise_on_error`, and `globs` are the same as for function `testfile()` above, except that `globs` defaults to `m.__dict__`.

There’s also a function to run the doctests associated with a single object. This function is provided for backward compatibility. There are no plans to deprecate it, but it’s rarely useful:

```
doctest.run_docstring_examples(f, globs=None, verbose=False, name="NoName", compileflags=None, optionflags=0)
```

Test examples associated with object `f`; for example, `f` may be a module, function, or class object.

A shallow copy of dictionary argument `globs` is used for the execution context.

Optional argument `name` is used in failure messages, and defaults to "NoName".

If optional argument `verbose` is true, output is generated even if there are no failures. By default, output is generated only in case of an example failure.

Optional argument `compileflags` gives the set of flags that should be used by the Python compiler when running the examples. By default, or if `None`, flags are deduced corresponding to the set of future features found in `globs`.

Optional argument `optionflags` works as for function `testfile()` above.

### 26.2.5 unittest API

As your collection of doctest’ed modules grows, you’ll want a way to run all their doctests systematically. `doctest` provides two functions that can be used to create `unittest` test suites from modules and text files containing doctests. To integrate with `unittest` test discovery, include a `load_tests()` function in your test module:

```
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):
```

26.2. doctest — Test interactive Python examples
tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
return tests

There are two main functions for creating unittest.TestSuite instances from text files and modules with doctests:

doctest.DocFileSuite(*paths, module_relative=True, package=None, setUp=None, tearDown=None, globs=None, optionflags=0, parser=DocTestParser(), encoding=None)

Convert doctest tests from one or more text files to a unittest.TestSuite.

The returned unittest.TestSuite is to be run by the unittest framework and runs the interactive examples in each file. If an example in any file fails, then the synthesized unit test fails, and a failureException exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Pass one or more paths (as strings) to text files to be examined.

Options may be provided as keyword arguments:

Optional argument module_relative specifies how the filenames in paths should be interpreted:

• If module_relative is True (the default), then each filename in paths specifies an OS-independent module-relative path. By default, this path is relative to the calling module’s directory; but if the package argument is specified, then it is relative to that package. To ensure OS-independence, each filename should use / characters to separate path segments, and may not be an absolute path (i.e., it may not begin with /).

• If module_relative is False, then each filename in paths specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument package is a Python package or the name of a Python package whose directory should be used as the base directory for module-relative filenames in paths. If no package is specified, then the calling module’s directory is used as the base directory for module-relative filenames. It is an error to specify package if module_relative is False.

Optional argument setUp specifies a set-up function for the test suite. This is called before running the tests in each file. The setUp function will be passed a DocTest object. The setUp function can access the test globals as the globs attribute of the test passed.

Optional argument tearDown specifies a tear-down function for the test suite. This is called after running the tests in each file. The tearDown function will be passed a DocTest object. The setUp function can access the test globals as the globs attribute of the test passed.

Optional argument globs is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, globs is a new empty dictionary.

Optional argument optionflags specifies the default doctest options for the tests, created by or-ing together individual option flags. See section Option Flags. See function set_unittest_reportflags() below for a better way to set reporting options.

Optional argument parser specifies a DocTestParser (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., DocTestParser()).

Optional argument encoding specifies an encoding that should be used to convert the file to unicode.

The global __file__ is added to the globals provided to doctests loaded from a text file using DocFileSuite().

doctest.DocTestSuite(module=None, globs=None, extraglobs=None, test_finder=None, setUp=None, tearDown=None, checker=None)

Convert doctest tests for a module to a unittest.TestSuite.

The returned unittest.TestSuite is to be run by the unittest framework and runs each doctest in the module. If any of the doctests fail, then the synthesized unit test fails, and a failureException exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.
Optional argument *module* provides the module to be tested. It can be a module object or a (possibly dotted) module name. If not specified, the module calling this function is used.

Optional argument *globs* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globs* is a new empty dictionary.

Optional argument *extraglobs* specifies an extra set of global variables, which is merged into *globs*. By default, no extra globals are used.

Optional argument *test_finder* is the *DocTestFinder* object (or a drop-in replacement) that is used to extract doctests from the module.

Optional arguments *setUp*, *tearDown*, and *optionflags* are the same as for function *DocFileSuite()* above.

This function uses the same search technique as *testmod()*.

**Note:** Unlike *testmod()* and *DocTestFinder*, this function raises a *ValueError* if *module* contains no docstrings. You can prevent this error by passing a *DocTestFinder* instance as the *test_finder* argument with its *exclude_empty* keyword argument set to *False*:

```python
>>> finder = doctest.DocTestFinder(exclude_empty=False)
>>> suite = doctest.DocTestSuite(test_finder=finder)
```

Under the covers, *DocTestSuite()* creates a *unittest.TestSuite* out of *doctest.DocTestCase* instances, and *DocTestCase* is a subclass of *unittest.TestCase*. *DocTestCase* isn’t documented here (it’s an internal detail), but studying its code can answer questions about the exact details of *unittest* integration.

Similarly, *DocFileSuite()* creates a *unittest.TestSuite* out of *doctest.DocFileCase* instances, and *DocFileCase* is a subclass of *DocTestCase*.

So both ways of creating a *unittest.TestSuite* run instances of *DocTestCase*. This is important for a subtle reason: when you run *doctest* functions yourself, you can control the *doctest* options in use directly, by passing option flags to *doctest* functions. However, if you’re writing a *unittest* framework, *unittest* ultimately controls when and how tests get run. The framework author typically wants to control *doctest* reporting options (perhaps, e.g., specified by command line options), but there’s no way to pass options through *unittest* to *doctest* test runners.

For this reason, *doctest* also supports a notion of *doctest* reporting flags specific to *unittest* support, via this function:

```python
doctest.set_unittest_reportflags(flags)
```

Set the *doctest* reporting flags to use.

Argument *flags* or’s together option flags. See section *Option Flags*. Only “reporting flags” can be used.

This is a module-global setting, and affects all future doctests run by module *unittest: the runTest()* method of *DocTestCase* looks at the option flags specified for the test case when the *DocTestCase* instance was constructed. If no reporting flags were specified (which is the typical and expected case), *doctest: s unittest* reporting flags are or’ed into the option flags, and the option flags so augmented are passed to the *DocTestRunner* instance created to run the doctest. If any reporting flags were specified when the *DocTestCase* instance was constructed, *doctest: s unittest* reporting flags are ignored.

The value of the *unittest* reporting flags in effect before the function was called is returned by the function.

### 26.2.6 Advanced API

The basic API is a simple wrapper that’s intended to make *doctest* easy to use. It is fairly flexible, and should meet most users’ needs; however, if you require more fine-grained control over testing, or wish to extend *doctest*’s capabilities, then you should use the advanced API.
The advanced API revolves around two container classes, which are used to store the interactive examples extracted from doctest cases:

- **Example**: A single Python `statement`, paired with its expected output.
- **DocTest**: A collection of Examples, typically extracted from a single docstring or text file.

Additional processing classes are defined to find, parse, and run, and check doctest examples:

- **DocTestFinder**: Finds all docstrings in a given module, and uses a DocTestParser to create a DocTest from every docstring that contains interactive examples.
- **DocTestParser**: Creates a DocTest object from a string (such as an object’s docstring).
- **DocTestRunner**: Executes the examples in a DocTest, and uses an OutputChecker to verify their output.
- **OutputChecker**: Compares the actual output from a doctest example with the expected output, and decides whether they match.

The relationships among these processing classes are summarized in the following diagram:

```
+-------+ +---------+ +---------+ +-------+ +---------+ +-------+ +---------+ +---------+ +-------+ +---------+ +-------+ +---------+ +-------+ +---------+ +-------+ +---------+ +-------+ +---------+ +-------+ +---------+ +-------+ +---------+ +-------+ +---------+ +-------+ +---------+ +-------+ +---------+ +-------+ +---------+ +-------+ +---------+ +-------+ +---------+ +-------+
|module| --DocTestFinder-> | DocTest | --DocTestRunner-> results
|       | ^ | ^ +---------+ | ^ | (printed)
|       | | | | Example | | |
|       | v | | .. | v |
| DocTestParser | Example | OutputChecker
| +---------|
```

**DocTest Objects**

```python
class doctest.DocTest (examples, globs, name, filename, lineno, docstring)
```

A collection of doctest examples that should be run in a single namespace. The constructor arguments are used to initialize the attributes of the same names.

**DocTest** defines the following attributes. They are initialized by the constructor, and should not be modified directly.

- **examples**
  
  A list of `Example` objects encoding the individual interactive Python examples that should be run by this test.

- **globs**
  
  The namespace (aka globals) that the examples should be run in. This is a dictionary mapping names to values. Any changes to the namespace made by the examples (such as binding new variables) will be reflected in `globs` after the test is run.

- **name**
  
  A string name identifying the `DocTest`. Typically, this is the name of the object or file that the test was extracted from.

- **filename**
  
  The name of the file that this `DocTest` was extracted from; or `None` if the filename is unknown, or if the `DocTest` was not extracted from a file.

- **lineno**
  
  The line number within `filename` where this `DocTest` begins, or `None` if the line number is unavailable. This line number is zero-based with respect to the beginning of the file.

- **docstring**
  
  The string that the test was extracted from, or ‘None’ if the string is unavailable, or if the test was not extracted from a string.
Example Objects

```python
class doctest.Example (source, want, exc_msg=None, lineno=0, indent=0, options=None)
```
A single interactive example, consisting of a Python statement and its expected output. The constructor arguments are used to initialize the attributes of the same names.

Example defines the following attributes. They are initialized by the constructor, and should not be modified directly.

**source**
A string containing the example’s source code. This source code consists of a single Python statement, and always ends with a newline; the constructor adds a newline when necessary.

**want**
The expected output from running the example’s source code (either from stdout, or a traceback in case of exception). want ends with a newline unless no output is expected, in which case it’s an empty string. The constructor adds a newline when necessary.

**exc_msg**
The exception message generated by the example, if the example is expected to generate an exception; or None if it is not expected to generate an exception. This exception message is compared against the return value of `traceback.format_exception_only()`. exc_msg ends with a newline unless it’s None. The constructor adds a newline if needed.

**lineno**
The line number within the string containing this example where the example begins. This line number is zero-based with respect to the beginning of the containing string.

**indent**
The example’s indentation in the containing string, i.e., the number of space characters that precede the example’s first prompt.

**options**
A dictionary mapping from option flags to True or False, which is used to override default options for this example. Any option flags not contained in this dictionary are left at their default value (as specified by the `DocTestRunner`’s `optionflags`). By default, no options are set.

DocTestFinder objects

```python
class doctest.DocTestFinder (verbose=False, parser=DocTestParser(), recurse=True, exclude_empty=True)
```
A processing class used to extract the DocTests that are relevant to a given object, from its docstring and the docstrings of its contained objects. DocTests can currently be extracted from the following object types: modules, functions, classes, methods, staticmethods, classmethods, and properties.

The optional argument verbose can be used to display the objects searched by the finder. It defaults to False (no output).

The optional argument parser specifies the `DocTestParser` object (or a drop-in replacement) that is used to extract doctests from docstrings.

If the optional argument recurse is false, then `DocTestFinder.find()` will only examine the given object, and not any contained objects.

If the optional argument exclude_empty is false, then `DocTestFinder.find()` will include tests for objects with empty docstrings.

`DocTestFinder` defines the following method:

**find**(obj[, name[, module[, globs[, extraglobs]]]])
Return a list of the DocTests that are defined by obj’s docstring, or by any of its contained objects’ docstrings.

The optional argument name specifies the object’s name; this name will be used to construct names for the returned DocTests. If name is not specified, then obj.__name__ is used.
The optional parameter *module* is the module that contains the given object. If the module is not specified or is `None`, then the test finder will attempt to automatically determine the correct module. The object’s module is used:

- As a default namespace, if *globs* is not specified.
- To prevent the DocTestFinder from extracting DocTests from objects that are imported from other modules. (Contained objects with modules other than *module* are ignored.)
- To find the name of the file containing the object.
- To help find the line number of the object within its file.

If *module* is `False`, no attempt to find the module will be made. This is obscure, of use mostly in testing doctest itself: if *module* is `False` or `None` but cannot be found automatically, then all objects are considered to belong to the (non-existent) module, so all contained objects will (recursively) be searched for doctests.

The globals for each DocTest is formed by combining *globs* and *extraglobs* (bindings in *extraglobs* override bindings in *globs*). A new shallow copy of the globals dictionary is created for each DocTest. If *globs* is not specified, then it defaults to the module’s `__dict__`, if specified, or `{}` otherwise. If *extraglobs* is not specified, then it defaults to `{}`.

**DocTestParser objects**

class `doctest.DoctestParser`

A processing class used to extract interactive examples from a string, and use them to create a DocTest object.

`DocTestParser` defines the following methods:

**get_doctest**(string, globs, name, filename, lineno)

Extract all doctest examples from the given string, and collect them into a DocTest object. *globs*, *name*, *filename*, and *lineno* are attributes for the new DocTest object. See the documentation for DocTest for more information.

**get_examples**(string, name='<string>')

Extract all doctest examples from the given string, and return them as a list of Example objects. Line numbers are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

**parse**(string, name='<string>')

Divide the given string into examples and intervening text, and return them as a list of alternating Examples and strings. Line numbers for the Examples are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

**DocTestRunner objects**

class `doctest.DoctestRunner`(checker=None, verbose=None, optionflags=0)

A processing class used to execute and verify the interactive examples in a DocTest.

The comparison between expected outputs and actual outputs is done by an OutputChecker. This comparison may be customized with a number of option flags; see section Option Flags for more information.

If the option flags are insufficient, then the comparison may also be customized by passing a subclass of OutputChecker to the constructor.

The test runner’s display output can be controlled in two ways. First, an output function can be passed to TestRunner.run(); this function will be called with strings that should be displayed. It defaults to `sys.stdout.write`. If capturing the output is not sufficient, then the display output can be also customized by subclassing DocTestRunner, and overriding the methods `report_start()`, `report_success()`, `report_unexpected_exception()`, and `report_failure()`.
The optional keyword argument `checker` specifies the `OutputChecker` object (or drop-in replacement) that should be used to compare the expected outputs to the actual outputs of doctest examples.

The optional keyword argument `verbose` controls the `DocTestRunner`'s verbosity. If `verbose` is `True`, then information is printed about each example, as it is run. If `verbose` is `False`, then only failures are printed. If `verbose` is unspecified, or `None`, then `verbose` output is used iff the command-line switch `-v` is used.

The optional keyword argument `optionflags` can be used to control how the test runner compares expected output to actual output, and how it displays failures. For more information, see section `Option Flags`.

`DocTestParser` defines the following methods:

- `report_start(out, test, example)`
  Report that the test runner is about to process the given example. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

  `example` is the example about to be processed. `test` is the test containing example. `out` is the output function that was passed to `DocTestRunner.run()`.

- `report_success(out, test, example, got)`
  Report that the given example ran successfully. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

  `example` is the example about to be processed. `got` is the actual output from the example. `test` is the test containing example. `out` is the output function that was passed to `DocTestRunner.run()`.

- `report_failure(out, test, example, got)`
  Report that the given example failed. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

  `example` is the example about to be processed. `got` is the actual output from the example. `test` is the test containing example. `out` is the output function that was passed to `DocTestRunner.run()`.

- `report_unexpected_exception(out, test, example, exc_info)`
  Report that the given example raised an unexpected exception. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

  `example` is the example about to be processed. `exc_info` is a tuple containing information about the unexpected exception (as returned by `sys.exc_info()`). `test` is the test containing example. `out` is the output function that was passed to `DocTestRunner.run()`.

- `run(test, compileflags=None, out=None, clear_globs=True)`
  Run the examples in `test` (a `DocTest` object), and display the results using the writer function `out`.

  The examples are run in the namespace `test.globs`. If `clear_globs` is true (the default), then this namespace will be cleared after the test runs, to help with garbage collection. If you would like to examine the namespace after the test completes, then use `clear_globs=False`.

  `compileflags` gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to `globs`.

  The output of each example is checked using the `DocTestRunner`'s output checker, and the results are formatted by the `DocTestRunner`'s `report_*()` methods.

- `summarize(verbose=None)`
  Print a summary of all the test cases that have been run by this `DocTestRunner`, and return a named tuple `TestResults(failed, attempted)`.

  The optional `verbose` argument controls how detailed the summary is. If the verbosity is not specified, then the `DocTestRunner`'s verbosity is used.

### OutputChecker objects

**class doctest.OutputChecker**

A class used to check the whether the actual output from a doctest example matches the expected output.
OutputChecker defines two methods: check_output(), which compares a given pair of outputs, and returns true if they match; and output_difference(), which returns a string describing the differences between two outputs.

OutputChecker defines the following methods:

check_output (want, got, optionflags)
Return True iff the actual output from an example (got) matches the expected output (want). These strings are always considered to match if they are identical; but depending on what option flags the test runner is using, several non-exact match types are also possible. See section Option Flags for more information about option flags.

output_difference (example, got, optionflags)
Return a string describing the differences between the expected output for a given example (example) and the actual output (got). optionflags is the set of option flags used to compare want and got.

26.2.7 Debugging
Doctest provides several mechanisms for debugging doctest examples:

• Several functions convert doctests to executable Python programs, which can be run under the Python debugger, pdb.

• The DebugRunner class is a subclass of DocTestRunner that raises an exception for the first failing example, containing information about that example. This information can be used to perform post-mortem debugging on the example.

• The unittest cases generated by DocTestSuite() support the debug() method defined by unittest.TestCase.

• You can add a call to pdb.set_trace() in a doctest example, and you’ll drop into the Python debugger when that line is executed. Then you can inspect current values of variables, and so on. For example, suppose a.py contains just this module docstring:

```python
>>> def f(x):
...  g(x*2)
>>> def g(x):
...  print(x+3)
...  import pdb; pdb.set_trace()
>>> f(3)
g
```

Then an interactive Python session may look like this:

```python
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
1    def g(x):
2       print(x+3)
3 ->   import pdb; pdb.set_trace()
[EOF]
(Pdb) p x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
-> g(x*2)
```

---

1096 Chapter 26. Development Tools
Functions that convert doctests to Python code, and possibly run the synthesized code under the debugger:

**doctest.script_from_examples(s)**
Convert text with examples to a script.

Argument `s` is a string containing doctest examples. The string is converted to a Python script, where doctest examples in `s` are converted to regular code, and everything else is converted to Python comments. The generated script is returned as a string. For example,

```python
import doctest
print(doctest.script_from_examples(r""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2
        Print their sum:
        >>> print(x+y)
            3
""))
```

displays:

```python
# Set x and y to 1 and 2.
x, y = 1, 2
# # Print their sum:
print(x+y)
## 3
```

This function is used internally by other functions (see below), but can also be useful when you want to transform an interactive Python session into a Python script.

**doctest.testsource(module, name)**
Convert the doctest for an object to a script.

Argument `module` is a module object, or dotted name of a module, containing the object whose doctests are of interest. Argument `name` is the name (within the module) of the object with the doctests of interest. The result is a string, containing the object’s docstring converted to a Python script, as described for `script_from_examples()` above. For example, if module `a.py` contains a top-level function `f()`, then

```python
import a, doctest
print(doctest.testsource(a, "a.f"))
```

prints a script version of function `f()`’s docstring, with doctests converted to code, and the rest placed in comments.
doctest.debug (module, name, pm=False)

Debug the doctests for an object.

The module and name arguments are the same as for function testsource() above. The synthesized
Python script for the named object’s docstring is written to a temporary file, and then that file is run under
the control of the Python debugger, pdb.

A shallow copy of module.__dict__ is used for both local and global execution context.

Optional argument pm controls whether post-mortem debugging is used. If pm has a true value, the script
file is run directly, and the debugger gets involved only if the script terminates via raising an unhandled
exception. If it does, then post-mortem debugging is invoked, via pdb.post_mortem(), passing the
traceback object from the unhandled exception. If pm is not specified, or is false, the script is run under
the debugger from the start, via passing an appropriate exec() call to pdb.run().

doctest.debug_src (src, pm=False, globs=None)

Debug the doctests in a string.

This is like function debug() above, except that a string containing doctest examples is specified directly,
via the src argument.

Optional argument pm has the same meaning as in function debug() above.

Optional argument globs gives a dictionary to use as both local and global execution context. If not specified,
or None, an empty dictionary is used. If specified, a shallow copy of the dictionary is used.

The DebugRunner class, and the special exceptions it may raise, are of most interest to testing framework
authors, and will only be sketched here. See the source code, and especially DebugRunner’s docstring (which
is a doctest!) for more details:

class doctest.DebugRunner (checker=None, verbose=None, optionflags=0)
A subclass of DocTestRunner that raises an exception as soon as a failure is encountered. If an unex-
pected exception occurs, an UnexpectedException exception is raised, containing the test, the exam-
ple, and the original exception. If the output doesn’t match, then a DocTestFailure exception is raised,
containing the test, the example, and the actual output.

For information about the constructor parameters and methods, see the documentation for
DocTestRunner in section Advanced API.

There are two exceptions that may be raised by DebugRunner instances:

exception doctest.DocTestFailure (test, example, got)
An exception raised by DocTestRunner to signal that a doctest example’s actual output did not match its
expected output. The constructor arguments are used to initialize the attributes of the same names.

DocTestFailure defines the following attributes:

DocTestFailure.test
The DocTest object that was being run when the example failed.

DocTestFailure.example
The Example that failed.

DocTestFailure.got
The example’s actual output.

exception doctest.UnexpectedException (test, example, exc_info)
An exception raised by DocTestRunner to signal that a doctest example raised an unexpected exception.
The constructor arguments are used to initialize the attributes of the same names.

UnexpectedException defines the following attributes:

UnexpectedException.test
The DocTest object that was being run when the example failed.

UnexpectedException.example
The Example that failed.
UnexpectedException.exc_info
A tuple containing information about the unexpected exception, as returned by sys.exc_info().

26.2.8 Soapbox

As mentioned in the introduction, doctest has grown to have three primary uses:

1. Checking examples in docstrings.
2. Regression testing.
3. Executable documentation / literate testing.

These uses have different requirements, and it is important to distinguish them. In particular, filling your docstrings with obscure test cases makes for bad documentation.

When writing a docstring, choose docstring examples with care. There’s an art to this that needs to be learned—it may not be natural at first. Examples should add genuine value to the documentation. A good example can often be worth many words. If done with care, the examples will be invaluable for your users, and will pay back the time it takes to collect them many times over as the years go by and things change. I’m still amazed at how often one of my doctest examples stops working after a “harmless” change.

Doctest also makes an excellent tool for regression testing, especially if you don’t skimp on explanatory text. By interleaving prose and examples, it becomes much easier to keep track of what’s actually being tested, and why. When a test fails, good prose can make it much easier to figure out what the problem is, and how it should be fixed. It’s true that you could write extensive comments in code-based testing, but few programmers do. Many have found that using doctest approaches instead leads to much clearer tests. Perhaps this is simply because doctest makes writing prose a little easier than writing code, while writing comments in code is a little harder. I think it goes deeper than just that: the natural attitude when writing a doctest-based test is that you want to explain the fine points of your software, and illustrate them with examples. This in turn naturally leads to test files that start with the simplest features, and logically progress to complications and edge cases. A coherent narrative is the result, instead of a collection of isolated functions that test isolated bits of functionality seemingly at random. It’s a different attitude, and produces different results, blurring the distinction between testing and explaining.

Regression testing is best confined to dedicated objects or files. There are several options for organizing tests:

• Write text files containing test cases as interactive examples, and test the files using testfile() or DocFileSuite(). This is recommended, although is easiest to do for new projects, designed from the start to use doctest.

• Define functions named _regrtest_topic that consist of single docstrings, containing test cases for the named topics. These functions can be included in the same file as the module, or separated out into a separate test file.

• Define a __test__ dictionary mapping from regression test topics to docstrings containing test cases.

26.3 unittest — Unit testing framework

(If you are already familiar with the basic concepts of testing, you might want to skip to the list of assert methods.)

The unittest unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

To achieve this, unittest supports some important concepts in an object-oriented way:

test fixture A test fixture represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

test case A test case is the individual unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class, TestCase, which may be used to create new test cases.
**test suite** A test suite is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

**test runner** A test runner is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

See Also:

Module doctest Another test-support module with a very different flavor.

Simple Smalltalk Testing: With Patterns Kent Beck’s original paper on testing frameworks using the pattern shared by unittest.

Nose and py.test Third-party unittest frameworks with a lighter-weight syntax for writing tests. For example,

```python
assert func(10) == 42.
```

The Python Testing Tools Taxonomy An extensive list of Python testing tools including functional testing frameworks and mock object libraries.

Testing in Python Mailing List A special-interest-group for discussion of testing, and testing tools, in Python.

The script `Tools/unittestgui/unittestgui.py` in the Python source distribution is a GUI tool for test discovery and execution. This is intended largely for ease of use for those new to unit testing. For production environments it is recommended that tests be driven by a continuous integration system such as Buildbot, Jenkins or Hudson.

### 26.3.1 Basic example

The unittest module provides a rich set of tools for constructing and running tests. This section demonstrates that a small subset of the tools suffice to meet the needs of most users.

Here is a short script to test three functions from the `random` module:

```python
import random
import unittest

class TestSequenceFunctions(unittest.TestCase):

    def setUp(self):
        self.seq = list(range(10))

    def test_shuffle(self):
        # make sure the shuffled sequence does not lose any elements
        random.shuffle(self.seq)
        self.seq.sort()
        self.assertEqual(self.seq, list(range(10)))

    # should raise an exception for an immutable sequence
    self.assertRaises(TypeError, random.shuffle, (1,2,3))

    def test_choice(self):
        element = random.choice(self.seq)
        self.assertTrue(element in self.seq)

    def test_sample(self):
        with self.assertRaises(ValueError):
            random.sample(self.seq, 20)
        for element in random.sample(self.seq, 5):
            self.assertTrue(element in self.seq)
```
if __name__ == '__main__':
    unittest.main()

A testcase is created by subclassing unittest.TestCase. The three individual tests are defined with methods whose names start with the letters test. This naming convention informs the test runner about which methods represent tests.

The crux of each test is a call to assertEqual() to check for an expected result; assertTrue() to verify a condition; or assertRaises() to verify that an expected exception gets raised. These methods are used instead of the assert statement so the test runner can accumulate all test results and produce a report.

When a setUp() method is defined, the test runner will run that method prior to each test. Likewise, if a tearDown() method is defined, the test runner will invoke that method after each test. In the example, setUp() was used to create a fresh sequence for each test.

The final block shows a simple way to run the tests. unittest.main() provides a command-line interface to the test script. When run from the command line, the above script produces an output that looks like this:

```
... 
Ran 3 tests in 0.000s
OK
```

Passing the -v option to your test script will instruct unittest.main() to enable a higher level of verbosity, and produce the following output:

```
test_choice (__main__.TestSequenceFunctions) ... ok
test_sample (__main__.TestSequenceFunctions) ... ok
test_shuffle (__main__.TestSequenceFunctions) ... ok
```

```
Ran 3 tests in 0.110s
OK
```

The above examples show the most commonly used unittest features which are sufficient to meet many everyday testing needs. The remainder of the documentation explores the full feature set from first principles.

### 26.3.2 Command-Line Interface

The unittest module can be used from the command line to run tests from modules, classes or even individual test methods:

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

You can pass in a list with any combination of module names, and fully qualified class or method names.

Test modules can be specified by file path as well:

```
python -m unittest tests/test_something.py
```

This allows you to use the shell filename completion to specify the test module. The file specified must still be importable as a module. The path is converted to a module name by removing the ‘.py’ and converting path separators into ‘.’. If you want to execute a test file that isn’t importable as a module you should execute the file directly instead.

You can run tests with more detail (higher verbosity) by passing in the -v flag:

```
python -m unittest -v test_module
```

When executed without arguments Test Discovery is started:

```
python -m unittest
```
For a list of all the command-line options:

```bash
python -m unittest -h
```

Changed in version 3.2: In earlier versions it was only possible to run individual test methods and not modules or classes.

### Command-line options

`unittest` supports these command-line options:

- `-b`, `-buffer`  
The standard output and standard error streams are buffered during the test run. Output during a passing test is discarded. Output is echoed normally on test fail or error and is added to the failure messages.

- `-c`, `-catch`  
Control-C during the test run waits for the current test to end and then reports all the results so far. A second control-C raises the normal `KeyboardInterrupt` exception.

  See [Signal Handling](#) for the functions that provide this functionality.

- `-f`, `-failfast`  
Stop the test run on the first error or failure.

New in version 3.2: The command-line options `-b`, `-c` and `-f` were added. The command line can also be used for test discovery, for running all of the tests in a project or just a subset.

#### 26.3.3 Test Discovery

New in version 3.2. Unittest supports simple test discovery. In order to be compatible with test discovery, all of the test files must be modules or packages importable from the top-level directory of the project (this means that their filenames must be valid identifiers).

Test discovery is implemented in `TestLoader.discover()`, but can also be used from the command line. The basic command-line usage is:

```bash
cd project_directory
python -m unittest discover
```

**Note:** As a shortcut, `python -m unittest` is the equivalent of `python -m unittest discover`. If you want to pass arguments to test discovery the `discover` sub-command must be used explicitly.

The `discover` sub-command has the following options:

- `-v`, `-verbose`  
  Verbose output

- `-s`, `-start-directory directory`  
  Directory to start discovery (default)

- `-p`, `-pattern pattern`  
  Pattern to match test files (*_test.py default)

- `-t`, `-top-level-directory directory`  
  Top level directory of project (defaults to start directory)

The `-s`, `-p`, and `-t` options can be passed in as positional arguments in that order. The following two command lines are equivalent:

```bash
python -m unittest discover -s project_directory -p '*_test.py'
python -m unittest discover project_directory '*_test.py'
```
As well as being a path it is possible to pass a package name, for example `myproject.subpackage.test`, as the start directory. The package name you supply will then be imported and its location on the filesystem will be used as the start directory.

**Caution:** Test discovery loads tests by importing them. Once test discovery has found all the test files from the start directory you specify it turns the paths into package names to import. For example `foo/bar/baz.py` will be imported as `foo.bar.baz`.

If you have a package installed globally and attempt test discovery on a different copy of the package then the import *could* happen from the wrong place. If this happens test discovery will warn you and exit. If you supply the start directory as a package name rather than a path to a directory then discover assumes that whichever location it imports from is the location you intended, so you will not get the warning.

Test modules and packages can customize test loading and discovery by through the `load_tests` protocol.

### 26.3.4 Organizing test code

The basic building blocks of unit testing are *test cases* — single scenarios that must be set up and checked for correctness. In `unittest`, test cases are represented by `unittest.TestCase` instances. To make your own test cases you must write subclasses of `TestCase` or use `FunctionTestCase`.

The testing code of a `TestCase` instance should be entirely self contained, such that it can be run either in isolation or in arbitrary combination with any number of other test cases.

The simplest `TestCase` subclass will simply implement a test method (i.e. a method whose name starts with `test`) in order to perform specific testing code:

```python
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def test_default_widget_size(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50))
```

Note that in order to test something, we use one of the `assert*()` methods provided by the `TestCase` base class. If the test fails, an exception will be raised, and `unittest` will identify the test case as a *failure*. Any other exceptions will be treated as *errors*.

Tests can be numerous, and their set-up can be repetitive. Luckily, we can factor out set-up code by implementing a method called `setUp()`, which the testing framework will automatically call for every single test we run:

```python
import unittest

class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50), 'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150), 'wrong size after resize')
```

**Note:** The order in which the various tests will be run is determined by sorting the test method names with respect to the built-in ordering for strings.
If the `setUp()` method raises an exception while the test is running, the framework will consider the test to have suffered an error, and the test method will not be executed.

Similarly, we can provide a `tearDown()` method that tidies up after the test method has been run:

```python
import unittest

class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
```

If `setUp()` succeeded, `tearDown()` will be run whether the test method succeeded or not.

Such a working environment for the testing code is called a fixture.

Test case instances are grouped together according to the features they test. `unittest` provides a mechanism for this: the test suite, represented by `unittest`'s `TestSuite` class. In most cases, calling `unittest.main()` will do the right thing and collect all the module’s test cases for you, and then execute them.

However, should you want to customize the building of your test suite, you can do it yourself:

```python
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTest('test_default_size'))
    suite.addTest(WidgetTest('test_resize'))
    return suite
```

You can place the definitions of test cases and test suites in the same modules as the code they are to test (such as `widget.py`), but there are several advantages to placing the test code in a separate module, such as `test_widget.py`:

- The test module can be run standalone from the command line.
- The test code can more easily be separated from shipped code.
- There is less temptation to change test code to fit the code it tests without a good reason.
- Test code should be modified much less frequently than the code it tests.
- Tested code can be refactored more easily.
- Tests for modules written in C must be in separate modules anyway, so why not be consistent?
- If the testing strategy changes, there is no need to change the source code.

### 26.3.5 Re-using old test code

Some users will find that they have existing test code that they would like to run from `unittest`, without converting every old test function to a `TestCase` subclass.

For this reason, `unittest` provides a `FunctionTestCase` class. This subclass of `TestCase` can be used to wrap an existing test function. Set-up and tear-down functions can also be provided.

Given the following test function:

```python
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

one can create an equivalent test case instance as follows, with optional set-up and tear-down methods:
testcase = unittest.FunctionTestCase(testSomething,
     setUp=makeSomethingDB,
     tearDown=deleteSomethingDB)

**Note:** Even though FunctionTestCase can be used to quickly convert an existing test base over to a unittest-based system, this approach is not recommended. Taking the time to set up proper TestCase subclasses will make future test refactorings infinitely easier.

In some cases, the existing tests may have been written using the doctest module. If so, doctest provides a DocTestSuite class that can automatically build unittest.TestSuite instances from the existing doctest-based tests.

### 26.3.6 Skipping tests and expected failures

New in version 3.1. Unittest supports skipping individual test methods and even whole classes of tests. In addition, it supports marking a test as a "expected failure," a test that is broken and will fail, but shouldn’t be counted as a failure on a TestResult.

Skipping a test is simply a matter of using the `skip()` decorator or one of its conditional variants.

Basic skipping looks like this:

```python
class MyTestCase(unittest.TestCase):
    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn’t happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass
```

This is the output of running the example above in verbose mode:

```
test_format (__main__.MyTestCase) ... skipped ’not supported in this library version’
test_nothing (__main__.MyTestCase) ... skipped ’demonstrating skipping’
test_windows_support (__main__.MyTestCase) ... skipped ’requires Windows’

Ran 3 tests in 0.005s
```

OK (skipped=3)

Classes can be skipped just like methods:

```python
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

Expected failures use the `expectedFailure()` decorator.
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")

It’s easy to roll your own skipping decorators by making a decorator that calls `skip()` on the test when it wants it to be skipped. This decorator skips the test unless the passed object has a certain attribute:

```python
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn’t have {!r}".format(obj, attr))
```

The following decorators implement test skipping and expected failures:

```python
@unittest.skip(reason)
    Unconditionally skip the decorated test. reason should describe why the test is being skipped.

@unittest.skipIf(condition, reason)
    Skip the decorated test if condition is true.

@unittest.skipUnless(condition, reason)
    Skip the decorated test unless condition is true.

@unittest.expectedFailure
    Mark the test as an expected failure. If the test fails when run, the test is not counted as a failure.
```

### exception unittest.SkipTest(reason)

This exception is raised to skip a test.

Usually you can use `TestCase.skipTest()` or one of the skipping decorators instead of raising this directly.

Skipped tests will not have `setUp()` or `tearDown()` run around them. Skipped classes will not have `setUpClass()` or `tearDownClass()` run.

### 26.3.7 Classes and functions

This section describes in depth the API of `unittest`.

#### Test cases

```python
class unittest.TestCase (methodName='runTest')
```

Instances of the `TestCase` class represent the logical test units in the `unittest` universe. This class is intended to be used as a base class, with specific tests being implemented by concrete subclasses. This class implements the interface needed by the test runner to allow it to drive the tests, and methods that the test code can use to check for and report various kinds of failure.

Each instance of `TestCase` will run a single base method: the method named `methodName`. However, the standard implementation of the default `methodName`, `runTest()`, will run every method starting with `test` as an individual test, and count successes and failures accordingly. Therefore, in most uses of `TestCase`, you will neither change the `methodName` nor reimplement the default `runTest()` method. Changed in version 3.2: `TestCase` can be instantiated successfully without providing a `methodName`. This makes it easier to experiment with `TestCase` from the interactive interpreter. `TestCase` instances provide three groups of methods: one group used to run the test, another used by the test implementation to check conditions and report failures, and some inquiry methods allowing information about the test itself to be gathered.

Methods in the first group (running the test) are:

```python
setUp()
```

Method called to prepare the test fixture. This is called immediately before calling the test method;
any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

tearDown()
Method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception, so the implementation in subclasses may need to be particularly careful about checking internal state. Any exception raised by this method will be considered an error rather than a test failure. This method will only be called if the setUp() succeeds, regardless of the outcome of the test method. The default implementation does nothing.

setUpClass()
A class method called before tests in an individual class run. setUpClass is called with the class as the only argument and must be decorated as a classmethod():

```python
@classmethod
def setUpClass(cls):
    ...
```

See Class and Module Fixtures for more details. New in version 3.2.

tearDownClass()
A class method called after tests in an individual class have run. tearDownClass is called with the class as the only argument and must be decorated as a classmethod():

```python
@classmethod
def tearDownClass(cls):
    ...
```

See Class and Module Fixtures for more details. New in version 3.2.

run(result=None)
Run the test, collecting the result into the TestResult object passed as result. If result is omitted or None, a temporary result object is created (by calling the defaultTestResult() method) and used. The result object is returned to run()’s caller.

The same effect may be had by simply calling theTestCaseinstance. Changed in version 3.3: Previous versions of run did not return the result. Neither did calling an instance.

skipTest(reason)
Calling this during a test method or setUp() skips the current test. See Skipping tests and expected failures for more information. New in version 3.1.

debug()
Run the test without collecting the result. This allows exceptions raised by the test to be propagated to the caller, and can be used to support running tests under a debugger.

TheTestCaseclass provides a number of methods to check for and report failures, such as:

<table>
<thead>
<tr>
<th>Method</th>
<th>Checks that</th>
<th>New in</th>
</tr>
</thead>
<tbody>
<tr>
<td>assertEqual(a, b)</td>
<td>a == b</td>
<td></td>
</tr>
<tr>
<td>assertNotEqual(a, b)</td>
<td>a != b</td>
<td></td>
</tr>
<tr>
<td>assertTrue(x)</td>
<td>bool(x) is True</td>
<td></td>
</tr>
<tr>
<td>assertFalse(x)</td>
<td>bool(x) is False</td>
<td></td>
</tr>
<tr>
<td>assertIs(a, b)</td>
<td>a is b</td>
<td>3.1</td>
</tr>
<tr>
<td>assertIsNot(a, b)</td>
<td>a is not b</td>
<td>3.1</td>
</tr>
<tr>
<td>assertIsNone(x)</td>
<td>x is None</td>
<td>3.1</td>
</tr>
<tr>
<td>assertIsNotNone(x)</td>
<td>x is not None</td>
<td>3.1</td>
</tr>
<tr>
<td>assertIn(a, b)</td>
<td>a in b</td>
<td>3.1</td>
</tr>
<tr>
<td>assertNotIn(a, b)</td>
<td>a not in b</td>
<td>3.1</td>
</tr>
<tr>
<td>assertIsInstance(a, b)</td>
<td>isinstance(a, b)</td>
<td>3.2</td>
</tr>
<tr>
<td>assertNotIsInstance(a, b)</td>
<td>not isinstance(a, b)</td>
<td>3.2</td>
</tr>
</tbody>
</table>
All the assert methods accept a `msg` argument that, if specified, is used as the error message on failure (see also `longMessage`). Note that the `msg` keyword argument can be passed to `assertRaises()`, `assertRaisesRegex()`, `assertWarns()`, `assertWarnsRegex()` only when they are used as a context manager.

**assertEqual** `(first, second, msg=None)`

Test that `first` and `second` are equal. If the values do not compare equal, the test will fail.

In addition, if `first` and `second` are the exact same type and one of list, tuple, dict, set, frozenset or str or any type that a subclass registers with `addTypeEqualityFunc()` the type-specific equality function will be called in order to generate a more useful default error message (see also the list of type-specific methods). Changed in version 3.1: Added the automatic calling of type-specific equality function. Changed in version 3.2: `assertMultilineEqual()` added as the default type equality function for comparing strings.

**assertNotEqual** `(first, second, msg=None)`

Test that `first` and `second` are not equal. If the values do compare equal, the test will fail.

**assertTrue** `(expr, msg=None)`

**assertFalse** `(expr, msg=None)`

Test that `expr` is true (or false).

Note that this is equivalent to `bool(expr)` is True and not to `expr is True` (use `assertIs(expr, True)` for the latter). This method should also be avoided when more specific methods are available (e.g. `assertEqual(a, b)` instead of `assertTrue(a == b)`), because they provide a better error message in case of failure.

**assertIs** `(first, second, msg=None)`

**assertIsNot** `(first, second, msg=None)`

Test that `first` and `second` evaluate (or don’t evaluate) to the same object. New in version 3.1.

**assertIsNone** `(expr, msg=None)`

**assertIsNotNone** `(expr, msg=None)`

Test that `expr` is (or is not) None. New in version 3.1.

**assertIn** `(first, second, msg=None)`

**assertNotIn** `(first, second, msg=None)`

Test that `first` is (or is not) in `second`. New in version 3.1.

**assertIsInstance** `(obj, cls, msg=None)`

**assertNotIsInstance** `(obj, cls, msg=None)`

Test that `obj` is (or is not) an instance of `cls` (which can be a class or a tuple of classes, as supported by `isinstance()`). To check for the exact type, use `assertIs(type(obj), cls)`. New in version 3.2.

It is also possible to check that exceptions and warnings are raised using the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Checks that</th>
<th>New in</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assertRaises(exc, fun, *args, **kwd)</code></td>
<td><code>fun(*args, **kwd)</code> raises <code>exc</code></td>
<td>3.1</td>
</tr>
<tr>
<td><code>assertRaisesRegex(exc, r, fun, *args, **kwd)</code></td>
<td><code>fun(*args, **kwd)</code> raises <code>exc</code> and the message matches regex <code>r</code></td>
<td>3.1</td>
</tr>
<tr>
<td><code>assertWarns(warn, fun, *args, **kwd)</code></td>
<td><code>fun(*args, **kwd)</code> raises <code>warn</code></td>
<td>3.2</td>
</tr>
<tr>
<td><code>assertWarnsRegex(warn, r, fun, *args, **kwd)</code></td>
<td><code>fun(*args, **kwd)</code> raises <code>warn</code> and the message matches regex <code>r</code></td>
<td>3.2</td>
</tr>
</tbody>
</table>

**assertRaises** `(exception, callable, *args, **kwd)`

Test that an exception is raised when `callable` is called with any positional or keyword arguments that are also passed to `assertRaises()`. The test passes if `exception` is raised, is an error if another exception is raised, or fails if no exception is raised. To catch any of a group of exceptions, a tuple containing the exception classes may be passed as `exception`.
If only the `exception` and possibly the `msg` arguments are given, return a context manager so that the code under test can be written inline rather than as a function:

```python
with self.assertRaises(SomeException):
    do_something()
```

When used as a context manager, `assertRaises()` accepts the additional keyword argument `msg`. The context manager will store the caught exception object in its `exception` attribute. This can be useful if the intention is to perform additional checks on the exception raised:

```python
with self.assertRaises(SomeException) as cm:
    do_something()
the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

Changed in version 3.1: Added the ability to use `assertRaises()` as a context manager. Changed in version 3.2: Added the `exception` attribute. Changed in version 3.3: Added the `msg` keyword argument when used as a context manager.

```python
assertRaisesRegex(exception, regex, callable, *args, **kwds)
```

Like `assertRaises()` but also tests that `regex` matches on the string representation of the raised exception. `regex` may be a regular expression object or a string containing a regular expression suitable for use by `re.search()`. Examples:

```python
self.assertRaisesRegex(ValueError, "invalid literal for.*XYZ'\$", int, 'XYZ')
```

or:

```python
with self.assertRaisesRegex(ValueError, 'literal'):
    int('XYZ')
```

New in version 3.1: under the name `assertRaisesRegex`.

```python
assertWarns(warning, callable, *args, **kwds)
```

Test that a warning is triggered when `callable` is called with any positional or keyword arguments that are also passed to `assertWarns()`. The test passes if `warning` is triggered and fails if it isn’t. Any exception is an error. To catch any of a group of warnings, a tuple containing the warning classes may be passed as `warnings`.

If only the `warning` and possibly the `msg` arguments are given, return a context manager so that the code under test can be written inline rather than as a function:

```python
with self.assertWarns(SomeWarning):
    do_something()
```

When used as a context manager, `assertWarns()` accepts the additional keyword argument `msg`. The context manager will store the caught warning object in its `warning` attribute, and the source line which triggered the warnings in the `filename` and `lineno` attributes. This can be useful if the intention is to perform additional checks on the warning caught:

```python
with self.assertWarns(SomeWarning) as cm:
    do_something()
```
self.assertIn('myfile.py', cm.filename)
self.assertEqual(320, cm.lineno)

This method works regardless of the warning filters in place when it is called. New in version 3.2. Changed in version 3.3: Added the msg keyword argument when used as a context manager.

assertWarnsRegex (warning, regex, callable, *args, **kwds)
assertWarnsRegex (warning, regex, msg=None)

Like assertWarns() but also tests that regex matches on the message of the triggered warning. regex may be a regular expression object or a string containing a regular expression suitable for use by re.search(). Example:

    self.assertWarnsRegex(DeprecationWarning,
                         r'legacy_function\(\) is deprecated',
                         legacy_function, 'XYZ')

or:

    with self.assertWarnsRegex(RuntimeWarning, 'unsafe frobnicating'):
        frobnicate('/etc/passwd')

New in version 3.2. Changed in version 3.3: Added the msg keyword argument when used as a context manager.

There are also other methods used to perform more specific checks, such as:

<table>
<thead>
<tr>
<th>Method</th>
<th>Checks that</th>
<th>New in</th>
</tr>
</thead>
<tbody>
<tr>
<td>assertAlmostEqual(a, b)</td>
<td>( round(a-b, 7) == 0 )</td>
<td></td>
</tr>
<tr>
<td>assertNotAlmostEqual(a, b)</td>
<td>( round(a-b, 7) != 0 )</td>
<td></td>
</tr>
<tr>
<td>assertGreater(a, b)</td>
<td>( a &gt; b )</td>
<td>3.1</td>
</tr>
<tr>
<td>assertGreaterEqual(a, b)</td>
<td>( a &gt;= b )</td>
<td>3.1</td>
</tr>
<tr>
<td>assertLess(a, b)</td>
<td>( a &lt; b )</td>
<td>3.1</td>
</tr>
<tr>
<td>assertLessEqual(a, b)</td>
<td>( a &lt;= b )</td>
<td>3.1</td>
</tr>
<tr>
<td>assertRegex(s, r)</td>
<td>( r.search(s) )</td>
<td>3.1</td>
</tr>
<tr>
<td>assertNotRegex(s, r)</td>
<td>( not r.search(s) )</td>
<td>3.2</td>
</tr>
<tr>
<td>assertCountEqual(a, b)</td>
<td>( a ) and ( b ) have the same elements in the same number, regardless of their order</td>
<td>3.2</td>
</tr>
</tbody>
</table>

assertAlmostEqual (first, second, places=7, msg=None, delta=None)
assertNotAlmostEqual (first, second, places=7, msg=None, delta=None)

Test that first and second are approximately (or not approximately) equal by computing the difference, rounding to the given number of decimal places (default 7), and comparing to zero. Note that these methods round the values to the given number of decimal places (i.e. like the round() function) and not significant digits.

If delta is supplied instead of places then the difference between first and second must be less or equal to (or greater than) delta.

Supplying both delta and places raises a TypeError. Changed in version 3.2: assertAlmostEqual() automatically considers almost equal objects that compare equal. assertNotAlmostEqual() automatically fails if the objects compare equal. Added the delta keyword argument.

assertGreater (first, second, msg=None)
assertGreaterEqual (first, second, msg=None)
assertLess (first, second, msg=None)
assertLessEqual (first, second, msg=None)
Test that first is respectively >, >=, < or <= than second depending on the method name. If not, the test will fail:

```python
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or equal to "4"
```

New in version 3.1.

assertRegex (text, regex, msg=None)
assertNotRegex (text, regex, msg=None)
Test that a regex search matches (or does not match) text. In case of failure, the error message will include the pattern and the text (or the pattern and the part of text that unexpectedly matched). regex may be a regular expression object or a string containing a regular expression suitable for use by `re.search()`. New in version 3.1: under the name `assertRegexpMatches()`. Changed in version 3.2: The method `assertRegexpMatches()` has been renamed to `assertRegex()`. New in version 3.2: `assertNotRegex()`.

assertCountEqual (first, second, msg=None)
Test that sequence first contains the same elements as second, regardless of their order. When they don’t, an error message listing the differences between the sequences will be generated.

Duplicate elements are not ignored when comparing first and second. It verifies whether each element has the same count in both sequences. Equivalent to: `assertEqual(Counter(list(first)), Counter(list(second)))` but works with sequences of unhashable objects as well. New in version 3.2.

The `assertEqual()` method dispatches the equality check for objects of the same type to different type-specific methods. These methods are already implemented for most of the built-in types, but it’s also possible to register new methods using `addTypeEqualityFunc()`:

```python
addTypeEqualityFunc (typeobj, function)
Registers a type-specific method called by `assertEqual()` to check if two objects of exactly the same typeobj (not subclasses) compare equal. function must take two positional arguments and a third msg=None keyword argument just as `assertEqual()` does. It must raise `self.failureException(msg)` when inequality between the first two parameters is detected – possibly providing useful information and explaining the inequalities in details in the error message. New in version 3.1.
```

The list of type-specific methods automatically used by `assertEqual()` are summarized in the following table. Note that it’s usually not necessary to invoke these methods directly.

<table>
<thead>
<tr>
<th>Method</th>
<th>Used to compare</th>
<th>New in</th>
</tr>
</thead>
<tbody>
<tr>
<td>assertMultiLineEqual (a, b)</td>
<td>strings</td>
<td>3.1</td>
</tr>
<tr>
<td>assertSequenceEqual (a, b)</td>
<td>sequences</td>
<td>3.1</td>
</tr>
<tr>
<td>assertListEqual (a, b)</td>
<td>lists</td>
<td>3.1</td>
</tr>
<tr>
<td>assertTupleEqual (a, b)</td>
<td>tuples</td>
<td>3.1</td>
</tr>
<tr>
<td>assertSetEqual (a, b)</td>
<td>sets or frozensets</td>
<td>3.1</td>
</tr>
<tr>
<td>assertDictEqual (a, b)</td>
<td>dicts</td>
<td>3.1</td>
</tr>
</tbody>
</table>

assertMultiLineEqual (first, second, msg=None)
Test that the multiline string first is equal to the string second. When not equal a diff of the two strings highlighting the differences will be included in the error message. This method is used by default when comparing strings with `assertEqual()`. New in version 3.1.

assertSequenceEqual (first, second, msg=None, seq_type=None)
Tests that two sequences are equal. If a seq_type is supplied, both first and second must be instances of seq_type or a failure will be raised. If the sequences are different an error message is constructed that shows the difference between the two.

This method is not called directly by `assertEqual()`, but it’s used to implement `assertListEqual()` and `assertTupleEqual()`. New in version 3.1.

assertListEqual (first, second, msg=None)
assertTupleEqual (first, second, msg=None)
Tests that two lists or tuples are equal. If not, an error message is constructed that shows only the
differences between the two. An error is also raised if either of the parameters are of the wrong type.
These methods are used by default when comparing lists or tuples with assertEqual(). New in
version 3.1.

assertSetEqual (first, second, msg=None)
Tests that two sets are equal. If not, an error message is constructed that lists the differences between
the sets. This method is used by default when comparing sets or frozensets with assertEqual().
Fails if either of first or second does not have a set.difference() method. New in version 3.1.

assertDictEqual (first, second, msg=None)
Test that two dictionaries are equal. If not, an error message is constructed that shows the differ-
ences in the dictionaries. This method will be used by default to compare dictionaries in calls to
assertEqual(). New in version 3.1.

Finally the TestCase provides the following methods and attributes:

fail (msg=None)
Signals a test failure unconditionally, with msg or None for the error message.

failureException
This class attribute gives the exception raised by the test method. If a test framework needs to use a
specialized exception, possibly to carry additional information, it must subclass this exception in order
to “play fair” with the framework. The initial value of this attribute is AssertionError.

longMessage
If set to True then any explicit failure message you pass in to the assert methods will be appended
to the end of the normal failure message. The normal messages contain useful information about the
objects involved, for example the message from assertEqual shows you the repr of the two unequal
objects. Setting this attribute to True allows you to have a custom error message in addition to the
normal one.

This attribute defaults to True. If set to False then a custom message passed to an assert method will
silence the normal message.

The class setting can be overridden in individual tests by assigning an instance attribute to True or
False before calling the assert methods. New in version 3.1.

maxDiff
This attribute controls the maximum length of diffs output by assert methods that report diffs
on failure. It defaults to 80*8 characters. Assert methods affected by this attribute are
assertSequenceEqual() (including all the sequence comparison methods that delegate to it),
assertDictEqual() and assertMultiLineEqual().

Setting maxDiff to None means that there is no maximum length of diffs. New in version 3.2.

Testing frameworks can use the following methods to collect information on the test:

countTestCases ()
Return the number of tests represented by this test object. For TestCase instances, this will always
be 1.

defaultTestResult ()
Return an instance of the test result class that should be used for this test case class (if no other result
instance is provided to the run() method).

For TestCase instances, this will always be an instance of TestResult; subclasses of TestCase
should override this as necessary.

id ()
Return a string identifying the specific test case. This is usually the full name of the test method,
including the module and class name.
shortDescription()

Returns a description of the test, or None if no description has been provided. The default implementation of this method returns the first line of the test method’s docstring, if available, or None. Changed in version 3.1: In 3.1 this was changed to add the test name to the short description even in the presence of a docstring. This caused compatibility issues with unittest extensions and adding the test name was moved to the TextTestResult in Python 3.2.

addCleanup(function, *args, **kwargs)

Add a function to be called after tearDown() to cleanup resources used during the test. Functions will be called in reverse order to the order they are added (LIFO). They are called with any arguments and keyword arguments passed into addCleanup() when they are added.

If setUp() fails, meaning that tearDown() is not called, then any cleanup functions added will still be called. New in version 3.1.

doCleanups()

This method is called unconditionally after tearDown(), or after setUp() if setUp() raises an exception.

It is responsible for calling all the cleanup functions added by addCleanup(). If you need cleanup functions to be called prior to tearDown() then you can call doCleanups() yourself.

doCleanups() pops methods off the stack of cleanup functions one at a time, so it can be called at any time. New in version 3.1.

class unittest.FunctionTestCase(testFunc, setUp=None, tearDown=None, description=None)

This class implements the portion of the TestCase interface which allows the test runner to drive the test, but does not provide the methods which test code can use to check and report errors. This is used to create test cases using legacy test code, allowing it to be integrated into a unittest-based test framework.

Deprecated aliases

For historical reasons, some of the TestCase methods had one or more aliases that are now deprecated. The following table lists the correct names along with their deprecated aliases:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Deprecated alias</th>
<th>Deprecated alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>assertEqual()</td>
<td>failUnlessEqual</td>
<td>assertEquality</td>
</tr>
<tr>
<td>assertNotEqual()</td>
<td>failIfEqual</td>
<td>assertNotEqual</td>
</tr>
<tr>
<td>assertTrue()</td>
<td>failUnless</td>
<td>assert_</td>
</tr>
<tr>
<td>assertFalse()</td>
<td>failIf</td>
<td></td>
</tr>
<tr>
<td>assertRaises()</td>
<td>failUnlessRaises</td>
<td>assertAlmostEqual</td>
</tr>
<tr>
<td>assertAlmostEqual()</td>
<td>failIfAlmostEqual</td>
<td>assertNotAlmostEqual</td>
</tr>
<tr>
<td>assertNotAlmostEqual()</td>
<td></td>
<td>assertAlmostEquals</td>
</tr>
<tr>
<td>assertNotEquals()</td>
<td></td>
<td>assertNotAlmostEqual</td>
</tr>
<tr>
<td>assertRegex()</td>
<td></td>
<td>assertRegexMatches</td>
</tr>
<tr>
<td>assertRaisesRegex()</td>
<td></td>
<td>assertRaisesRegexp</td>
</tr>
</tbody>
</table>

Deprecated since version 3.1: the fail* aliases listed in the second column. Deprecated since version 3.2: the assert* aliases listed in the third column. Deprecated since version 3.2: assertRegexMatches and assertRaisesRegex have been renamed to assertRegex() and assertRaisesRegex()

Grouping tests

class unittest.TestSuite(tests=())

This class represents an aggregation of individual tests cases and test suites. The class presents the interface needed by the test runner to allow it to be run as any other test case. Running a TestSuite instance is the same as iterating over the suite, running each test individually.

If tests is given, it must be an iterable of individual test cases or other test suites that will be used to build the suite initially. Additional methods are provided to add test cases and suites to the collection later on.
TestSuite objects behave much like TestCase objects, except they do not actually implement a test. Instead, they are used to aggregate tests into groups of tests that should be run together. Some additional methods are available to add tests to TestSuite instances:

**addTest**(test)
Add aTestCase or TestSuite to the suite.

**addTests**(tests)
Add all the tests from an iterable of TestCase and TestSuite instances to this test suite.

This is equivalent to iterating over tests, calling addTest() for each element.

TestSuite shares the following methods with TestCase:

**run**(result)
Run the tests associated with this suite, collecting the result into the test result object passed as result. Note that unlikeTestCase.run(), TestSuite.run() requires the result object to be passed in.

**debug**( )
Run the tests associated with this suite without collecting the result. This allows exceptions raised by the test to be propagated to the caller and can be used to support running tests under a debugger.

**countTestCases**( )
Return the number of tests represented by this test object, including all individual tests and sub-suites.

**__iter__**( )
Tests grouped by a TestSuite are always accessed by iteration. Subclasses can lazily provide tests by overriding __iter__(). Note that this method may be called several times on a single suite (for example when counting tests or comparing for equality) so the tests returned must be the same for repeated iterations. Changed in version 3.2: In earlier versions the TestSuite accessed tests directly rather than through iteration, so overriding __iter__() wasn’t sufficient for providing tests.

In the typical usage of a TestSuite object, the run() method is invoked by a TestRunner rather than by the end-user test harness.

**Loading and running tests**

**class unittest.TestLoader**
The TestLoader class is used to create test suites from classes and modules. Normally, there is no need to create an instance of this class; the unittest module provides an instance that can be shared as unittest.defaultTestLoader. Using a subclass or instance, however, allows customization of some configurable properties.

TestLoader objects have the following methods:

**loadTestsFromTestCase**(testCaseClass)
Return a suite of all tests cases contained in theTestCase-derived testCaseClass.

**loadTestsFromModule**(module)
Return a suite of all tests cases contained in the given module. This method searches module for classes derived fromTestCase and creates an instance of the class for each test method defined for the class.

**Note:** While using a hierarchy ofTestCase-derived classes can be convenient in sharing fixtures and helper functions, defining test methods on base classes that are not intended to be instantiated directly does not play well with this method. Doing so, however, can be useful when the fixtures are different and defined in subclasses.

If a module provides a load_tests function it will be called to load the tests. This allows modules to customize test loading. This is the load_tests protocol. Changed in version 3.2: Support for load_tests added.
loadTestsFromName\(\text{name, module=\text{None}}\)

Return a suite of all tests cases given a string specifier.

The specifier \text{name} is a “dotted name” that may resolve either to a module, a test case class, a test method within a test case class, a \text{TestSuite} instance, or a callable object which returns a \text{TestCase} or \text{TestSuite} instance. These checks are applied in the order listed here; that is, a method on a possible test case class will be picked up as “a test method within a test case class”, rather than “a callable object”.

For example, if you have a module \text{SampleTests} containing a \text{TestCase}-derived class \text{SampleTestCase} with three test methods (\text{test_one()}, \text{test_two()}, and \text{test_three()}), the specifier ‘\text{SampleTests.SampleTestCase}’ would cause this method to return a suite which will run all three test methods. Using the specifier ‘\text{SampleTests.SampleTestCase.test_two}’ would cause it to return a test suite which will run only the \text{test_two()} test method. The specifier can refer to modules and packages which have not been imported; they will be imported as a side-effect.

The method optionally resolves \text{name} relative to the given \text{module}.

loadTestsFromNames\(\text{\textit{names, module=\text{None}}}\)

Similar to \text{loadTestsFromName()}, but takes a sequence of names rather than a single name. The return value is a test suite which supports all the tests defined for each name.

getTestCaseNames\(\text{\textit{testCaseClass}}\)

Return a sorted sequence of method names found within \text{testCaseClass}; this should be a subclass of \text{TestCase}.

discover\(\text{\textit{start_dir, pattern='test*.py', top_level_dir=None}}\)

Find and return all test modules from the specified start directory, recursing into subdirectories to find them. Only test files that match \text{pattern} will be loaded. (Using shell style pattern matching.) Only module names that are importable (i.e. are valid Python identifiers) will be loaded.

All test modules must be importable from the top level of the project. If the start directory is not the top level directory then the top level directory must be specified separately.

If importing a module fails, for example due to a syntax error, then this will be recorded as a single error and discovery will continue.

If a test package name (directory with \text{__init__.py}) matches the pattern then the package will be checked for a \text{load_tests} function. If this exists then it will be called with \text{loader, tests, pattern}.

If \text{load_tests} exists then discovery does \text{not} recurse into the package, \text{load_tests} is responsible for loading all tests in the package.

The pattern is deliberately not stored as a loader attribute so that packages can continue discovery themselves. \text{top_level_dir} is stored so \text{load_tests} does not need to pass this argument in to \text{loader.discover()}.  \text{start_dir} can be a dotted module name as well as a directory. New in version 3.2.

The following attributes of a \text{TestLoader} can be configured either by subclassing or assignment on an instance:

\text{testMethodPrefix}

String giving the prefix of method names which will be interpreted as test methods. The default value is ‘test’.

This affects \text{getTestCaseNames()} and all the \text{loadTestsFrom*()} methods.

\text{sortTestMethodsUsing}

Function to be used to compare method names when sorting them in \text{getTestCaseNames()} and all the \text{loadTestsFrom*()} methods.

\text{suiteClass}

Callable object that constructs a test suite from a list of tests. No methods on the resulting object are needed. The default value is the \text{TestSuite} class.
This affects all the `loadTestsFrom*()` methods.

**Class: unittest.TestCase**

This class is used to compile information about which tests have succeeded and which have failed.

A `TestResult` object stores the results of a set of tests. The `TestCase` and `TestSuite` classes ensure that results are properly recorded; test authors do not need to worry about recording the outcome of tests.

Testing frameworks built on top of `unittest` may want access to the `TestResult` object generated by running a set of tests for reporting purposes; a `TestResult` instance is returned by the `TestRunner.run()` method for this purpose.

`TestResult` instances have the following attributes that will be of interest when inspecting the results of running a set of tests:

- **errors**
  A list containing 2-tuples of `TestCase` instances and strings holding formatted tracebacks. Each tuple represents a test which raised an unexpected exception.

- **failures**
  A list containing 2-tuples of `TestCase` instances and strings holding formatted tracebacks. Each tuple represents a test where a failure was explicitly signalled using the `TestCase.assert*()` methods.

- **skipped**
  A list containing 2-tuples of `TestCase` instances and strings holding the reason for skipping the test. New in version 3.1.

- **expectedFailures**
  A list containing 2-tuples of `TestCase` instances and strings holding formatted tracebacks. Each tuple represents an expected failure of the test case.

- **unexpectedSuccesses**
  A list containing `TestCase` instances that were marked as expected failures, but succeeded.

- **shouldStop**
  Set to `True` when the execution of tests should stop by `stop()`.

- **testsRun**
  The total number of tests run so far.

- **buffer**
  If set to `True`, `sys.stdout` and `sys.stderr` will be buffered in between `startTest()` and `stopTest()` being called. Collected output will only be echoed onto the real `sys.stdout` and `sys.stderr` if the test fails or errors. Any output is also attached to the failure/error message. New in version 3.2.

- **failfast**
  If set to `True` `stop()` will be called on the first failure or error, halting the test run. New in version 3.2.

- **wasSuccessful()**
  Return `True` if all tests run so far have passed, otherwise returns `False`.

- **stop()**
  This method can be called to signal that the set of tests being run should be aborted by setting the `shouldStop` attribute to `True`. `TestRunner` objects should respect this flag and return without running any additional tests.

  For example, this feature is used by the `TextTestRunner` class to stop the test framework when the user signals an interrupt from the keyboard. Interactive tools which provide `TestRunner` implementations can use this in a similar manner.

The following methods of the `TestResult` class are used to maintain the internal data structures, and may be extended in subclasses to support additional reporting requirements. This is particularly useful in building tools which support interactive reporting while tests are being run.
**startTest** *(test)*  
Called when the test case *test* is about to be run.

**stopTest** *(test)*  
Called after the test case *test* has been executed, regardless of the outcome.

**startTestRun** *(test)*  
Called once before any tests are executed. New in version 3.1.

**stopTestRun** *(test)*  
Called once after all tests are executed. New in version 3.1.

**addError** *(test, err)*  
Called when the test case *test* raises an unexpected exception. *err* is a tuple of the form returned by `sys.exc_info()`: `(type, value, traceback)`.  
The default implementation appends a tuple *(test, formatted_err)* to the instance’s `errors` attribute, where `formatted_err` is a formatted traceback derived from `err`.

**addFailure** *(test, err)*  
Called when the test case *test* signals a failure. *err* is a tuple of the form returned by `sys.exc_info()`: `(type, value, traceback)`.  
The default implementation appends a tuple *(test, formatted_err)* to the instance’s `failures` attribute, where `formatted_err` is a formatted traceback derived from `err`.

**addSuccess** *(test)*  
Called when the test case *test* succeeds.  
The default implementation does nothing.

**addSkip** *(test, reason)*  
Called when the test case *test* is skipped. *reason* is the reason the test gave for skipping.  
The default implementation appends a tuple *(test, reason)* to the instance’s `skipped` attribute.

**addExpectedFailure** *(test, err)*  
Called when the test case *test* fails, but was marked with the `expectedFailure()` decorator.  
The default implementation appends a tuple *(test, formatted_err)* to the instance’s `expectedFailures` attribute, where `formatted_err` is a formatted traceback derived from `err`.

**addUnexpectedSuccess** *(test)*  
Called when the test case *test* was marked with the `expectedFailure()` decorator, but succeeded.  
The default implementation appends the test to the instance’s `unexpectedSuccesses` attribute.

**class unittest.TextTestResult** *(stream, descriptions, verbosity)*  
A concrete implementation of `TestResult` used by the `TextTestRunner`. New in version 3.2: This class was previously named `_TextTestResult`. The old name still exists as an alias but is deprecated.

**unittest.defaultTestLoader**  
Instance of the `TestLoader` class intended to be shared. If no customization of the `TestLoader` is needed, this instance can be used instead of repeatedly creating new instances.

**class unittest.TextTestRunner** *(stream=None, descriptions=True, verbosity=1, failfast=False, buffer=False, resultclass=None, warnings=None)*  
A basic test runner implementation that outputs results to a stream. If *stream* is `None`, the default, `sys.stderr` is used as the output stream. This class has a few configurable parameters, but is essentially very simple. Graphical applications which run test suites should provide alternate implementations.

By default this runner shows `DeprecationWarning`, `PendingDeprecationWarning`, and `ImportWarning` even if they are ignored by default. Deprecation warnings caused by `deprecated` `unittest` methods are also special-cased and, when the warning filters are ‘default’ or ‘always’, they will appear only once per-module, in order to avoid too many warning messages. This behavior can be overridden using the `-Wd` or `-Wa` options and leaving `warnings` to `None`. Changed in version 3.2: Added the `warnings` argument. Changed in version 3.2: The default stream is set to `sys.stderr` at instantiation time rather than import time.
_makeResult()

This method returns the instance of TestResult used by run(). It is not intended to be called directly, but can be overridden in subclasses to provide a custom TestResult.

_makeResult() instantiates the class or callable passed in the TextTestRunner constructor as the resultclass argument. It defaults to TextTestResult if no resultclass is provided. The result class is instantiated with the following arguments:

stream, descriptions, verbosity

unittest.main(module='__main__', defaultTest=None, argv=None, testRunner=None, testLoader=unittest.defaultTestLoader, exit=True, verbosity=1, failfast=None, catchbreak=None, buffer=None, warnings=None)

A command-line program that loads a set of tests from module and runs them; this is primarily for making test modules conveniently executable. The simplest use for this function is to include the following line at the end of a test script:

if __name__ == '__main__':
    unittest.main()

You can run tests with more detailed information by passing in the verbosity argument:

if __name__ == '__main__':
    unittest.main(verbosity=2)

The argv argument can be a list of options passed to the program, with the first element being the program name. If not specified or None, the values of sys.argv are used.

The testRunner argument can either be a test runner class or an already created instance of it. By default main calls sys.exit() with an exit code indicating success or failure of the tests run.

The testLoader argument has to be a TestLoader instance, and defaults to defaultTestLoader. main supports being used from the interactive interpreter by passing in the argument exit=False. This displays the result on standard output without calling sys.exit():

>>> from unittest import main
>>> main(module='test_module', exit=False)

The failfast, catchbreak and buffer parameters have the same effect as the same-name command-line options.

The warning argument specifies the warning filter that should be used while running the tests. If it’s not specified, it will remain None if a -W option is passed to python, otherwise it will be set to ‘default’.

Calling main actually returns an instance of the TestProgram class. This stores the result of the tests run as the result attribute. Changed in version 3.1: The exit parameter was added. Changed in version 3.2: The verbosity, failfast, catchbreak, buffer and warnings parameters were added.

load_tests Protocol

New in version 3.2. Modules or packages can customize how tests are loaded from them during normal test runs or test discovery by implementing a function called load_tests.

If a test module defines load_tests it will be called by TestLoader.loadTestsFromModule() with the following arguments:

load_tests(loader, standard_tests, None)
It should return a `TestSuite`.

`loader` is the instance of `TestLoader` doing the loading. `standard_tests` are the tests that would be loaded by default from the module. It is common for test modules to only want to add or remove tests from the standard set of tests. The third argument is used when loading packages as part of test discovery.

A typical `load_tests` function that loads tests from a specific set of `TestCase` classes may look like:

```python
def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite
```

If discovery is started, either from the command line or by calling `TestLoader.discover()`, with a pattern that matches a package name then the package `__init__.py` will be checked for `load_tests`.

**Note:** The default pattern is `‘test*.py’`. This matches all Python files that start with `‘test’` but won’t match any test directories.

A pattern like `‘test*’` will match test packages as well as modules.

If the package `__init__.py` defines `load_tests` then it will be called and discovery not continued into the package. `load_tests` is called with the following arguments:

```python
load_tests(loader, standard_tests, pattern)
```

This should return a `TestSuite` representing all the tests from the package. (`standard_tests` will only contain tests collected from `__init__.py`.)

Because the pattern is passed into `load_tests` the package is free to continue (and potentially modify) test discovery. A ‘do nothing’ `load_tests` function for a test package would look like:

```python
def load_tests(loader, standard_tests, pattern):
    # top level directory cached on loader instance
    this_dir = os.path.dirname(__file__)
    package_tests = loader.discover(start_dir=this_dir, pattern=pattern)
    standard_tests.addTests(package_tests)
    return standard_tests
```

### 26.3.8 Class and Module Fixtures

Class and module level fixtures are implemented in `TestSuite`. When the test suite encounters a test from a new class then `tearDownClass()` from the previous class (if there is one) is called, followed by `setUpClass()` from the new class.

Similarly if a test is from a different module from the previous test then `tearDownModule` from the previous module is run, followed by `setUpModule` from the new module.

After all the tests have run the final `tearDownClass` and `tearDownModule` are run.

Note that shared fixtures do not play well with [potential] features like test parallelization and they break test isolation. They should be used with care.

The default ordering of tests created by the unittest test loaders is to group all tests from the same modules and classes together. This will lead to `setUpClass/setUpModule` (etc) being called exactly once per class and module. If you randomize the order, so that tests from different modules and classes are adjacent to each other, then these shared fixture functions may be called multiple times in a single test run.

Shared fixtures are not intended to work with suites with non-standard ordering. A `BaseTestSuite` still exists for frameworks that don’t want to support shared fixtures.
If there are any exceptions raised during one of the shared fixture functions the test is reported as an error. Because there is no corresponding test instance an \_ErrorHolder object (that has the same interface as a \_TestCase) is created to represent the error. If you are just using the standard unittest test runner then this detail doesn’t matter, but if you are a framework author it may be relevant.

**setUpClass and tearDownClass**

These must be implemented as class methods:

```python
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

If you want the setUpClass and tearDownClass on base classes called then you must call up to them yourself. The implementations in Test\_Case are empty.

If an exception is raised during a setUpClass then the tests in the class are not run and the tearDownClass is not run. Skipped classes will not have setUpClass or tearDownClass run. If the exception is a \_SkipTest exception then the class will be reported as having been skipped instead of as an error.

**setUpModule and tearDownModule**

These should be implemented as functions:

```python
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

If an exception is raised in a setUpModule then none of the tests in the module will be run and the tearDownModule will not be run. If the exception is a \_SkipTest exception then the module will be reported as having been skipped instead of as an error.

### 26.3.9 Signal Handling

New in version 3.2. The \(-c/\--catch\) command-line option to unittest, along with the catchbreak parameter to \_unittest\_main(), provide more friendly handling of control-C during a test run. With catch break behavior enabled control-C will allow the currently running test to complete, and the test run will then end and report all the results so far. A second control-c will raise a \_KeyboardInterrupt in the usual way.

The control-c handling signal handler attempts to remain compatible with code or tests that install their own signal.\_SIGINT handler. If the unittest handler is called but isn’t the installed signal.\_SIGINT handler, i.e. it has been replaced by the system under test and delegated to, then it calls the default handler. This will normally be the expected behavior by code that replaces an installed handler and delegates to it. For individual tests that need unittest control-c handling disabled the removeHandler() decorator can be used.

There are a few utility functions for framework authors to enable control-c handling functionality within test frameworks.

```python
unittest.installHandler()
```

Install the control-c handler. When a signal.\_SIGINT is received (usually in response to the user pressing control-c) all registered results have stop() called.
unittest.registerResult (result)
Register a TestResult object for control-c handling. Registering a result stores a weak reference to it, so it doesn’t prevent the result from being garbage collected.

Registering a TestResult object has no side-effects if control-c handling is not enabled, so test frameworks can unconditionally register all results they create independently of whether or not handling is enabled.

unittest.removeResult (result)
Remove a registered result. Once a result has been removed then stop() will no longer be called on that result object in response to a control-c.

unittest.removeHandler (function=None)
When called without arguments this function removes the control-c handler if it has been installed. This function can also be used as a test decorator to temporarily remove the handler whilst the test is being executed:

```python
@unittest.removeHandler
def test_signal_handling(self):
    ...
```

26.4 unittest.mock — mock object library

New in version 3.3. unittest.mock is a library for testing in Python. It allows you to replace parts of your system under test with mock objects and make assertions about how they have been used.

unittest.mock provides a core Mock class removing the need to create a host of stubs throughout your test suite. After performing an action, you can make assertions about which methods / attributes were used and arguments they were called with. You can also specify return values and set needed attributes in the normal way.

Additionally, mock provides a patch() decorator that handles patching module and class level attributes within the scope of a test, along with sentinel for creating unique objects. See the quick guide for some examples of how to use Mock, MagicMock and patch().

Mock is very easy to use and is designed for use with unittest. Mock is based on the ‘action -> assertion’ pattern instead of ‘record -> replay’ used by many mocking frameworks.

There is a backport of unittest.mock for earlier versions of Python, available as mock on PyPI.

Source code: Lib/unittest/mock.py

26.4.1 Quick Guide

Mock and MagicMock objects create all attributes and methods as you access them and store details of how they have been used. You can configure them, to specify return values or limit what attributes are available, and then make assertions about how they have been used:

```python
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
side_effect allows you to perform side effects, including raising an exception when a mock is called:

```python
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
...
KeyError: 'foo'
```
Mock has many other ways you can configure it and control its behaviour. For example the `spec` argument configures the mock to take its specification from another object. Attempting to access attributes or methods on the mock that don’t exist on the spec will fail with an `AttributeError`.

The `patch()` decorator / context manager makes it easy to mock classes or objects in a module under test. The object you specify will be replaced with a mock (or other object) during the test and restored when the test ends:

```python
>>> from unittest.mock import patch
>>> @patch('module.ClassName2')
... @patch('module.ClassName1')
... def test(MockClass1, MockClass2):
...     module.ClassName1()
...     module.ClassName2()
...     assert MockClass1 is module.ClassName1
...     assert MockClass2 is module.ClassName2
...     assert MockClass1.called
...     assert MockClass2.called
...
>>> test()
```

**Note:** When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal Python order that decorators are applied). This means from the bottom up, so in the example above the mock for `module.ClassName1` is passed in first.

With `patch` it matters that you patch objects in the namespace where they are looked up. This is normally straightforward, but for a quick guide read *where to patch.*

As well as a decorator `patch` can be used as a context manager in a with statement:

```python
>>> with patch.object(ProductionClass, 'method', return_value=None) as mock_method:
...     thing = ProductionClass()
...     thing.method(1, 2, 3)
...     mock_method.assert_called_once_with(1, 2, 3)
```

There is also `patch.dict()` for setting values in a dictionary just during a scope and restoring the dictionary to its original state when the test ends:

```python
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...     assert foo == original
```

Mock supports the mocking of Python magic methods. The easiest way of using magic methods is with the `MagicMock` class. It allows you to do things like:

```python
>>> mock = MagicMock()
>>> mock.__str__.return_value = 'foobarbaz'
```

```
'foobaz
>>> mock.__str__.assert_called_with()

Mock allows you to assign functions (or other Mock instances) to magic methods and they will be called appropriately. The MagicMock class is just a Mock variant that has all of the magic methods pre-created for you (well, all the useful ones anyway).

The following is an example of using magic methods with the ordinary Mock class:

```python
global mock

mock = Mock()
mock.__str__ = Mock(return_value='wheeeeee')
str(mock)
'wheeeeee'
```

For ensuring that the mock objects in your tests have the same api as the objects they are replacing, you can use auto-speccing. Auto-speccing can be done through the autospec argument to patch, or the create_autospec() function. Auto-speccing creates mock objects that have the same attributes and methods as the objects they are replacing, and any functions and methods (including constructors) have the same call signature as the real object.

This ensures that your mocks will fail in the same way as your production code if they are used incorrectly:

```python
global mock

from unittest.mock import create_autospec

mock_function = create_autospec(function, return_value='fishy')
mock_function(1, 2, 3)
mock_function.assert_called_once_with(1, 2, 3)
mock_function('wrong arguments')
Traceback (most recent call last):
... TypeError: <lambda>() takes exactly 3 arguments (1 given)
```

create_autospec can also be used on classes, where it copies the signature of the __init__ method, and on callable objects where it copies the signature of the __call__ method.

## 26.4.2 The Mock Class

Mock is a flexible mock object intended to replace the use of stubs and test doubles throughout your code. Mocks are callable and create attributes as new mocks when you access them. Accessing the same attribute will always return the same mock. Mocks record how you use them, allowing you to make assertions about what your code has done to them.

MagicMock is a subclass of Mock with all the magic methods pre-created and ready to use. There are also non-callable variants, useful when you are mocking out objects that aren’t callable: NonCallableMock and NonCallableMagicMock.

The patch() decorators makes it easy to temporarily replace classes in a particular module with a Mock object. By default patch will create a MagicMock for you. You can specify an alternative class of Mock using the new_callable argument to patch.

```python
class unittest.mock.Mock(spec=None, side_effect=None, return_value=DEFAULT, wraps=None, name=None, spec_set=None, **kwargs)
```

Create a new Mock object. Mock takes several optional arguments that specify the behaviour of the Mock object:

```python
*spec: This can be either a list of strings or an existing object (a class or instance) that acts as the specification for the mock object. If you pass in an object then a list of strings is formed by calling dir
```

The only exceptions are magic methods and attributes (those that have leading and trailing double underscores). Mock doesn’t create these but instead of raises an AttributeError. This is because the interpreter will often implicitly request these methods, and gets very confused to get a new Mock object when it expects a magic method. If you need magic method support see magic methods.
on the object (excluding unsupported magic attributes and methods). Accessing any attribute not in
this list will raise an `AttributeError`.

If `spec` is an object (rather than a list of strings) then `__class__` returns the class of the spec object.
This allows mocks to pass `isinstance` tests.

• `spec_set`: A stricter variant of `spec`. If used, attempting to set or get an attribute on the mock that isn’t
  on the object passed as `spec_set` will raise an `AttributeError`.

• `side_effect`: A function to be called whenever the Mock is called. See the `side_effect` attribute.
  Useful for raising exceptions or dynamically changing return values. The function is called with the
  same arguments as the mock, and unless it returns `DEFAULT`, the return value of this function is used
  as the return value.

Alternatively `side_effect` can be an exception class or instance. In this case the exception will be raised
when the mock is called.

If `side_effect` is an iterable then each call to the mock will return the next value from the iterable.

A `side_effect` can be cleared by setting it to `None`.

• `return_value`: The value returned when the mock is called. By default this is a new Mock (created on
  first access). See the `return_value` attribute.

• `wraps`: Item for the mock object to wrap. If `wraps` is not None then calling the Mock will pass the call
  through to the wrapped object (returning the real result). Attribute access on the mock will return a
  Mock object that wraps the corresponding attribute of the wrapped object (so attempting to access an
  attribute that doesn’t exist will raise an `AttributeError`).

If the mock has an explicit `return_value` set then calls are not passed to the wrapped object and the
`return_value` is returned instead.

• `name`: If the mock has a name then it will be used in the repr of the mock. This can be useful for
debugging. The name is propagated to child mocks.

Mocks can also be called with arbitrary keyword arguments. These will be used to set attributes on the mock
after it is created. See the `configure_mock()` method for details.

**assert_called_with** (*args, **kwargs*)

This method is a convenient way of asserting that calls are made in a particular way:

```python
>>> mock = Mock()
>>> mock.method(1, 2, 3, test='wow')
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_with(1, 2, 3, test='wow')
```

**assert_called_once_with** (*args, **kwargs*)

Assert that the mock has been called exactly once and with the specified arguments.

```python
>>> mock = Mock(return_value=None)
>>> mock('foo', bar='baz')
>>> mock.assert_called_once_with('foo', bar='baz')
Traceback (most recent call last):
...
AssertionError: Expected 'mock' to be called once. Called 2 times.
```

**assert_any_call** (*args, **kwargs*)

assert the mock has been called with the specified arguments.

The assert passes if the mock has ever been called, unlike `assert_called_with()` and
`assert_called_once_with()` that only pass if the call is the most recent one.
mock = Mock(return_value=None)
mock(1, 2, arg='thing')
mock('some', 'thing', 'else')
mock.assert_any_call(1, 2, arg='thing')

assert_has_calls(calls, any_order=False)
assert the mock has been called with the specified calls. The mock_calls list is checked for the calls.
If any_order is False (the default) then the calls must be sequential. There can be extra calls before or after the specified calls.
If any_order is True then the calls can be in any order, but they must all appear in mock_calls.

mock = Mock(return_value=None)
mock(1)
mock(2)
mock(3)
mock(4)
calls = [call(2), call(3)]
mock.assert_has_calls(calls)
calls = [call(4), call(2), call(3)]
mock.assert_has_calls(calls, any_order=True)

reset_mock()
The reset_mock method resets all the call attributes on a mock object:

mock = Mock(return_value=None)
mock('hello')
mock.called
True
mock.reset_mock()
mock.called
False

This can be useful where you want to make a series of assertions that reuse the same object. Note that reset_mock doesn’t clear the return value, side_effect or any child attributes you have set using normal assignment. Child mocks and the return value mock (if any) are reset as well.

mock_add_spec(spec, spec_set=False)
Add a spec to a mock. spec can either be an object or a list of strings. Only attributes on the spec can be fetched as attributes from the mock.
If spec_set is True then only attributes on the spec can be set.

attach_mock(mock, attribute)
Attach a mock as an attribute of this one, replacing its name and parent. Calls to the attached mock will be recorded in the method_calls and mock_calls attributes of this one.

configure_mock(**kwargs)
Set attributes on the mock through keyword arguments.
Attributes plus return values and side effects can be set on child mocks using standard dot notation and unpacking a dictionary in the method call:

mock = Mock()
attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
mock.configure_mock(**attrs)
mock.method() 3
mock.other()

Traceback (most recent call last):
The same thing can be achieved in the constructor call to mocks:

```python
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ... KeyError
```

`configure_mock` exists to make it easier to do configuration after the mock has been created.

__dir__()  
Mock objects limit the results of `dir(some_mock)` to useful results. For mocks with a `spec` this includes all the permitted attributes for the mock.

See `FILTER_DIR` for what this filtering does, and how to switch it off.

__get_child_mock__(**kw)  
Create the child mocks for attributes and return value. By default child mocks will be the same type as the parent. Subclasses of Mock may want to override this to customize the way child mocks are made.

For non-callable mocks the callable variant will be used (rather than any custom subclass).

`called`  
A boolean representing whether or not the mock object has been called:

```python
>>> mock = Mock(return_value=None)
>>> mock.called
False
>>> mock()
>>> mock.called
True
```

`call_count`  
An integer telling you how many times the mock object has been called:

```python
>>> mock = Mock(return_value=None)
>>> mock.call_count
0
>>> mock()
>>> mock()
>>> mock.call_count
2
```

`return_value`  
Set this to configure the value returned by calling the mock:

```python
>>> mock = Mock()
>>> mock.return_value = 'fish'
>>> mock()
'fish'
```

The default return value is a mock object and you can configure it in the normal way:
>>> mock = Mock()
>>> mock.return_value.attribute = sentinel.Attribute
>>> mock.return_value()
Mock(name='mock()' id='...
>>> mock.return_value.assert_called_with()

return_value can also be set in the constructor:

>>> mock = Mock(return_value=3)
3
>>> mock()
3

side_effect
This can either be a function to be called when the mock is called, or an exception (class or instance) to be raised.

If you pass in a function it will be called with same arguments as the mock and unless the function returns the DEFAULT singleton the call to the mock will then return whatever the function returns. If the function returns DEFAULT then the mock will return its normal value (from the return_value).

An example of a mock that raises an exception (to test exception handling of an API):

>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
>>> mock()
Traceback (most recent call last):
...
Exception: Boom!

Using side_effect to return a sequence of values:

>>> mock = Mock()
>>> mock.side_effect = [3, 2, 1]
>>> mock(), mock(), mock()
(3, 2, 1)

The side_effect function is called with the same arguments as the mock (so it is wise for it to take arbitrary args and keyword arguments) and whatever it returns is used as the return value for the call. The exception is if side_effect returns DEFAULT, in which case the normal return_value is used.

>>> mock = Mock(return_value=3)
>>> def side_effect(*args, **kwargs):
... return DEFAULT
...
>>> mock.side_effect = side_effect
>>> mock()
3

side_effect can be set in the constructor. Here's an example that adds one to the value the mock is called with and returns it:

>>> side_effect = lambda value: value + 1
>>> mock = Mock(side_effect=side_effect)
>>> mock(3)
4
>>> mock(-8)
-7
Setting `side_effect` to `None` clears it:

```python
>>> m = Mock(side_effect=KeyError, return_value=3)
Traceback (most recent call last):
  ...
KeyError
>>> m.side_effect = None
>>> m()
3
```

**call_args**

This is either `None` (if the mock hasn’t been called), or the arguments that the mock was last called with. This will be in the form of a tuple: the first member is any ordered arguments the mock was called with (or an empty tuple) and the second member is any keyword arguments (or an empty dictionary).

```python
>>> mock = Mock(return_value=None)
>>> print mock.call_args
None
>>> mock()
>>> mock.call_args
call()
>>> mock.call_args == ()
True
>>> mock(3, 4)
>>> mock.call_args
call(3, 4)
>>> mock.call_args == ((3, 4),)
True
>>> mock(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args
call(3, 4, 5, key='fish', next='w00t!')
```

`call_args`, along with members of the lists `call_args_list`, `method_calls`, and `mock_calls` are call objects. These are tuples, so they can be unpacked to get at the individual arguments and make more complex assertions. See **calls as tuples**.

**call_args_list**

This is a list of all the calls made to the mock object in sequence (so the length of the list is the number of times it has been called). Before any calls have been made it is an empty list. The call object can be used for conveniently constructing lists of calls to compare with `call_args_list`.

```python
>>> mock = Mock(return_value=None)
>>> mock()
>>> mock(3, 4)
>>> mock(key='fish', next='w00t!')
>>> mock.call_args_list
[call(), call(3, 4), call(key='fish', next='w00t!')]
>>> expected = [({}, ((3, 4),)), {'key': 'fish', 'next': 'w00t!'}]
>>> mock.call_args_list == expected
True
```

Members of `call_args_list` are call objects. These can be unpacked as tuples to get at the individual arguments. See **calls as tuples**.

**method_calls**

As well as tracking calls to themselves, mocks also track calls to methods and attributes, and their methods and attributes:
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.property/method.attribute()
<Mock name='mock.property.method.attribute()' id='...'>
>>> mock.method_calls
[call.method(), call.property/method.attribute()]

Members of `method_calls` are `call` objects. These can be unpacked as tuples to get at the individual arguments. See `calls as tuples`.

`mock_calls`

`mock_calls` records all calls to the mock object, its methods, magic methods and return value mocks.

>>> mock = MagicMock()
>>> result = mock(1, 2, 3)
>>> mock.first(a=3)
1
>>> result(1)
<MagicMock name='mock()' id='...'>
>>> expected = [call(1, 2, 3), call.first(a=3), call.second(), ...
call.__int__(), call().__call__(1)]
>>> mock.mock_calls == expected
True

Members of `mock_calls` are `call` objects. These can be unpacked as tuples to get at the individual arguments. See `calls as tuples`.

`__class__`

Normally the `__class__` attribute of an object will return its type. For a mock object with a `spec` `__class__` returns the spec class instead. This allows mock objects to pass `isinstance` tests for the object they are replacing / masquerading as:

```python
>>> mock = Mock(spec=3)
>>> isinstance(mock, int)
True
```

`__class__` is assignable to, this allows a mock to pass an `isinstance` check without forcing you to use a `spec`:

```python
>>> mock = Mock()
>>> mock.__class__ = dict
>>> isinstance(mock, dict)
True
```

`class unittest.mock.NonCallableMock(spec=None, wraps=None, name=None, spec_set=None, **kwargs)`

A non-callable version of `Mock`. The constructor parameters have the same meaning of `Mock`, with the exception of `return_value` and `side_effect` which have no meaning on a non-callable mock.

Mock objects that use a class or an instance as a `spec` or `spec_set` are able to pass `isinstance` tests:

```python
>>> mock = Mock(spec=SomeClass)
>>> isinstance(mock, SomeClass)
True
>>> mock = Mock(spec_set=SomeClass())
```
isinstance(mock, SomeClass)

>>> isisinstance(mock, SomeClass)

"True"

The Mock classes have support for mocking magic methods. See magic methods for the full details.

The mock classes and the patch() decorators all take arbitrary keyword arguments for configuration. For the patch decorators the keywords are passed to the constructor of the mock being created. The keyword arguments are for configuring attributes of the mock:

>>> m = MagicMock(attribute=3, other='fish')
>>> m.attribute
3
>>> m.other
'fish'

The return value and side effect of child mocks can be set in the same way, using dotted notation. As you can’t use dotted names directly in a call you have to create a dictionary and unpack it using **:

>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
...
KeyError

class unittest.mock.PropertyMock(*args, **kwargs)

A mock intended to be used as a property, or other descriptor, on a class. PropertyMock provides __get__ and __set__ methods so you can specify a return value when it is fetched.

Fetching a PropertyMock instance from an object calls the mock, with no args. Setting it calls the mock with the value being set.

>>> class Foo:
...     @property
...     def foo(self):
...         return 'something'
...     @foo.setter
...     def foo(self, value):
...         pass
...
>>> with patch('__main__.Foo.foo', new_callable=PropertyMock) as mock_foo:
...     mock_foo.return_value = 'mockity-mock'
...     this_foo = Foo()
...     print this_foo.foo
...     this_foo.foo = 6
...
mockity-mock
>>> mock_foo.mock_calls
[call(), call(6)]

Because of the way mock attributes are stored you can’t directly attach a PropertyMock to a mock object. Instead you can attach it to the mock type object:

>>> m = MagicMock()
>>> p = PropertyMock(return_value=3)
>>> type(m).foo = p
>>> m.foo
1130 Chapter 26. Development Tools
Mock objects are callable. The call will return the value set as the return_value attribute. The default return value is a new Mock object; it is created the first time the return value is accessed (either explicitly or by calling the Mock) - but it is stored and the same one returned each time.

Calls made to the object will be recorded in the attributes like call_args and call_args_list.

If side_effect is set then it will be called after the call has been recorded, so if side_effect raises an exception the call is still recorded.

The simplest way to make a mock raise an exception when called is to make side_effect an exception class or instance:

```python
>>> m = MagicMock(side_effect=IndexError)
>>> m(1, 2, 3)
Traceback (most recent call last):
  ...
IndexError
>>> m.mock_calls
[call(1, 2, 3)]
>>> m.side_effect = KeyError('Bang!')
>>> m('two', 'three', 'four')
Traceback (most recent call last):
  ...
KeyError: 'Bang'
>>> m.mock_calls
[call(1, 2, 3), call('two', 'three', 'four')]
```

If side_effect is a function then whatever that function returns is what calls to the mock return. The side_effect function is called with the same arguments as the mock. This allows you to vary the return value of the call dynamically, based on the input:

```python
>>> def side_effect(value):
...    return value + 1
...    return

>>> m = MagicMock(side_effect=side_effect)
>>> m(1)
2
>>> m(2)
3
>>> m.mock_calls
[call(1), call(2)]
```

If you want the mock to still return the default return value (a new mock), or any set return value, then there are two ways of doing this. Either return mock.return_value from inside side_effect, or return DEFAULT:

```python
>>> m = MagicMock()
>>> def side_effect(*args, **kwargs):
...    return m.return_value
...    return

>>> m.side_effect = side_effect
>>> m.return_value = 3
>>> m()
3
>>> def side_effect(*args, **kwargs):
...    return DEFAULT
...    return
```

26.4. unittest.mock — mock object library
>>> m.side_effect = side_effect
>>> m()
3

To remove a `side_effect`, and return to the default behaviour, set the `side_effect` to `None`:

```python
>>> m = MagicMock(return_value=6)
>>> def side_effect(*args, **kwargs):
...     return 3
...
>>> m.side_effect = side_effect
>>> m()
3
>>> m.side_effect = None
>>> m()
6
```

The `side_effect` can also be any iterable object. Repeated calls to the mock will return values from the iterable (until the iterable is exhausted and a `StopIteration` is raised):

```python
>>> m = MagicMock(side_effect=[1, 2, 3])
>>> m()
1
>>> m()
2
>>> m()
3
>>> m()
Traceback (most recent call last):
  ... StopIteration
```

If any members of the iterable are exceptions they will be raised instead of returned:

```python
>>> iterable = (33, ValueError, 66)
>>> m = MagicMock(side_effect=iterable)
>>> m()
33
>>> m()
Traceback (most recent call last):
  ... ValueError
>>> m()
66
```

### Deleting Attributes

Mock objects create attributes on demand. This allows them to pretend to be objects of any type.

You may want a mock object to return `False` to a `hasattr` call, or raise an `AttributeError` when an attribute is fetched. You can do this by providing an object as a `spec` for a mock, but that isn’t always convenient.

You “block” attributes by deleting them. Once deleted, accessing an attribute will raise an `AttributeError`.

```python
>>> mock = MagicMock()
>>> hasattr(mock, 'm')
True
>>> del mock.m
>>> hasattr(mock, 'm')
False
>>> del mock.f
>>> mock.f
```
Mock names and the name attribute

Since “name” is an argument to the Mock constructor, if you want your mock object to have a “name” attribute you can’t just pass it in at creation time. There are two alternatives. One option is to use configure_mock():

```python
>>> mock = MagicMock()
>>> mock.configure_mock(name='my_name')
>>> mock.name
'my_name'
```

A simpler option is to simply set the “name” attribute after mock creation:

```python
>>> mock = MagicMock()
>>> mock.name = 'foo'
```

Attaching Mocks as Attributes

When you attach a mock as an attribute of another mock (or as the return value) it becomes a “child” of that mock. Calls to the child are recorded in the method_calls and mock_calls attributes of the parent. This is useful for configuring child mocks and then attaching them to the parent, or for attaching mocks to a parent that records all calls to the children and allows you to make assertions about the order of calls between mocks:

```python
>>> parent = MagicMock()
>>> child1 = MagicMock(return_value=None)
>>> child2 = MagicMock(return_value=None)
>>> parent.child1 = child1
>>> parent.child2 = child2
>>> child1(1)
>>> child2(2)
>>> parent.mock_calls
[call.child1(1), call.child2(2)]
```

The exception to this is if the mock has a name. This allows you to prevent the “parenting” if for some reason you don’t want it to happen.

```python
>>> mock = MagicMock()
>>> not_a_child = MagicMock(name='not-a-child')
>>> mock.attribute = not_a_child
>>> mock.attribute()
<MagicMock name='not-a-child()' id='...'>
>>> mock.mock_calls
[]
```

Mocks created for you by patch() are automatically given names. To attach mocks that have names to a parent you use the attach_mock() method:

```python
>>> thing1 = object()
>>> thing2 = object()
>>> parent = MagicMock()
>>> with patch('__main__.thing1', return_value=None) as child1:
...     with patch('__main__.thing2', return_value=None) as child2:
...         parent.attach_mock(child1, 'child1')
...         parent.attach_mock(child2, 'child2')
...         child1('one')
...         child2('two')
...```
26.4.3 The patchers

The patch decorators are used for patching objects only within the scope of the function they decorate. They automatically handle the unpatching for you, even if exceptions are raised. All of these functions can also be used in with statements or as class decorators.

patch

Note: patch is straightforward to use. The key is to do the patching in the right namespace. See the section where to patch.

unittest.mock.patch(target, new=DEFAULT, spec=None, create=False, spec_set=None, autospec=None, new_callable=None, **kwargs)

patch acts as a function decorator, class decorator or a context manager. Inside the body of the function or with statement, the target is patched with a new object. When the function/with statement exits the patch is undone.

If new is omitted, then the target is replaced with a MagicMock. If patch is used as a decorator and new is omitted, the created mock is passed in as an extra argument to the decorated function. If patch is used as a context manager the created mock is returned by the context manager.

target should be a string in the form 'package.module.ClassName'. The target is imported and the specified object replaced with the new object, so the target must be importable from the environment you are calling patch from. The target is imported when the decorated function is executed, not at decoration time.

The spec and spec_set keyword arguments are passed to the MagicMock if patch is creating one for you.

In addition you can pass spec=True or spec_set=True, which causes patch to pass in the object being mocked as the spec/spec_set object.

new_callable allows you to specify a different class, or callable object, that will be called to create the new object. By default MagicMock is used.

A more powerful form of spec is autospec. If you set autospec=True then the mock with be created with a spec from the object being replaced. All attributes of the mock will also have the spec of the corresponding attribute of the object being replaced. Methods and functions being mocked will have their arguments checked and will raise a TypeError if they are called with the wrong signature. For mocks replacing a class, their return value (the ‘instance’) will have the same spec as the class. See the create_autospec() function and Autospeccing.

Instead of autospec=True you can pass autospec=some_object to use an arbitrary object as the spec instead of the one being replaced.

By default patch will fail to replace attributes that don’t exist. If you pass in create=True, and the attribute doesn’t exist, patch will create the attribute for you when the patched function is called, and delete it again afterwards. This is useful for writing tests against attributes that your production code creates at runtime. It is off by default because it can be dangerous. With it switched on you can write passing tests against APIs that don’t actually exist!

Patch can be used as a TestCase class decorator. It works by decorating each test method in the class. This reduces the boilerplate code when your test methods share a common patchings set. patch finds tests by looking for method names that start with patch.TEST_PREFIX. By default this is test, which matches the way unittest finds tests. You can specify an alternative prefix by setting patch.TEST_PREFIX.

Patch can be used as a context manager, with the with statement. Here the patching applies to the indented block after the with statement. If you use “as” then the patched object will be bound to the name after the “as”; very useful if patch is creating a mock object for you.
*patch* takes arbitrary keyword arguments. These will be passed to the *Mock* (or *new_callable*) on construction.

*patch.dict(...), patch.multiple(...) and patch.object(....) are available for alternate use-cases.

*patch* as function decorator, creating the mock for you and passing it into the decorated function:

```python
>>> @patch('__main__.SomeClass')
... def function(normal_argument, mock_class):
...     print(mock_class is SomeClass)
...
>>> function(None)
True
```

Patching a class replaces the class with a *MagicMock* instance. If the class is instantiated in the code under test then it will be the *return_value* of the mock that will be used.

If the class is instantiated multiple times you could use *side_effect* to return a new mock each time. Alternatively you can set the *return_value* to be anything you want.

To configure return values on methods of *instances* on the patched class you must do this on the *return_value*. For example:

```python
>>> class Class:
...     def method(self):
...         pass
...
>>> with patch('__main__.Class') as MockClass:
...     instance = MockClass.return_value
...     instance.method.return_value = 'foo'
...     assert Class() is instance
...     assert Class().method() == 'foo'
...
```

If you use *spec* or *spec_set* and *patch* is replacing a class, then the return value of the created mock will have the same spec.

```python
>>> Original = Class
>>> patcher = patch('__main__.Class', spec=True)
>>> MockClass = patcher.start()
>>> instance = MockClass()
>>> assert isinstance(instance, Original)
>>> patcher.stop()
```

The *new_callable* argument is useful where you want to use an alternative class to the default *MagicMock* for the created mock. For example, if you wanted a *NonCallableMock* to be used:

```python
>>> thing = object()
>>> with patch('__main__.thing', new_callable=NonCallableMock) as mock_thing:
...     assert thing is mock_thing
...     thing()
...
```

```
Traceback (most recent call last):
...
TypeError: 'NonCallableMock' object is not callable
```

Another use case might be to replace an object with a *io.StringIO* instance:

```python
>>> from io import StringIO
>>> def foo():
...     print 'Something'
...
>>> with patch('sys.stdout', new_callable=StringIO) as mock_stdout:
...     foo()
```
When `patch` is creating a mock for you, it is common that the first thing you need to do is to configure the mock. Some of that configuration can be done in the call to `patch`. Any arbitrary keywords you pass into the call will be used to set attributes on the created mock:

```python
>>> patcher = patch('__main__.thing', first='one', second='two')
>>> mock_thing = patcher.start()
>>> mock_thing.first
'one'
>>> mock_thing.second
'two'
```

As well as attributes on the created mock attributes, like the `return_value` and `side_effect`, of child mocks can also be configured. These aren’t syntactically valid to pass in directly as keyword arguments, but a dictionary with these as keys can still be expanded into a `patch` call using `**`:

```python
>>> config = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> patcher = patch('__main__.thing', **config)
>>> mock_thing = patcher.start()
>>> mock_thing.method()
3
>>> mock_thing.other()
Traceback (most recent call last):
 ...
KeyError
```

**patch.object**

patch.object(target, attribute, new=DEFAULT, spec=None, create=False, spec_set=None, autospec=None, new_callable=None, **kwargs)

patch the named member (attribute) on an object (target) with a mock object.

`patch.object` can be used as a decorator, class decorator or a context manager. Arguments `new`, `spec`, `create`, `spec_set`, `autospec` and `new_callable` have the same meaning as for `patch`. Like `patch`, `patch.object` takes arbitrary keyword arguments for configuring the mock object it creates.

When used as a class decorator `patch.object` honours `patch.TEST_PREFIX` for choosing which methods to wrap.

You can either call `patch.object` with three arguments or two arguments. The three argument form takes the object to be patched, the attribute name and the object to replace the attribute with.

When calling with the two argument form you omit the replacement object, and a mock is created for you and passed in as an extra argument to the decorated function:

```python
>>> @patch.object(SomeClass, 'class_method')
... def test(mock_method):
...   SomeClass.class_method(3)
...   mock_method.assert_called_with(3)
... >>> test()
```

`spec`, `create` and the other arguments to `patch.object` have the same meaning as they do for `patch`.

**patch.dict**

patch.dict(in_dict=(), clear=False, **kwargs)

Patch a dictionary, or dictionary like object, and restore the dictionary to its original state after the test.
**in_dict** can be a dictionary or a mapping like container. If it is a mapping then it must at least support getting, setting and deleting items plus iterating over keys.

**in_dict** can also be a string specifying the name of the dictionary, which will then be fetched by importing it.

**values** can be a dictionary of values to set in the dictionary. **values** can also be an iterable of (key, value) pairs.

If **clear** is True then the dictionary will be cleared before the new values are set.

**patch.dict** can also be called with arbitrary keyword arguments to set values in the dictionary.

**patch.dict** can be used as a context manager, decorator or class decorator. When used as a class decorator **patch.dict** honours **patch.TEST_PREFIX** for choosing which methods to wrap.

**patch.dict** can be used to add members to a dictionary, or simply let a test change a dictionary, and ensure the dictionary is restored when the test ends.

```python
def foo = {}
def with patch.dict(foo, {'newkey': 'newvalue'}):
    assert foo == {'newkey': 'newvalue'}

>>> assert foo == {}  

>>> import os
>>> with patch.dict('os.environ', {'newkey': 'newvalue'}):
    print os.environ['newkey']

newvalue
>>> assert 'newkey' not in os.environ
```

Keywords can be used in the **patch.dict** call to set values in the dictionary:

```python
>>> mymodule = MagicMock()
>>> mymodule.function.return_value = 'fish'
>>> with patch.dict('sys.modules', mymodule=mymodule):
    mymodule.function('some', 'args')

'fish'
```

**patch.dict** can be used with dictionary like objects that aren’t actually dictionaries. At the very minimum they must support item getting, setting, deleting and either iteration or membership test. This corresponds to the magic methods **__getitem__**, **__setitem__**, **__delitem__** and either **__iter__** or **__contains__**.

```python
class Container:
    def __init__(self):
        self.values = {}
    def __getitem__(self, name):
        return self.values[name]
    def __setitem__(self, name, value):
        self.values[name] = value
    def __delitem__(self, name):
        del self.values[name]
    def __iter__(self):
        return iter(self.values)

>>> thing = Container()
>>> thing['one'] = 1
>>> with patch.dict(thing, one=2, two=3):
    assert thing['one'] == 2
    assert thing['two'] == 3
```
>>> assert thing['one'] == 1
>>> assert list(thing) == ['one']

patch.multiple

patch.multiple(target, spec=None, create=False, spec_set=None, autospec=None, new_callable=None, **kwargs)
Perform multiple patches in a single call. It takes the object to be patched (either as an object or a string to fetch the object by importing) and keyword arguments for the patches:

```python
with patch.multiple(settings, FIRST_PATCH='one', SECOND_PATCH='two'):
...
```

Use DEFAULT as the value if you want patch.multiple to create mocks for you. In this case the created mocks are passed into a decorated function by keyword, and a dictionary is returned when patch.multiple is used as a context manager.

patch.multiple can be used as a decorator, class decorator or a context manager. The arguments spec, spec_set, create, autospec and new_callable have the same meaning as for patch. These arguments will be applied to all patches done by patch.multiple.

When used as a class decorator patch.multiple honours patch.TEST_PREFIX for choosing which methods to wrap.

If you want patch.multiple to create mocks for you, then you can use DEFAULT as the value. If you use patch.multiple as a decorator then the created mocks are passed into the decorated function by keyword.

```python
>>> thing = object()
>>> other = object()
>>> @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(thing, other):
...    assert isinstance(thing, MagicMock)
...    assert isinstance(other, MagicMock)
...    ...
>>> test_function()
```

patch.multiple can be nested with other patch decorators, but put arguments passed by keyword after any of the standard arguments created by patch:

```python
>>> @patch('sys.exit')
... @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(mock_exit, other, thing):
...    assert 'other' in repr(other)
...    assert 'thing' in repr(thing)
...    assert 'exit' in repr(mock_exit)
...
>>> test_function()
```

If patch.multiple is used as a context manager, the value returned by the context manager is a dictionary where created mocks are keyed by name:

```python
>>> with patch.multiple('__main__', thing=DEFAULT, other=DEFAULT) as values:
...    assert 'other' in repr(values['other'])
...    assert 'thing' in repr(values['thing'])
...    assert values['thing'] is thing
...    assert values['other'] is other
...
patch methods: start and stop

All the patchers have `start` and `stop` methods. These make it simpler to do patching in `setUp` methods or where you want to do multiple patches without nesting decorators or with statements.

To use them call `patch`, `patch.object` or `patch.dict` as normal and keep a reference to the returned `patcher` object. You can then call `start` to put the patch in place and `stop` to undo it.

If you are using `patch` to create a mock for you then it will be returned by the call to `patcher.start`.

>>> patcher = patch('package.module.ClassName')
>>> from package import module
>>> original = module.ClassName
>>> new_mock = patcher.start()
>>> assert module.ClassName is not original
>>> assert module.ClassName is new_mock
>>> patcher.stop()
>>> assert module.ClassName is original
>>> assert module.ClassName is not new_mock

A typical use case for this might be for doing multiple patches in the `setUp` method of a `TestCase`:

```python
>>> class MyTest(TestCase):
...     def setUp(self):
...         self.patcher1 = patch('package.module.Class1')
...         self.patcher2 = patch('package.module.Class2')
...         self.MockClass1 = self.patcher1.start()
...         self.MockClass2 = self.patcher2.start()
...
...     def tearDown(self):
...         self.patcher1.stop()
...         self.patcher2.stop()
...
...     def test_something(self):
...         assert package.module.Class1 is self.MockClass1
...         assert package.module.Class2 is self.MockClass2
...
>>> MyTest('test_something').run()
```

Caution: If you use this technique you must ensure that the patching is "undone" by calling `stop`. This can be fiddlier than you might think, because if an exception is raised in the `setUp` then `tearDown` is not called. `unittest.TestCase.addCleanup()` makes this easier:

```python
>>> class MyTest(TestCase):
...     def setUp(self):
...         patcher = patch('package.module.Class')
...         self.MockClass = patcher.start()
...         self.addCleanup(patcher.stop)
...
...     def test_something(self):
...         assert package.module.Class is self.MockClass
...
>>> MyTest('test_something').run()
```

As an added bonus you no longer need to keep a reference to the `patcher` object.

It is also possible to stop all patches which have been started by using `patch.stopall`.

```python
patch.stopall()
```
Stop all active patches. Only stops patches started with `start`. 
TEST_PREFIX

All of the patchers can be used as class decorators. When used in this way they wrap every test method on the class. The patchers recognise methods that start with test as being test methods. This is the same way that the unittest.TestLoader finds test methods by default.

It is possible that you want to use a different prefix for your tests. You can inform the patchers of the different prefix by setting patch.TEST_PREFIX:

```python
>>> patch.TEST_PREFIX = 'foo'
>>> value = 3
>>> @patch('__main__.value', 'not three')
... class Thing:
...     def foo_one(self):
...         print value
...     def foo_two(self):
...         print value
...
>>> Thing().foo_one()
not three
>>> Thing().foo_two()
not three
>>> value
3
```

Nesting Patch Decorators

If you want to perform multiple patches then you can simply stack up the decorators.

You can stack up multiple patch decorators using this pattern:

```python
>>> @patch.object(SomeClass, 'class_method')
... @patch.object(SomeClass, 'static_method')
... def test(mock1, mock2):
...     assert SomeClass.static_method is mock1
...     assert SomeClass.class_method is mock2
...     SomeClass.class_method('foo')
...     SomeClass.class_method('bar')
...     return mock1, mock2
...
>>> mock1, mock2 = test()
>>> mock1.assert_called_once_with('foo')
>>> mock2.assert_called_once_with('bar')
```

Note that the decorators are applied from the bottom upwards. This is the standard way that Python applies decorators. The order of the created mocks passed into your test function matches this order.

Where to patch

patch works by (temporarily) changing the object that a name points to with another one. There can be many names pointing to any individual object, so for patching to work you must ensure that you patch the name used by the system under test.

The basic principle is that you patch where an object is looked up, which is not necessarily the same place as where it is defined. A couple of examples will help to clarify this.

Imagine we have a project that we want to test with the following structure:
a.py
   -> Defines SomeClass

b.py
   -> from a import SomeClass
   -> some_function instantiates SomeClass

Now we want to test some_function but we want to mock out SomeClass using patch. The problem is that when we import module b, which we will have to do then it imports SomeClass from module a. If we use patch to mock out a.SomeClass then it will have no effect on our test; module b already has a reference to the real SomeClass and it looks like our patching had no effect.

The key is to patch out SomeClass where it is used (or where it is looked up). In this case some_function will actually look up SomeClass in module b, where we have imported it. The patching should look like:

```
@patch('b.SomeClass')
```

However, consider the alternative scenario where instead of from a import SomeClass module b does import a and some_function uses a.SomeClass. Both of these import forms are common. In this case the class we want to patch is being looked up on the a module and so we have to patch a.SomeClass instead:

```
@patch('a.SomeClass')
```

### Patching Descriptors and Proxy Objects

Both patch and patch.object correctly patch and restore descriptors: class methods, static methods and properties. You should patch these on the class rather than an instance. They also work with some objects that proxy attribute access, like the django settings object.

### 26.4.4 MagicMock and magic method support

#### Mocking Magic Methods

Mock supports mocking the Python protocol methods, also known as “magic methods”. This allows mock objects to replace containers or other objects that implement Python protocols.

Because magic methods are looked up differently from normal methods, this support has been specially implemented. This means that only specific magic methods are supported. The supported list includes almost all of them. If there are any missing that you need please let us know.

You mock magic methods by setting the method you are interested in to a function or a mock instance. If you are using a function then it must take self as the first argument.

```
>>> def __str__(self):
     ...     return 'fooble'
     ...
>>>
>>> mock = Mock()
>>> mock.__str__ = __str__
>>> str(mock)
'fooble'
>>>
>>> mock = Mock()
>>> mock.__str__ = Mock()
>>> mock.__str__.return_value = 'fooble'
>>> str(mock)
'fooble'
```

3 Magic methods should be looked up on the class rather than the instance. Different versions of Python are inconsistent about applying this rule. The supported protocol methods should work with all supported versions of Python.

4 The function is basically hooked up to the class, but each Mock instance is kept isolated from the others.
>>> mock = Mock()
>>> mock.__iter__ = Mock(return_value=iter([]))
>>> list(mock)
[]

One use case for this is for mocking objects used as context managers in a `with` statement:

>>> mock = Mock()
>>> mock.__enter__ = Mock(return_value='foo')
>>> mock.__exit__ = Mock(return_value=False)

```python
>>> with mock as m:
...     assert m == 'foo'
...```
```python
>>> mock.__enter__.assert_called_with()
>>> mock.__exit__.assert_called_with(None, None, None)
```

Calls to magic methods do not appear in `method_calls`, but they are recorded in `mock_calls`.

---

**Note:** If you use the *spec* keyword argument to create a mock then attempting to set a magic method that isn’t in the spec will raise an `AttributeError`.

---

The full list of supported magic methods is:

- `__hash__`, `__sizeof__`, `__repr__` and `__str__`
- `__dir__`, `__format__` and `__subclasses__`
- `__floor__`, `__trunc__` and `__ceil__`
- Comparisons: `__cmp__`, `__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__` and `__ne__`
- Container methods: `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`, `__len__`, `__iter__`, `__getslice__`, `__setslice__`, `__reversed__` and `__missing__`
- Context manager: `__enter__` and `__exit__`
- Unary numeric methods: `__neg__`, `__pos__` and `__invert__`
- The numeric methods (including right hand and in-place variants): `__add__`, `__sub__`, `__mul__`, `__div__`, `__floordiv__`, `__mod__`, `__divmod__`, `__lshift__`, `__rshift__`, `__and__`, `__xor__`, `__or__`, and `__pow__`
- Numeric conversion methods: `__complex__`, `__int__`, `__float__`, `__index__` and `__coerce__`
- Descriptor methods: `__get__`, `__set__` and `__delete__`
- Pickling: `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` and `__setstate__`

The following methods exist but are *not* supported as they are either in use by mock, can’t be set dynamically, or can cause problems:

- `__getattr__`, `__setattr__`, `__init__` and `__new__`
- `__prepare__`, `__instancecheck__`, `__subclasscheck__`, `__del__`

---

**Magic Mock**

There are two `MagicMock` variants: `MagicMock` and `NonCallableMock`.

```python
class unittest.mock.MagicMock(*args, **kw)
```

`MagicMock` is a subclass of `Mock` with default implementations of most of the magic methods. You can use `MagicMock` without having to configure the magic methods yourself.

The constructor parameters have the same meaning as for `Mock`. 
If you use the spec or spec_set arguments then only magic methods that exist in the spec will be created.

```python
class unittest.mock.NonCallableMagicMock(*args, **kw)
A non-callable version of MagicMock.
```

The constructor parameters have the same meaning as for MagicMock, with the exception of return_value and side_effect which have no meaning on a non-callable mock.

The magic methods are setup with MagicMock objects, so you can configure them and use them in the usual way:

```python
>>> mock = MagicMock()  
>>> mock[3] = 'fish'  
>>> mock.__setitem__.assert_called_with(3, 'fish')
>>> mock.__getitem__.return_value = 'result'  
>>> mock[2]  
'result'
```

By default many of the protocol methods are required to return objects of a specific type. These methods are preconfigured with a default return value, so that they can be used without you having to do anything if you aren’t interested in the return value. You can still set the return value manually if you want to change the default.

Methods and their defaults:

- `__lt__`: NotImplemented
- `__gt__`: NotImplemented
- `__le__`: NotImplemented
- `__ge__`: NotImplemented
- `__int__`: 1
- `__contains__`: False
- `__len__`: 1
- `__iter__`: iter([])
- `__exit__`: False
- `__complex__`: 1j
- `__float__`: 1.0
- `__bool__`: True
- `__index__`: 1
- `__hash__`: default hash for the mock
- `__str__`: default str for the mock
- `__sizeof__`: default sizeof for the mock

For example:

```python
>>> mock = MagicMock()  
>>> int(mock)  
1  
>>> len(mock)  
0  
>>> list(mock)  
[]  
>>> object() in mock  
False
```

The two equality method, `__eq__` and `__ne__`, are special. They do the default equality comparison on identity, using a side effect, unless you change their return value to return something else:
>>> MagicMock() == 3
False
>>> MagicMock() != 3
True

>>> mock = MagicMock()
>>> mock._eq_.return_value = True
>>> mock == 3
True

The return value of `MagicMock.__iter__` can be any iterable object and isn’t required to be an iterator:

>>> mock = MagicMock()
>>> mock._iter_.return_value = ['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']

If the return value is an iterator, then iterating over it once will consume it and subsequent iterations will result in an empty list:

>>> mock._iter_.return_value = iter(['a', 'b', 'c'])
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
[]

MagicMock has all of the supported magic methods configured except for some of the obscure and obsolete ones. You can still set these up if you want.

Magic methods that are supported but not setup by default in `MagicMock` are:

- `__subclasses__`
- `__dir__`
- `__format__`
- `__get__, __set__, and __delete__`
- `__reversed__, __missing__`
- `__reduce__, __reduce_ex__, __getinitargs__, __getnewargs__, __getstate__ and __setstate__`
- `__getformat__ and __setformat__`

### 26.4.5 Helpers

#### sentinel

`unittest.mock.sentinel`

The `sentinel` object provides a convenient way of providing unique objects for your tests.

Attributes are created on demand when you access them by name. Accessing the same attribute will always return the same object. The objects returned have a sensible `repr` so that test failure messages are readable.

Sometimes when testing you need to test that a specific object is passed as an argument to another method, or returned. It can be common to create named `sentinel` objects to test this. `sentinel` provides a convenient way of creating and testing the identity of objects like this.

In this example we monkey patch `method` to return `sentinel.some_object`:

```python
>>> real = ProductionClass()
>>> real.method = Mock(name="method")
>>> real.method.return_value = sentinel.some_object
```
>>> result = real.method()
>>> assert result is sentinel.some_object
>>> sentinel.some_object

DEFAULT

unittest.mock.DEFAULT

The DEFAULT object is a pre-created sentinel (actually sentinel.DEFAULT). It can be used by side_effect functions to indicate that the normal return value should be used.

call

unittest.mock.call(*args, **kwargs)

call is a helper object for making simpler assertions, for comparing with call_args, call_args_list, mock_calls and method_calls. call can also be used with assert_has_calls().

>>> m = MagicMock(return_value=None)
>>> m(1, 2, a='foo', b='bar')
>>> m()
>>> m.call_args_list == [call(1, 2, a='foo', b='bar'), call()]
True

call.call_list()

For a call object that represents multiple calls, call_list returns a list of all the intermediate calls as well as the final call.

call_list is particularly useful for making assertions on “chained calls”. A chained call is multiple calls on a single line of code. This results in multiple entries in mock_calls on a mock. Manually constructing the sequence of calls can be tedious.

call_list() can construct the sequence of calls from the same chained call:

>>> m = MagicMock()
>>> m(1).method(arg='foo').other(arg='bar')(2.0)
<MagicMock name='mock().method().other()' id='...'>
>>> kall = call(1).method(arg='foo').other(arg='bar')(2.0)
>>> kall.call_list()
[call(1),
call().method(arg='foo'),
call().method().other('bar'),
call().method().other()(2.0)]
>>> m.mock_calls == kall.call_list()
True

A call object is either a tuple of (positional args, keyword args) or (name, positional args, keyword args) depending on how it was constructed. When you construct them yourself this isn’t particularly interesting, but the call objects that are in the Mock.call_args, Mock.call_args_list and Mock.mock_calls attributes can be introspected to get at the individual arguments they contain.

The call objects in Mock.call_args and Mock.call_args_list are two-tuples of (positional args, keyword args) whereas the call objects in Mock.mock_calls, along with ones you construct yourself, are three-tuples of (name, positional args, keyword args).

You can use their “tupleness” to pull out the individual arguments for more complex introspection and assertions. The positional arguments are a tuple (an empty tuple if there are no positional arguments) and the keyword arguments are a dictionary:
>>> m = MagicMock(return_value=None)
>>> m(1, 2, 3, arg='one', arg2='two')
>>> kall = m.call_args
>>> args, kwargs = kall
>>> args
(1, 2, 3)
>>> kwargs
{'arg2': 'two', 'arg': 'one'}
>>> args is kall[0]
True
>>> kwargs is kall[1]
True

>>> m = MagicMock()
>>> m.foo(4, 5, 6, arg='two', arg2='three')
<MagicMock name='mock.foo()' id='...'>
>>> kall = m.mock_calls[0]
>>> name, args, kwargs = kall
>>> name
'foo'
>>> args
(4, 5, 6)
>>> kwargs
{'arg2': 'three', 'arg': 'two'}
>>> name is m.mock_calls[0][0]
True

create_autospec

unittest.mock.create_autospec(spec, spec_set=False, instance=False, **kwargs)

Create a mock object using another object as a spec. Attributes on the mock will use the corresponding
attribute on the spec object as their spec.

Functions or methods being mocked will have their arguments checked to ensure that they are called with
the correct signature.

If spec_set is True then attempting to set attributes that don’t exist on the spec object will raise an Attribu-
teError.

If a class is used as a spec then the return value of the mock (the instance of the class) will have the same
spec. You can use a class as the spec for an instance object by passing instance=True. The returned mock
will only be callable if instances of the mock are callable.

create_autospec also takes arbitrary keyword arguments that are passed to the constructor of the created
mock.

See Autospeccing for examples of how to use auto-speccing with create_autospec and the autospec argument to
patch().

ANY

unittest.mock.ANY

Sometimes you may need to make assertions about some of the arguments in a call to mock, but either not care
about some of the arguments or want to pull them individually out of call_args and make more complex
assertions on them.

To ignore certain arguments you can pass in objects that compare equal to everything. Calls to
assert_called_with() and assert_called_once_with() will then succeed no matter what was
passed in.
>>> mock = Mock(return_value=None)
>>> mock('foo', bar=object())
>>> mock.assert_called_once_with('foo', bar=ANY)

ANY can also be used in comparisons with call lists like mock_calls:

>>> m = MagicMock(return_value=None)
>>> m(1)
>>> m(1, 2)
>>> m(object())
>>> m.mock_calls == [call(1), call(1, 2), ANY]
True

**FILTER\_DIR**

unittest.mock.FILTER\_DIR

FILTER\_DIR is a module level variable that controls the way mock objects respond to dir (only for Python 2.6 or more recent). The default is True, which uses the filtering described below, to only show useful members. If you dislike this filtering, or need to switch it off for diagnostic purposes, then set mock.FILTER\_DIR = False.

With filtering on, dir(some_mock) shows only useful attributes and will include any dynamically created attributes that wouldn’t normally be shown. If the mock was created with a spec (or autospec of course) then all the attributes from the original are shown, even if they haven’t been accessed yet:

```python
>>> dir(Mock())
['assert_any_call',
 'assert_called_once_with',
 'assert_called_with',
 'assert_has_calls',
 'attach_mock',
 ...]
```

```python
>>> from urllib import request
>>> dir(Mock(spec=request))
['AbstractBasicAuthHandler',
 'AbstractDigestAuthHandler',
 'AbstractHTTPHandler',
 'BaseHandler',
 ...]
```

Many of the not-very-useful (private to Mock rather than the thing being mocked) underscore and double underscore prefixed attributes have been filtered from the result of calling dir on a Mock. If you dislike this behaviour you can switch it off by setting the module level switch FILTER\_DIR:

```python
>>> from unittest import mock
>>> mock.FILTER\_DIR = False
```

```python
>>> dir(mock.Mock())
['_NonCallableMock__get_return_value',
 '_NonCallableMock__get_side_effect',
 '_NonCallableMock__return_value_doc',
 '_NonCallableMock__set_return_value',
 '_NonCallableMock__set_side_effect',
 '__call__',
 '__class__',
 ...]
```

Alternatively you can just use vars(my_mock) (instance members) and dir(type(my_mock)) (type members) to bypass the filtering irrespective of mock.FILTER\_DIR.
**mock_open**

`unittest.mock.mock_open (mock=None, read_data=None)`

A helper function to create a mock to replace the use of `open`. It works for `open` called directly or used as a context manager.

The `mock` argument is the mock object to configure. If `None` (the default) then a `MagicMock` will be created for you, with the API limited to methods or attributes available on standard file handles.

`read_data` is a string for the `~io.IOBase.read` method of the file handle to return. This is an empty string by default.

Using `open` as a context manager is a great way to ensure your file handles are closed properly and is becoming common:

```python
with open('/some/path', 'w') as f:
    f.write('something')
```

The issue is that even if you mock out the call to `open` it is the returned object that is used as a context manager (and has `__enter__` and `__exit__` called).

Mocking context managers with a `MagicMock` is common enough and fiddly enough that a helper function is useful.

```python
>>> m = mock_open()
>>> with patch('__main__.open', m, create=True):
...     with open('foo', 'w') as h:
...         h.write('some stuff')
...     m.mock_calls
[call('foo', 'w'),
call().__enter__(),
call().write('some stuff'),
call().__exit__(None, None, None)]
>>> m.assert_called_once_with('foo', 'w')
>>> handle = m()
>>> handle.write.assert_called_once_with('some stuff')
```

And for reading files:

```python
>>> with patch('__main__.open', mock_open(read_data='bibble'), create=True) as m:
...     with open('foo') as h:
...         result = h.read()
...     m.assert_called_once_with('foo')
>>> assert result == 'bibble'
```

**Autospeccing**

Autospeccing is based on the existing `spec` feature of mock. It limits the api of mocks to the api of an original object (the spec), but it is recursive (implemented lazily) so that attributes of mocks only have the same api as the attributes of the spec. In addition mocked functions / methods have the same call signature as the original so they raise a `TypeError` if they are called incorrectly.

Before I explain how auto-speccing works, here’s why it is needed.

`Mock` is a very powerful and flexible object, but it suffers from two flaws when used to mock out objects from a system under test. One of these flaws is specific to the `Mock` api and the other is a more general problem with using mock objects.

First the problem specific to `Mock`. `Mock` has two assert methods that are extremely handy: `assert_called_with()` and `assert_called_once_with()`.
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
>>> mock(1, 2, 3)
Traceback (most recent call last):
 ...
AssertionError: Expected ‘mock’ to be called once. Called 2 times.

Because mocks auto-create attributes on demand, and allow you to call them with arbitrary arguments, if you misspell one of these assert methods then your assertion is gone:

>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(4, 5, 6)
Your tests can pass silently and incorrectly because of the typo.

The second issue is more general to mocking. If you refactor some of your code, rename members and so on, any tests for code that is still using the old api but uses mocks instead of the real objects will still pass. This means your tests can all pass even though your code is broken.

Note that this is another reason why you need integration tests as well as unit tests. Testing everything in isolation is all fine and dandy, but if you don’t test how your units are “wired together” there is still lots of room for bugs that tests might have caught.

`mock` already provides a feature to help with this, called speccing. If you use a class or instance as the spec for a mock then you can only access attributes on the mock that exist on the real class:

>>> from urllib import request
>>> mock = Mock(spec=request.Request)
>>> mock.assert_called_with
Traceback (most recent call last):
 ...
AttributeError: Mock object has no attribute ‘assret_called_with’

The spec only applies to the mock itself, so we still have the same issue with any methods on the mock:

>>> mock.has_data()
<Mock object at 0x...>
>>> mock.has_data.assert_called_with()

Auto-speccing solves this problem. You can either pass autospec=True to `patch / patch.object` or use the create_autospec function to create a mock with a spec. If you use the autospec=True argument to `patch` then the object that is being replaced will be used as the spec object. Because the speccing is done “lazily” (the spec is created as attributes on the mock are accessed) you can use it with very complex or deeply nested objects (like modules that import modules that import modules) without a big performance hit.

Here’s an example of it in use:

>>> from urllib import request
>>> patcher = patch('__main__.request', autospec=True)
>>> mock_request = patcher.start()
>>> request is mock_request
True
>>> mock_request.Request
<Mock name='request.Request' spec='Request' id='...'/>

You can see that `request.Request` has a spec. `request.Request` takes two arguments in the constructor (one of which is `self`). Here’s what happens if we try to call it incorrectly:

>>> req = request.Request()
Traceback (most recent call last):
 ...
TypeError: <lambda>() takes at least 2 arguments (1 given)
The spec also applies to instantiated classes (i.e. the return value of specced mocks):

```python
>>> req = request.Request('foo')
>>> req
<NonCallableMagicMock name='request.Request()' spec='Request' id='...'>
```

`Request` objects are not callable, so the return value of instantiating our mocked out `request.Request` is a non-callable mock. With the spec in place any typos in our asserts will raise the correct error:

```python
>>> req.add_header('spam', 'eggs')
<MagicMock name='request.Request().add_header()' id='...'>
>>> req.add_header.assert_called_with
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assert_called_with'
```

In many cases you will just be able to add `autospec=True` to your existing `patch` calls and then be protected against bugs due to typos and api changes.

As well as using `autospec` through `patch` there is a `create_autospec()` for creating autospeced mocks directly:

```python
>>> from urllib import request
>>> mock_request = create_autospec(request)
>>> mock_request.Request('foo', 'bar')
<NonCallableMagicMock name='mock.Request()' spec='Request' id='...'>
```

This isn’t without caveats and limitations however, which is why it is not the default behaviour. In order to know what attributes are available on the spec object, autospec has to introspect (access attributes) the spec. As you traverse attributes on the mock a corresponding traversal of the original object is happening under the hood. If any of your specced objects have properties or descriptors that can trigger code execution then you may not be able to use autospec. On the other hand it is much better to design your objects so that introspection is safe.

A more serious problem is that it is common for instance attributes to be created in the `__init__` method and not to exist on the class at all. `autospec` can’t know about any dynamically created attributes and restricts the api to visible attributes.

```python
>>> class Something:
...    def __init__(self):
...        self.a = 33
...
>>> with patch('__main__.Something', autospec=True):
...    thing = Something()
...    thing.a
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

There are a few different ways of resolving this problem. The easiest, but not necessarily the least annoying, way is to simply set the required attributes on the mock after creation. Just because `autospec` doesn’t allow you to fetch attributes that don’t exist on the spec it doesn’t prevent you setting them:

```python
>>> with patch('__main__.Something', autospec=True):
...    thing = Something()
...    thing.a = 33
...
```

There is a more aggressive version of both `spec` and `autospec` that `does` prevent you setting non-existent attributes. This is useful if you want to ensure your code only `sets` valid attributes too, but obviously it prevents this particular scenario:

---

5 This only applies to classes or already instantiated objects. Calling a mocked class to create a mock instance does not create a real instance. It is only attribute lookups - along with calls to `dir` - that are done.
>>> with patch('__main__.Something', autospec=True, spec_set=True):
...    thing = Something()
...    thing.a = 33
...
Traceback (most recent call last):
...
AttributeError: Mock object has no attribute 'a'

Probably the best way of solving the problem is to add class attributes as default values for instance members initialised in `__init__`. Note that if you are only setting default attributes in `__init__` then providing them via class attributes (shared between instances of course) is faster too. e.g.

```python
class Something:
    a = 33
```

This brings up another issue. It is relatively common to provide a default value of `None` for members that will later be an object of a different type. `None` would be useless as a spec because it wouldn’t let you access any attributes or methods on it. As `None` is *never* going to be useful as a spec, and probably indicates a member that will normally of some other type, `autospec` doesn’t use a spec for members that are set to `None`. These will just be ordinary mocks (well - MagicMocks):

```python
>>> class Something:
...    member = None
...
>>> mock = create_autospec(Something)
>>> mock.member.foo.bar.baz()
<Mock name='mock.member.foo.bar.baz()' id='...'>
```

If modifying your production classes to add defaults isn’t to your liking then there are more options. One of these is simply to use an instance as the spec rather than the class. The other is to create a subclass of the production class and add the defaults to the subclass without affecting the production class. Both of these require you to use an alternative object as the spec. Thankfully `patch` supports this - you can simply pass the alternative object as the `autospec` argument:

```python
>>> class Something:
...    def __init__(self):
...        self.a = 33
...
>>> class SomethingForTest(Something):
...    a = 33
...
>>> p = patch('__main__.Something', autospec=SomethingForTest)
>>> mock = p.start()
>>> mock.a
<NonCallableMagicMock name='Something.a' spec='int' id='...'>
```

## 26.5 `unittest.mock` — getting started

New in version 3.3.

### 26.5.1 Using Mock

**Mock Patching Methods**

Common uses for `Mock` objects include:

- Patching methods
- Recording method calls on objects
You might want to replace a method on an object to check that it is called with the correct arguments by another part of the system:

```python
>>> real = SomeClass()
>>> real.method = MagicMock(name='method')
>>> real.method(3, 4, 5, key='value')
<Mock name='method()' id='...'>
```

Once our mock has been used (`real.method` in this example) it has methods and attributes that allow you to make assertions about how it has been used.

**Note:** In most of these examples the `Mock` and `MagicMock` classes are interchangeable. As the `MagicMock` is the more capable class it makes a sensible one to use by default.

Once the mock has been called its `called` attribute is set to `True`. More importantly we can use the `assert_called_with()` or `assert_called_once_with()` method to check that it was called with the correct arguments.

This example tests that calling `ProductionClass().method` results in a call to the `something` method:

```python
>>> class ProductionClass:
...     def method(self):
...         self.something(1, 2, 3)
...     def something(self, a, b, c):
...         pass
...
>>> real = ProductionClass()
>>> real.something = MagicMock()
>>> real.method()
>>> real.something.assert_called_once_with(1, 2, 3)
```

### Mock for Method Calls on an Object

In the last example we patched a method directly on an object to check that it was called correctly. Another common use case is to pass an object into a method (or some part of the system under test) and then check that it is used in the correct way.

The simple `ProductionClass` below has a `closer` method. If it is called with an object then it calls `close` on it.

```python
>>> class ProductionClass:
...     def closer(self, something):
...         something.close()
...
>>> real = ProductionClass()
>>> real.something = MagicMock()
>>> real.method()
>>> real.something.assert_called_once_with(1, 2, 3)
```

So to test it we need to pass in an object with a `close` method and check that it was called correctly.

```python
>>> real = ProductionClass()
>>> mock = Mock()
>>> real.closer(mock)
>>> mock.close.assert_called_with()
```

We don’t have to do any work to provide the ‘close’ method on our mock. Accessing close creates it. So, if ‘close’ hasn’t already been called then accessing it in the test will create it, but `assert_called_with()` will raise a failure exception.

### Mocking Classes

A common use case is to mock out classes instantiated by your code under test. When you patch a class, then that class is replaced with a mock. Instances are created by `calling the class`. This means you access the “mock instance” by looking at the return value of the mocked class.
In the example below we have a function `some_function` that instantiates `Foo` and calls a method on it. The call to `patch` replaces the class `Foo` with a mock. The `Foo` instance is the result of calling the mock, so it is configured by modifying the mock `return_value`.

```python
>>> def some_function():
...     instance = module.Foo()
...     return instance.method()
... 
>>> with patch('module.Foo') as mock:
...     instance = mock.return_value
...     instance.method.return_value = 'the result'
...     result = some_function()
...     assert result == 'the result'
```

### Naming your mocks

It can be useful to give your mocks a name. The name is shown in the `repr` of the mock and can be helpful when the mock appears in test failure messages. The name is also propagated to attributes or methods of the mock:

```python
>>> mock = MagicMock(name='foo')
>>> mock
<MagicMock name='foo' id='...'>
>>> mock.method
<MagicMock name='foo.method' id='...'>
```

### Tracking all Calls

Often you want to track more than a single call to a method. The `mock_calls` attribute records all calls to child attributes of the mock - and also to their children.

```python
>>> mock = MagicMock()
>>> mock.method()
<MagicMock name='mock.method()' id='...'>
>>> mock.attribute.method(10, x=53)
<MagicMock name='mock.attribute.method()' id='...'>
>>> mock.mock_calls
[call.method(), call.attribute.method(10, x=53)]
```

If you make an assertion about `mock_calls` and any unexpected methods have been called, then the assertion will fail. This is useful because as well as asserting that the calls you expected have been made, you are also checking that they were made in the right order and with no additional calls:

You use the `call` object to construct lists for comparing with `mock_calls`:

```python
>>> expected = [call.method(), call.attribute.method(10, x=53)]
>>> mock.mock_calls == expected
True
```

### Setting Return Values and Attributes

Setting the return values on a mock object is trivially easy:

```python
>>> mock = Mock()
>>> mock.return_value = 3
>>> mock()
3
```

Of course you can do the same for methods on the mock:
>>> mock = Mock()
>>> mock.method.return_value = 3
>>> mock.method()
3

The return value can also be set in the constructor:

>>> mock = Mock(return_value=3)
>>> mock()
3

If you need an attribute setting on your mock, just do it:

>>> mock = Mock()
>>> mock.x = 3
>>> mock.x
3

Sometimes you want to mock up a more complex situation, like for example mock.connection.cursor().execute("SELECT 1"). If we wanted this call to return a list, then we have to configure the result of the nested call.

We can use call to construct the set of calls in a “chained call” like this for easy assertion afterwards:

>>> mock = Mock()
>>> cursor = mock.connection.cursor.return_value
>>> cursor.execute.return_value = ['foo']
>>> mock.connection.cursor().execute("SELECT 1")
['foo']
>>> expected = call.connection.cursor().execute("SELECT 1").call_list()
>>> mock.mock_calls
[call.connection.cursor(), call.connection.cursor().execute('SELECT 1')]
>>> mock.mock_calls == expected
True

It is the call to .call_list() that turns our call object into a list of calls representing the chained calls.

Raising exceptions with mocks

A useful attribute is side_effect. If you set this to an exception class or instance then the exception will be raised when the mock is called.

>>> mock = Mock(side_effect=Exception('Boom!'))
>>> mock()
Traceback (most recent call last):
... Exception: Boom!

Side effect functions and iterables

side_effect can also be set to a function or an iterable. The use case for side_effect as an iterable is where your mock is going to be called several times, and you want each call to return a different value. When you set side_effect to an iterable every call to the mock returns the next value from the iterable:

>>> mock = MagicMock(side_effect=[4, 5, 6])
>>> mock()
4
>>> mock()
5
>>> mock()
6
For more advanced use cases, like dynamically varying the return values depending on what the mock is called with, `side_effect` can be a function. The function will be called with the same arguments as the mock. Whatever the function returns is what the call returns:

```python
>>> vals = {(1, 2): 1, (2, 3): 2}
>>> def side_effect(*args):
...     return vals[args]
... >>> mock = MagicMock(side_effect=side_effect)
... >>> mock(1, 2)
1
>>> mock(2, 3)
2
```

Creating a Mock from an Existing Object

One problem with over use of mocking is that it couples your tests to the implementation of your mocks rather than your real code. Suppose you have a class that implements `some_method`. In a test for another class, you provide a mock of this object that also provides `some_method`. If later you refactor the first class, so that it no longer has `some_method` - then your tests will continue to pass even though your code is now broken!

Mock allows you to provide an object as a specification for the mock, using the `spec` keyword argument. Accessing methods / attributes on the mock that don’t exist on your specification object will immediately raise an attribute error. If you change the implementation of your specification, then tests that use that class will start failing immediately without you having to instantiate the class in those tests.

```python
>>> mock = Mock(spec=SomeClass)
>>> mock.old_method()
AttributeError: object has no attribute ‘old_method’
```

If you want a stronger form of specification that prevents the setting of arbitrary attributes as well as the getting of them then you can use `spec_set` instead of `spec`.

26.5.2 Patch Decorators

**Note:** With `patch` it matters that you patch objects in the namespace where they are looked up. This is normally straightforward, but for a quick guide read `where to patch`.

A common need in tests is to patch a class attribute or a module attribute, for example patching a builtin or patching a class in a module to test that it is instantiated. Modules and classes are effectively global, so patching on them has to be undone after the test or the patch will persist into other tests and cause hard to diagnose problems.

mock provides three convenient decorators for this: `patch`, `patch.object` and `patch.dict`. `patch` takes a single string, of the form `package.module.Class.attribute` to specify the attribute you are patching. It also optionally takes a value that you want the attribute (or class or whatever) to be replaced with. ‘patch.object’ takes an object and the name of the attribute you would like patched, plus optionally the value to patch it with.

```python
>>> original = SomeClass.attribute
>>> @patch.object(SomeClass, 'attribute', sentinel.attribute)
...     def test():
...         assert SomeClass.attribute == sentinel.attribute
... >>> test()
>>> assert SomeClass.attribute == original
```
>>> @patch('package.module.attribute', sentinel.attribute)
...     def test():
...         from package.module import attribute
...         assert attribute is sentinel.attribute
...
>>> test()

If you are patching a module (including builtins) then use patch instead of patch.object:

>>> mock = MagicMock(return_value=sentinel.file_handle)
>>> with patch('builtins.open', mock):
...     handle = open('filename', 'r')
...
>>> mock.assert_called_with('filename', 'r')
>>> assert handle == sentinel.file_handle, "incorrect file handle returned"

The module name can be ‘dotted’, in the form package.module if needed:

>>> @patch('package.module.ClassName.attribute', sentinel.attribute)
...     def test():
...         from package.module import ClassName
...         assert ClassName.attribute == sentinel.attribute
...
>>> test()

A nice pattern is to actually decorate test methods themselves:

>>> class MyTest(unittest2.TestCase):
...     @patch.object(SomeClass, 'attribute', sentinel.attribute)
...     def test_something(self):
...         self.assertEqual(SomeClass.attribute, sentinel.attribute)
...
>>> original = SomeClass.attribute
>>> MyTest('test_something').test_something()
>>> assert SomeClass.attribute == original

If you want to patch with a Mock, you can use patch with only one argument (or patch.object with two arguments). The mock will be created for you and passed into the test function / method:

>>> class MyTest(unittest2.TestCase):
...     @patch.object(SomeClass, 'static_method')
...     def test_something(self, mock_method):
...         SomeClass.static_method()
...         mock_method.assert_called_with()
...
>>> MyTest('test_something').test_something()

You can stack up multiple patch decorators using this pattern:

>>> class MyTest(unittest2.TestCase):
...     @patch('package.module.ClassName1')
...     @patch('package.module.ClassName2')
...     def test_something(self, MockClass2, MockClass1):
...         self.assertIs(package.module.ClassName1, MockClass1)
...         self.assertIs(package.module.ClassName2, MockClass2)
...
>>> MyTest('test_something').test_something()

When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal python order that decorators are applied). This means from the bottom up, so in the example above the mock for test_module.ClassName2 is passed in first.

There is also patch.dict() for setting values in a dictionary just during a scope and restoring the dictionary to its original state when the test ends:
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original

`patch`, `patch.object` and `patch.dict` can all be used as context managers.

Where you use `patch` to create a mock for you, you can get a reference to the mock using the “as” form of the with statement:

```python
>>> class ProductionClass:
...     def method(self):
...         pass
...
>>> with patch.object(ProductionClass, 'method') as mock_method:
...     mock_method.return_value = None
...     real = ProductionClass()
...     real.method(1, 2, 3)
...
>>> mock_method.assert_called_with(1, 2, 3)
```

As an alternative `patch`, `patch.object` and `patch.dict` can be used as class decorators. When used in this way it is the same as applying the decorator individually to every method whose name starts with "test".

### 26.5.3 Further Examples

Here are some more examples for some slightly more advanced scenarios.

**Mocking chained calls**

Mocking chained calls is actually straightforward with mock once you understand the `return_value` attribute. When a mock is called for the first time, or you fetch its `return_value` before it has been called, a new `Mock` is created.

This means that you can see how the object returned from a call to a mocked object has been used by interrogating the `return_value` mock:

```python
>>> mock = Mock()  
>>> mock().foo(a=2, b=3)  
<Mock name='mock().foo()' id='...'>  
>>> mock.return_value.foo.assert_called_with(a=2, b=3)
```

From here it is a simple step to configure and then make assertions about chained calls. Of course another alternative is writing your code in a more testable way in the first place...

So, suppose we have some code that looks a little bit like this:

```python
>>> class Something:
...     def __init__(self):
...         self.backend = BackendProvider()  
...     def method(self):
...         response = self.backend.get_endpoint('foobar').create_call('spam', 'eggs').start_call()  
...         # more code
```

Assuming that `BackendProvider` is already well tested, how do we test `method()`? Specifically, we want to test that the code section `# more code` uses the response object in the correct way.

As this chain of calls is made from an instance attribute we can monkey patch the `backend` attribute on a `Something` instance. In this particular case we are only interested in the return value from the final call to `start_call` so we
don’t have much configuration to do. Let’s assume the object it returns is ‘file-like’, so we’ll ensure that our
response object uses the builtin file as its spec.
To do this we create a mock instance as our mock backend and create a mock response object for it. To set the
response as the return value for that final start_call we could do this:

```
mock_backend.get_endpoint.return_value.create_call.return_value.start_call.return_value =
mock_response.
```

We can do that in a slightly nicer way using the configure_mock() method to directly set the return value for
us:

```python
>>> something = Something()
>>> mock_response = Mock(spec=file)
>>> mock_backend = Mock()
>>> config = {'get_endpoint.return_value.create_call.return_value.start_call.return_value': mock_response}
>>> mock_backend.configure_mock(**config)
```

With these we monkey patch the “mock backend” in place and can make the real call:

```python
>>> something.backend = mock_backend
>>> something.method()
```

Using mock_calls we can check the chained call with a single assert. A chained call is several calls in one line
of code, so there will be several entries in mock_calls. We can use call.call_list() to create this list of
calls for us:

```python
>>> chained = call.get_endpoint('foobar').create_call('spam', 'eggs').start_call()
>>> call_list = chained.call_list()
>>> assert mock_backend.mock_calls == call_list
```

Partial mocking

In some tests I wanted to mock out a call to datetime.date.today() to return a known date, but I didn’t want to
prevent the code under test from creating new date objects. Unfortunately datetime.date is written in C, and so I
couldn’t just monkey-patch out the static date.today method.

I found a simple way of doing this that involved effectively wrapping the date class with a mock, but passing
through calls to the constructor to the real class (and returning real instances).

The patch decorator is used here to mock out the date class in the module under test. The side_effect
attribute on the mock date class is then set to a lambda function that returns a real date. When the mock date class
is called a real date will be constructed and returned by side_effect.

```python
>>> from datetime import date
>>> with patch('mymodule.date') as mock_date:
...     mock_date.today.return_value = date(2010, 10, 8)
...     mock_date.side_effect = lambda *args, **kw: date(*args, **kw)
...     ...
...     assert mymodule.date.today() == date(2010, 10, 8)
...     assert mymodule.date(2009, 6, 8) == date(2009, 6, 8)
...     ...
```

Note that we don’t patch datetime.date globally, we patch date in the module that uses it. See where to patch.

When date.today() is called a known date is returned, but calls to the date(...) constructor still return normal dates.
Without this you can find yourself having to calculate an expected result using exactly the same algorithm as the
code under test, which is a classic testing anti-pattern.

Calls to the date constructor are recorded in the mock_date attributes (call_count and friends) which may also be
useful for your tests.

An alternative way of dealing with mocking dates, or other builtin classes, is discussed in this blog entry.
Mocking a Generator Method

A Python generator is a function or method that uses the `yield` statement to return a series of values when iterated over.

A generator method / function is called to return the generator object. It is the generator object that is then iterated over. The protocol method for iteration is `__iter__`, so we can mock this using a `MagicMock`.

Here’s an example class with an “iter” method implemented as a generator:

```python
>>> class Foo:
...     def iter(self):
...         for i in [1, 2, 3]:
...             yield i
... >>> foo = Foo()
>>> list(foo.iter())
[1, 2, 3]
```

How would we mock this class, and in particular its “iter” method?

To configure the values returned from the iteration (implicit in the call to `list`), we need to configure the object returned by the call to `foo.iter()`.

```python
>>> mock_foo = MagicMock()
>>> mock_foo.iter.return_value = iter([1, 2, 3])
>>> list(mock_foo.iter())
[1, 2, 3]
```

Applying the same patch to every test method

If you want several patches in place for multiple test methods the obvious way is to apply the patch decorators to every method. This can feel like unnecessary repetition. For Python 2.6 or more recent you can use `patch` (in all its various forms) as a class decorator. This applies the patches to all test methods on the class. A test method is identified by methods whose names start with `test`:

```python
>>> @patch('mymodule.SomeClass')
... class MyTest(TestCase):
...     def test_one(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...     def test_two(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...     def not_a_test(self):
...         return 'something'
... >>> MyTest('test_one').test_one()
>>> MyTest('test_two').test_two()
>>> MyTest('test_two').not_a_test()
'something'
```

An alternative way of managing patches is to use the `patch methods: start and stop`. These allow you to move the patching into your `setUp` and `tearDown` methods.

```python
>>> class MyTest(TestCase):
...     def setUp(self):
...         self.patcher = patch('mymodule.foo')
...         self.mock_foo = self.patcher.start()
...     def tearDown(self):
...         self.mock_foo = self.patcher.stop()
... >>> MyTest('test_one').test_one()
... >>> MyTest('test_two').test_two()
... >>> MyTest('test_two').not_a_test()
... 'something'
```

6 There are also generator expressions and more advanced uses of generators, but we aren’t concerned about them here. A very good introduction to generators and how powerful they are is: Generator Tricks for Systems Programmers.
def test_foo(self):
    self.assertIs(mymodule.foo, self.mock_foo)

def tearDown(self):
    self.patcher.stop()

MyTest('test_foo').run()

Mocking Unbound Methods

Whilst writing tests today I needed to patch an unbound method (patching the method on the class rather than on the instance). I needed self to be passed in as the first argument because I want to make asserts about which objects were calling this particular method. The issue is that you can’t patch with a mock for this, because if you replace an unbound method with a mock it doesn’t become a bound method when fetched from the instance, and so it doesn’t get self passed in. The workaround is to patch the unbound method with a real function instead. The patch() decorator makes it so simple to patch out methods with a mock that having to create a real function becomes a nuisance.

If you pass autospec=True to patch then it does the patching with a real function object. This function object has the same signature as the one it is replacing, but delegates to a mock under the hood. You still get your mock auto-created in exactly the same way as before. What it means though, is that if you use it to patch out an unbound method on a class the mocked function will be turned into a bound method if it is fetched from an instance. It will have self passed in as the first argument, which is exactly what I wanted:

class Foo:
    def foo(self):
        pass

with patch.object(Foo, 'foo', autospec=True) as mock_foo:
    mock_foo.return_value = 'foo'
    foo = Foo()
    foo.foo()

'foo'

mock_foo.assert_called_once_with(foo)

If we don’t use autospec=True then the unbound method is patched out with a Mock instance instead, and isn’t called with self.

Checking multiple calls with mock

mock has a nice API for making assertions about how your mock objects are used.
>>> mock = Mock()
>>> mock.foo_bar.return_value = None
>>> mock.foo_bar('baz', spam='eggs')
>>> mock.foo_bar.assert_called_with('baz', spam='eggs')
If your mock is only being called once you can use the `assert_called_once_with()` method that also
asserts that the call_count is one.

```python
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
Traceback (most recent call last):
  ...  
AssertionError: Expected to be called once. Called 2 times.
```
Both `assert_called_with` and `assert_called_once_with` make assertions about the *most recent* call. If your
mock is going to be called several times, and you want to make assertions about *all* those calls you can use
`call_args_list`:

```python
>>> mock = Mock(return_value=None)
>>> mock(1, 2, 3)
>>> mock(4, 5, 6)
>>> mock()
>>> mock.call_args_list
[call(1, 2, 3), call(4, 5, 6), call()]
```
The `call` helper makes it easy to make assertions about these calls. You can build up a list of expected calls and
compare it to `call_args_list`. This looks remarkably similar to the repr of the `call_args_list`:

```python
>>> expected = [call(1, 2, 3), call(4, 5, 6), call()]
>>> mock.call_args_list == expected
True
```

**Coping with mutable arguments**

Another situation is rare, but can bite you, is when your mock is called with mutable arguments. `call_args` and
`call_args_list` store references to the arguments. If the arguments are mutated by the code under test then you can
no longer make assertions about what the values were when the mock was called.

Here’s some example code that shows the problem. Imagine the following functions defined in ‘mymodule’:

```python
def frob(val):
    pass

def grob(val):
    "First frob and then clear val"
    frob(val)
    val.clear()
```

When we try to test that `grob` calls `frob` with the correct argument look what happens:

```python
>>> with patch('mymodule.frob') as mock_frob:
...     val = set([6])
...     mymodule.grob(val)
...     ...
```
```python
Traceback (most recent call last):
  ...
AssertionError: Expected: ((set([6]),), {})
Called with: ((set([6]),), {})
```
One possibility would be for mock to copy the arguments you pass in. This could then cause problems if you do assertions that rely on object identity for equality.

Here’s one solution that uses the side_effect functionality. If you provide a side_effect function for a mock then side_effect will be called with the same args as the mock. This gives us an opportunity to copy the arguments and store them for later assertions. In this example I’m using another mock to store the arguments so that I can use the mock methods for doing the assertion. Again a helper function sets this up for me.

```python
>>> from copy import deepcopy
>>> from unittest.mock import Mock, patch, DEFAULT
>>> def copy_call_args(mock):
...     new_mock = Mock()
...     def side_effect(*args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         new_mock(*args, **kwargs)
...         return DEFAULT
...     mock.side_effect = side_effect
...     return new_mock
...
>>> with patch('mymodule.frob') as mock_frob:
...     new_mock = copy_call_args(mock_frob)
...     val = set([6])
...     mymodule.grob(val)
...     new_mock.assert_called_with(set([6]))
>>> new_mock.call_args
<Mock mock object>

Mock method call arguments are copied, and you can use the mock methods to make assertions.

Note: If your mock is only going to be used once there is an easier way of checking arguments at the point they are called. You can simply do the checking inside a side_effect function.

```python
>>> def side_effect(arg):
...     assert arg == set([6])
...     ...
>>> mock = Mock(side_effect=side_effect)
>>> mock(set([6]))
>>> mock(set())
Traceback (most recent call last):
  ...
AssertionError
```

An alternative approach is to create a subclass of Mock or MagicMock that copies (using copy.deepcopy()) the arguments. Here’s an example implementation:

```python
>>> from copy import deepcopy
>>> class CopyingMock(MagicMock):
...     def __call__(self, *args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         return super(CopyingMock, self).__call__(*args, **kwargs)
...
>>> c = CopyingMock(return_value=None)
>>> arg = set()
>>> c(arg)
>>> c.assert_called_with(set())
```
c.assert_called_with(arg)

Traceback (most recent call last):
...
AssertionError: Expected call: mock(set([1]))
Actual call: mock(set([]))

When you subclass Mock or MagicMock all dynamically created attributes, and the return_value will use your subclass automatically. That means all children of a CopyingMock will also have the type CopyingMock.

Nesting Patches

Using patch as a context manager is nice, but if you do multiple patches you can end up with nested with statements indenting further and further to the right:

class MyTest(Test Case):
  ...
  ...
  ...  def test_foo(self):
  ...  
  ...  
  ...  
  ...  
  ...  
  ...  
  ...  
  ...  
  ...  
  ...  
  ...  
  ...  
  ...  
  ...  
  ...  
  
With unittest cleanup functions and the patch methods: start and stop we can achieve the same effect without the nested indentation. A simple helper method, create_patch, puts the patch in place and returns the created mock for us:

class MyTest(Test Case):
  ...
  ...
  ...
  ...  def create_patch(self, name):
  ...  
  ...  
  ...  
  ...  
  ...  
  
Mocking a dictionary with MagicMock

You may want to mock a dictionary, or other container object, recording all access to it whilst having it still behave like a dictionary.
We can do this with `Mock`, which will behave like a dictionary, and using `side_effect` to delegate dictionary access to a real underlying dictionary that is under our control.

When the `__getitem__` and `__setitem__` methods of our `MagicMock` are called (normal dictionary access) then `side_effect` is called with the key (and in the case of `__setitem__` the value too). We can also control what is returned.

After the `MagicMock` has been used we can use attributes like `call_args_list` to assert about how the dictionary was used:

```python
def getitem(name):
    return my_dict[name]

def setitem(name, val):
    my_dict[name] = val

mock = MagicMock()
mock.__getitem__.side_effect = getitem
mock.__setitem__.side_effect = setitem

my_dict = {'a': 1, 'b': 2, 'c': 3}
```

Note: An alternative to using `MagicMock` is to use `Mock` and only provide the magic methods you specifically want:

```python
mock = Mock()
mock.__setitem__ = Mock(side_effect=getitem)
mock.__getitem__ = Mock(side_effect=setitem)
```

A third option is to use `MagicMock` but passing in `dict` as the `spec` (or `spec_set`) argument so that the `MagicMock` created only has dictionary magic methods available:

```python
mock = MagicMock(spec_set=dict)
mock.__getitem__.side_effect = getitem
```

With these side effect functions in place, the `mock` will behave like a normal dictionary but recording the access. It even raises a `KeyError` if you try to access a key that doesn’t exist.

```python
mock['a']
1
mock['c']
3
mock['d']
Traceback (most recent call last):
  ...
KeyError: 'd'
mock['b'] = 'fish'
mock['d'] = 'eggs'
mock['b']
'fish'
mock['d']
'eggs'
```

After it has been used you can make assertions about the access using the normal mock methods and attributes:

```python
mock.__getitem__.call_args_list
[call('a'), call('c'), call('d'), call('b'), call('d')]
mock.__setitem__.call_args_list
[call('b', 'fish'), call('d', 'eggs')]
mock.call_args_list
[('a', 1), ('c', 3), ('b', 'fish'), ('d', 'eggs')]
```
Mock subclasses and their attributes

There are various reasons why you might want to subclass `Mock`. One reason might be to add helper methods. Here’s a silly example:

```python
>>> class MyMock(MagicMock):
...     def has Beencalled(self):
...         return self.called
...
>>> mymock = MyMock(return_value=None)

>>> mymock
<MyMock id='...'>

>>> mymock.has_Been_called()
False

>>> mymock()

>>> mymock.has_Been_called()
True
```

The standard behaviour for `Mock` instances is that attributes and the return value mocks are of the same type as the mock they are accessed on. This ensures that `Mock` attributes are `Mocks` and `MagicMock` attributes are `MagicMocks`\(^7\). So if you’re subclassing to add helper methods then they’ll also be available on the attributes and return value mock of instances of your subclass.

```python
>>> mymock.foo
<MyMock name='mock.foo' id='...'>

>>> mymock.foo.has_Been_called()
False

>>> mymock.foo()

>>> mymock.foo.has_Been_called()
True
```

Sometimes this is inconvenient. For example, one user is subclassing mock to created a Twisted adaptor. Having this applied to attributes too actually causes errors.

Mock (in all its flavours) uses a method called `_get_child_mock` to create these “sub-mocks” for attributes and return values. You can prevent your subclass being used for attributes by overriding this method. The signature is that it takes arbitrary keyword arguments (**kwargs) which are then passed onto the mock constructor:

```python
>>> class Subclass(MagicMock):
...     def _get_child_mock(self, **kwargs):
...         return MagicMock(**kwargs)
...
>>> mymock = Subclass()

>>> mymock.foo

>>> assert isinstance(mymock, Subclass)

>>> assert not isinstance(mymock.foo, Subclass)

>>> assert not isinstance(mymock(), Subclass)
```

Mocking imports with patch.dict

One situation where mocking can be hard is where you have a local import inside a function. These are harder to mock because they aren’t using an object from the module namespace that we can patch out.

Generally local imports are to be avoided. They are sometimes done to prevent circular dependencies, for which there is usually a much better way to solve the problem (refactor the code) or to prevent “up front costs” by delaying the import. This can also be solved in better ways than an unconditional local import (store the module as a class or module attribute and only do the import on first use).

\(^7\) An exception to this rule are the non-callable mocks. Attributes use the callable variant because otherwise non-callable mocks couldn’t have callable methods.
That aside there is a way to use mock to affect the results of an import. Importing fetches an object from the `sys.modules` dictionary. Note that it fetches an object, which need not be a module. Importing a module for the first time results in a module object being put in `sys.modules`, so usually when you import something you get a module back. This need not be the case however.

This means you can use `patch.dict()` to temporarily put a mock in place in `sys.modules`. Any imports whilst this patch is active will fetch the mock. When the patch is complete (the decorated function exits, the with statement body is complete or `patcher.stop()` is called) then whatever was there previously will be restored safely.

Here’s an example that mocks out the ‘fooble’ module.

```python
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     import fooble
...     fooble.blob()
...     <Mock name='mock.blob()' id='...'>
>>> assert 'fooble' not in sys.modules
>>> mock.blob.assert_called_once_with()
```

As you can see the `import fooble` succeeds, but on exit there is no ‘fooble’ left in `sys.modules`.

This also works for the `from module import name` form:

```python
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     from fooble import blob
...     blob.blip()
...     <Mock name='mock.blob.blip()' id='...'>
>>> mock.blob.blip.assert_called_once_with()
```

With slightly more work you can also mock package imports:

```python
>>> mock = Mock()
>>> modules = {'package': mock, 'package.module': mock.module}
>>> with patch.dict('sys.modules', modules):
...     from package.module import fooble
...     fooble()
...     <Mock name='mock.module.fooble()' id='...'>
>>> mock.module.fooble.assert_called_once_with()
```

### Tracking order of calls and less verbose call assertions

The `Mock` class allows you to track the order of method calls on your mock objects through the `method_calls` attribute. This doesn’t allow you to track the order of calls between separate mock objects, however we can use `mock_calls` to achieve the same effect.

Because mocks track calls to child mocks in `mock_calls`, and accessing an arbitrary attribute of a mock creates a child mock, we can create our separate mocks from a parent one. Calls to those child mock will then all be recorded, in order, in the `mock_calls` of the parent:

```python
>>> manager = Mock()
>>> mock_foo = manager.foo
>>> mock_bar = manager.bar
>>> mock_foo.something()
<Mock name='mock.foo.something()' id='...'>
>>> mock_bar.other.thing()
<Mock name='mock.bar.other.thing()' id='...'>
```
>>> manager.mock_calls
[call.foo.something(), call.bar.other.thing()]

We can then assert about the calls, including the order, by comparing with the mock_calls attribute on the manager mock:

>>> expected_calls = [call.foo.something(), call.bar.other.thing()]
>>> manager.mock_calls == expected_calls
True

If patch is creating, and putting in place, your mocks then you can attach them to a manager mock using the attach_mock() method. After attaching calls will be recorded in mock_calls of the manager.

>>> manager = MagicMock()
>>> with patch('mymodule.Class1') as MockClass1:
...     with patch('mymodule.Class2') as MockClass2:
...         manager.attach_mock(MockClass1, 'MockClass1')
...         manager.attach_mock(MockClass2, 'MockClass2')
...         MockClass1().foo()
...         MockClass2().bar()
...<MagicMock name='mock.MockClass1().foo()' id='...'>
<MagicMock name='mock.MockClass2().bar()' id='...'>

>>> manager.mock_calls
[call.MockClass1(),
call.MockClass1().foo(),
call.MockClass2(),
call.MockClass2().bar()]

If many calls have been made, but you’re only interested in a particular sequence of them then an alternative is to use the assert_has_calls() method. This takes a list of calls (constructed with the call object). If that sequence of calls are in mock_calls then the assert succeeds.

>>> m = MagicMock()
>>> m().foo().bar().baz()
(...)
>>> calls = [call.fifty('50'), call(1), call.seven(7)]

More complex argument matching

Using the same basic concept as ANY we can implement matchers to do more complex assertions on objects used as arguments to mocks.

Suppose we expect some object to be passed to a mock that by default compares equal based on object identity (which is the Python default for user defined classes). To use assert_called_with() we would need to pass in the exact same object. If we are only interested in some of the attributes of this object then we can create a matcher that will check these attributes for us.
You can see in this example how a ‘standard’ call to `assert_called_with` isn’t sufficient:

```python
>>> class Foo:
...     def __init__(self, a, b):
...         self.a, self.b = a, b
... >>> mock = Mock(return_value=None)
>>> mock(Foo(1, 2))
>>> mock.assert_called_with(Foo(1, 2))
Traceback (most recent call last):
  ...  
AssertionError: Expected: call(<__main__.Foo object at 0x...>)
Actual call: call(<__main__.Foo object at 0x...>)
```

A comparison function for our `Foo` class might look something like this:

```python
>>> def compare(self, other):
...     if not type(self) == type(other):
...         return False
...     if self.a != other.a:
...         return False
...     if self.b != other.b:
...         return False
...     return True
... >>>
```

And a matcher object that can use comparison functions like this for its equality operation would look something like this:

```python
>>> class Matcher:
...     def __init__(self, compare, some_obj):
...         self.compare = compare
...         self.some_obj = some_obj
...     def __eq__(self, other):
...         return self.compare(self.some_obj, other)
... >>>
```

Putting all this together:

```python
>>> match_foo = Matcher(compare, Foo(1, 2))
>>> mock.assert_called_with(match_foo)
```

The `Matcher` is instantiated with our compare function and the `Foo` object we want to compare against. In `assert_called_with` the `Matcher` equality method will be called, which compares the object the mock was called with against the one we created our matcher with. If they match then `assert_called_with` passes, and if they don’t an `AssertionError` is raised:

```python
>>> match_wrong = Matcher(compare, Foo(3, 4))
>>> mock.assert_called_with(match_wrong)
```

With a bit of tweaking you could have the comparison function raise the `AssertionError` directly and provide a more useful failure message.

As of version 1.5, the Python testing library PyHamcrest provides similar functionality, that may be useful here, in the form of its equality matcher (`hamcrest.library.integration.match_equality`).
26.6 2to3 - Automated Python 2 to 3 code translation

2to3 is a Python program that reads Python 2.x source code and applies a series of fixers to transform it into valid Python 3.x code. The standard library contains a rich set of fixers that will handle almost all code. 2to3 supporting library lib2to3 is, however, a flexible and generic library, so it is possible to write your own fixers for 2to3. lib2to3 could also be adapted to custom applications in which Python code needs to be edited automatically.

26.6.1 Using 2to3

2to3 will usually be installed with the Python interpreter as a script. It is also located in the Tools/scripts directory of the Python root.

2to3’s basic arguments are a list of files or directories to transform. The directories are recursively traversed for Python sources.

Here is a sample Python 2.x source file, example.py:

```python
def greet(name):
    print "Hello, (0)!".format(name)
print "What’s your name?"
name = raw_input()
greet(name)
```

It can be converted to Python 3.x code via 2to3 on the command line:

```
$ 2to3 example.py
```

A diff against the original source file is printed. 2to3 can also write the needed modifications right back to the source file. (A backup of the original file is made unless -n is also given.) Writing the changes back is enabled with the -w flag:

```
$ 2to3 -w example.py
```

After transformation, example.py looks like this:

```python
def greet(name):
    print("Hello, (0)!".format(name))
print("What’s your name?")
name = input()
greet(name)
```

Comments and exact indentation are preserved throughout the translation process.

By default, 2to3 runs a set of predefined fixers. The -l flag lists all available fixers. An explicit set of fixers to run can be given with -f. Likewise the -x explicitly disables a fixer. The following example runs only the imports and has_key fixers:

```
$ 2to3 -f imports -f has_key example.py
```

This command runs every fixer except the apply fixer:

```
$ 2to3 -x apply example.py
```

Some fixers are explicit, meaning they aren’t run by default and must be listed on the command line to be run. Here, in addition to the default fixers, the idioms fixer is run:

```
$ 2to3 -f all -f idioms example.py
```

Notice how passing all enables all default fixers.

Sometimes 2to3 will find a place in your source code that needs to be changed, but 2to3 cannot fix automatically. In this case, 2to3 will print a warning beneath the diff for a file. You should address the warning in order to have compliant 3.x code.
2to3 can also refactor doctests. To enable this mode, use the `-d` flag. Note that only doctests will be refactored. This also doesn’t require the module to be valid Python. For example, doctest like examples in a reST document could also be refactored with this option.

The `-v` option enables output of more information on the translation process.

Since some print statements can be parsed as function calls or statements, 2to3 cannot always read files containing the print function. When 2to3 detects the presence of the `from __future__ import print_function` compiler directive, it modifies its internal grammar to interpret `print()` as a function. This change can also be enabled manually with the `-p` flag. Use `-p` to run fixers on code that already has had its print statements converted.

The `-o` or `--output-dir` option allows specification of an alternate directory for processed output files to be written to. The `-n` flag is required when using this as backup files do not make sense when not overwriting the input files. New in version 3.2.3: The `-o` option was added. The `-W` or `--write-unchanged-files` flag tells 2to3 to always write output files even if no changes were required to the file. This is most useful with `-o` so that an entire Python source tree is copied with translation from one directory to another. This option implies the `-w` flag as it would not make sense otherwise. New in version 3.2.3: The `-W` flag was added. The `--add-suffix` option specifies a string to append to all output filenames. The `-n` flag is required when specifying this as backups are not necessary when writing to different filenames. Example:

```
$ 2to3 -n -W --add-suffix=3 example.py
```

Will cause a converted file named `example.py3` to be written. New in version 3.2.3: The `--add-suffix` option was added. To translate an entire project from one directory tree to another use:

```
$ 2to3 --output-dir=python3-version/mycode -W -n python2-version/mycode
```

### 26.6.2 Fixers

Each step of transforming code is encapsulated in a fixer. The command `2to3 -l` lists them. As documented above, each can be turned on and off individually. They are described here in more detail.

**apply**

Removes usage of `apply()`. For example `apply(function, *args, **kwargs)` is converted to `function(*args, **kwargs)`.

**basestring**

Converts `basestring` to `str`.

**buffer**

Converts `buffer` to `memoryview`. This fixer is optional because the `memoryview` API is similar but not exactly the same as that of `buffer`.

**callable**

Converts `callable(x)` to `isinstance(x, collections.Callable)`, adding an import to `collections` if needed. Note `callable(x)` has returned in Python 3.2, so if you do not intend to support Python 3.1, you can disable this fixer.

**dict**

Fixes dictionary iteration methods. `dict.iteritems()` is converted to `dict.items()`, `dict.iterkeys()` to `dict.keys()`, and `dict.itervalues()` to `dict.values()`. Similarly, `dict.viewitems()`, `dict.viewkeys()` and `dict.viewvalues()` are converted respectively to `dict.items()`, `dict.keys()` and `dict.values()`. It also wraps existing usages of `dict.items()`, `dict.keys()` and `dict.values()` in a call to `list`.

**except**

Converts `except X, T` to `except X as T`.

**exec**

Converts the `exec` statement to the `exec()` function.
execfile
Removes usage of execfile(). The argument to execfile() is wrapped in calls to open(), compile(), and exec().

exitfunc
Changes assignment of sys.exitfunc to use of the atexit module.

filter
Wraps filter() usage in a list call.

funcattrs
Fixes function attributes that have been renamed. For example, my_function.func_closure is converted to my_function.__closure__.

future
Removes from __future__ import new_feature statements.

getcwd
 Renames os.getcwdu() to os.getcwd().

has_key
Changes dict.has_key(key) to key in dict.

idioms
This optional fixer performs several transformations that make Python code more idiomatic. Type comparisons like type(x) is SomeClass and type(x) == SomeClass are converted to isinstance(x, SomeClass). While 1 becomes while True. This fixer also tries to make use of sorted() in appropriate places. For example, this block

L = list(some_iterable)
L.sort()

is changed to

L = sorted(some_iterable)

import
Detects sibling imports and converts them to relative imports.

imports
Handles module renames in the standard library.

imports2
Handles other modules renames in the standard library. It is separate from the imports fixer only because of technical limitations.

input
Converts input(prompt) to eval(input(prompt))

intern
Converts intern() to sys.intern().

isinstance
Fixes duplicate types in the second argument of isinstance(). For example, isinstance(x, (int, int)) is converted to isinstance(x, (int)).

itertools_imports
Removes imports of itertools.ifilter(), itertools.izip(), and itertools.imap(). Imports of itertools.ifilterfalse() are also changed to itertools.filterfalse().

itertools
Changes usage of itertools.ifilter(), itertools.izip(), and itertools.imap() to their built-in equivalents. itertools.ifilterfalse() is changed to itertools.filterfalse().

26.6. 2to3 - Automated Python 2 to 3 code translation

1171
long
Renames long to int.

map
Wraps map() in a list call. It also changes map(None, x) to list(x). Using from future_builtins import map disables this fixer.

metaclass
Converts the old metaclass syntax (__metaclass__ = Meta in the class body) to the new (class X(metaclass=Meta)).

methodattrs
Fixes old method attribute names. For example, meth.im_func is converted to meth.__func__.

ne
Converts the old not-equal syntax, <>, to !=.

next
Converts the use of iterator’s next() methods to the next() function. It also renames next() methods to __next__().

nonzero
Renames __nonzero__() to __bool__().

numliterals
Converts octal literals into the new syntax.

operator
Converts calls to various functions in the operator module to other, but equivalent, function calls. When needed, the appropriate import statements are added, e.g. import collections. The following mapping are made:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator.isCallable(obj)</td>
<td>hasattr(obj, '<strong>call</strong>')</td>
</tr>
<tr>
<td>operator.sequenceIncludes(obj)</td>
<td>operator.contains(obj)</td>
</tr>
<tr>
<td>operator.isSequenceType(obj)</td>
<td>isinstance(obj, collections.Sequence)</td>
</tr>
<tr>
<td>operator.isMappingType(obj)</td>
<td>isinstance(obj, collections.Mapping)</td>
</tr>
<tr>
<td>operator.isNumberType(obj)</td>
<td>isinstance(obj, numbers.Number)</td>
</tr>
<tr>
<td>operator.repeat(obj, n)</td>
<td>operator.mul(obj, n)</td>
</tr>
<tr>
<td>operator.irepeat(obj, n)</td>
<td>operator.imul(obj, n)</td>
</tr>
</tbody>
</table>

paren
Add extra parenthesis where they are required in list comprehensions. For example, [x for x in 1, 2] becomes [x for x in (1, 2)].

print
Converts the print statement to the print() function.

raise
Converts raise E, V to raise E(V), and raise E, V, T to raise E(V).with_traceback(T). If E is a tuple, the translation will be incorrect because substituting tuples for exceptions has been removed in 3.0.

raw_input
Converts raw_input() to input().

reduce
Handles the move of reduce() to functools.reduce().

renames
Changes sys.maxint to sys.maxsize.

repr
Replaces backtick repr with the repr() function.

set_literal
Replaces use of the set constructor with set literals. This fixer is optional.
The Python Library Reference, Release 3.3.3

standard_error
   Renames `StandardError` to `Exception`.

sys_exc
   Changes the deprecated `sys.exc_value`, `sys.exc_type`, `sys.exc_traceback` to use
   `sys.exc_info()`.

throw
   Fixes the API change in generator’s `throw()` method.

tuple_params
   Removes implicit tuple parameter unpacking. This fixer inserts temporary variables.

types
   Fixes code broken from the removal of some members in the `types` module.

unicode
   Renames `unicode` to `str`.

urllib
   Handles the rename of `urllib` and `urllib2` to the `urllib` package.

ws_comma
   Removes excess whitespace from comma separated items. This fixer is optional.

xrange
   Renames `xrange()` to `range()` and wraps existing `range()` calls with `list`.

xreadlines
   Changes `for x in file.xreadlines()` to `for x in file`.

zip
   Wraps `zip()` usage in a `list` call. This is disabled when `from future_builtins import zip` appears.

26.6.3 lib2to3 - 2to3’s library

Note: The `lib2to3` API should be considered unstable and may change drastically in the future.

26.7 test — Regression tests package for Python

Note: The `test` package is meant for internal use by Python only. It is documented for the benefit of the core developers of Python. Any use of this package outside of Python’s standard library is discouraged as code mentioned here can change or be removed without notice between releases of Python.

The `test` package contains all regression tests for Python as well as the modules `test.support` and `test.regrtest`. `test.support` is used to enhance your tests while `test.regrtest` drives the testing suite.

Each module in the `test` package whose name starts with `test_` is a testing suite for a specific module or feature. All new tests should be written using the `unittest` or `doctest` module. Some older tests are written using a “traditional” testing style that compares output printed to `sys.stdout`; this style of test is considered deprecated.

See Also:

Module `unittest` Writing PyUnit regression tests.

Module `doctest` Tests embedded in documentation strings.
26.7.1 Writing Unit Tests for the test package

It is preferred that tests that use the unittest module follow a few guidelines. One is to name the test module by starting it with test_ and end it with the name of the module being tested. The test methods in the test module should start with test_ and end with a description of what the method is testing. This is needed so that the methods are recognized by the test driver as test methods. Also, no documentation string for the method should be included. A comment (such as # Tests function returns only True or False) should be used to provide documentation for test methods. This is done because documentation strings get printed out if they exist and thus what test is being run is not stated.

A basic boilerplate is often used:

```python
import unittest
from test import support

class MyTestCase1(unittest.TestCase):
    # Only use setUp() and tearDown() if necessary
    def setUp(self):
        ... code to execute in preparation for tests ...

    def tearDown(self):
        ... code to execute to clean up after tests ...

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...

    def test_feature_two(self):
        # Test feature two.
        ... testing code ...

    ... more test methods ...

class MyTestCase2(unittest.TestCase):
    ... same structure as MyTestCase1 ...

... more test classes ...

if __name__ == '__main__':
    unittest.main()
```

This code pattern allows the testing suite to be run by test.regrtest, on its own as a script that supports the unittest CLI, or via the python -m unittest CLI.

The goal for regression testing is to try to break code. This leads to a few guidelines to be followed:

- The testing suite should exercise all classes, functions, and constants. This includes not just the external API that is to be presented to the outside world but also “private” code.
- Whitebox testing (examining the code being tested when the tests are being written) is preferred. Blackbox testing (testing only the published user interface) is not complete enough to make sure all boundary and edge cases are tested.
- Make sure all possible values are tested including invalid ones. This makes sure that not only all valid values are acceptable but also that improper values are handled correctly.
- Exhaust as many code paths as possible. Test where branching occurs and thus tailor input to make sure as many different paths through the code are taken.
- Add an explicit test for any bugs discovered for the tested code. This will make sure that the error does not crop up again if the code is changed in the future.
• Make sure to clean up after your tests (such as close and remove all temporary files).
• If a test is dependent on a specific condition of the operating system then verify the condition already exists before attempting the test.
• Import as few modules as possible and do it as soon as possible. This minimizes external dependencies of tests and also minimizes possible anomalous behavior from side-effects of importing a module.
• Try to maximize code reuse. On occasion, tests will vary by something as small as what type of input is used. Minimize code duplication by subclassing a basic test class with a class that specifies the input:

```python
class TestFuncAcceptsSequencesMixin:
    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = [1, 2, 3]

class AcceptStrings(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = 'abc'

class AcceptTuples(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = (1, 2, 3)
```

When using this pattern, remember that all classes that inherit from `unittest.TestCase` are run as tests. The `Mixin` class in the example above does not have any data and so can’t be run by itself, thus it does not inherit from `unittest.TestCase`.

See Also:
Test Driven Development A book by Kent Beck on writing tests before code.

## 26.7.2 Running tests using the command-line interface

The `test` package can be run as a script to drive Python’s regression test suite, thanks to the `-m` option: `python -m test`. Under the hood, it uses `test.regrtest` (the call `python -m test.regrtest` used in previous Python versions still works). Running the script by itself automatically starts running all regression tests in the `test` package. It does this by finding all modules in the package whose name starts with `test_`, importing them, and executing the function `test_main()` if present or loading the tests via `unittest.TestLoader.loadTestsFromModule` if `test_main` does not exist. The names of tests to execute may also be passed to the script. Specifying a single regression test (`python -m test test_spam`) will minimize output and only print whether the test passed or failed.

Running `test` directly allows what resources are available for tests to use to be set. You do this by using the `-u` command-line option. Specifying `all` as the value for the `-u` option enables all possible resources: `python -m test -uall`. If all but one resource is desired (a more common case), a comma-separated list of resources that are not desired may be listed after `all`. The command `python -m test -uall,-audio,-largefile` will run `test` with all resources except the `audio` and `largefile` resources. For a list of all resources and more command-line options, run `python -m test -h`.

Some other ways to execute the regression tests depend on what platform the tests are being executed on. On Unix, you can run `make test` at the top-level directory where Python was built. On Windows, executing `rt.bat` from your `PCBuild` directory will run all regression tests.

## 26.8 test.support — Utilities for the Python test suite

The `test.support` module provides support for Python’s regression test suite.
Note: `test.support` is not a public module. It is documented here to help Python developers write tests. The API of this module is subject to change without backwards compatibility concerns between releases.

This module defines the following exceptions:

**exception** `test.support.TestFailed`
- Exception to be raised when a test fails. This is deprecated in favor of `unittest`-based tests and `unittest.TestCase`'s assertion methods.

**exception** `test.support.ResourceDenied`
- Subclass of `unittest.SkipTest`. Raised when a resource (such as a network connection) is not available. Raised by the `requires()` function.

The `test.support` module defines the following constants:

`test.support.verbose`
- True when verbose output is enabled. Should be checked when more detailed information is desired about a running test. `verbose` is set by `test.regrtest`.

`test.support.is_jython`
- True if the running interpreter is Jython.

`test.support.TESTFN`
- Set to a name that is safe to use as the name of a temporary file. Any temporary file that is created should be closed and unlinked (removed).

The `test.support` module defines the following functions:

`test.support.forget(module_name)`
- Remove the module named `module_name` from `sys.modules` and delete any byte-compiled files of the module.

`test.support.is_resource_enabled(resource)`
- Return True if `resource` is enabled and available. The list of available resources is only set when `test.regrtest` is executing the tests.

`test.support.requires(resource, msg=None)`
- Raise `ResourceDenied` if `resource` is not available. `msg` is the argument to `ResourceDenied` if it is raised. Always returns True if called by a function whose `__name__` is `'__main__'`. Used when tests are executed by `test.regrtest`.

`test.support.findfile(filename, subdir=None)`
- Return the path to the file named `filename`. If no match is found `filename` is returned. This does not equal a failure since it could be the path to the file.

Setting `subdir` indicates a relative path to use to find the file rather than looking directly in the path directories.

`test.support.run_unittest(*classes)`
- Execute `unittest.TestCase` subclasses passed to the function. The function scans the classes for methods starting with the prefix `test_` and executes the tests individually.

It is also legal to pass strings as parameters; these should be keys in `sys.modules`. Each associated module will be scanned by `unittest.TestLoader.loadTestsFromModule()`. This is usually seen in the following `test_main()` function:

```python
def test_main():
    support.run_unittest(__name__)
```

This will run all tests defined in the named module.

`test.support.run_doctest(module, verbosity=None)`
- Run `doctest.testmod()` on the given `module`. Return `(failure_count, test_count)`.

If `verbosity` is None, `doctest.testmod()` is run with verbosity set to `verbose`. Otherwise, it is run with verbosity set to `None`. 

Chapter 26. Development Tools
test.support.check_warnings(*filters, quiet=True)

A convenience wrapper for warnings.catch_warnings() that makes it easier to test that a warning was correctly raised. It is approximately equivalent to calling warnings.catch_warnings(record=True) with warnings.simplefilter() set to always and with the option to automatically validate the results that are recorded.

check_warnings accepts 2-tuples of the form ("message regexp", WarningCategory) as positional arguments. If one or more filters are provided, or if the optional keyword argument quiet is False, it checks to make sure the warnings are as expected: each specified filter must match at least one of the warnings raised by the enclosed code or the test fails, and if any warnings are raised that do not match any of the specified filters the test fails. To disable the first of these checks, set quiet to True.

If no arguments are specified, it defaults to:

check_warnings("", Warning), quiet=True)

In this case all warnings are caught and no errors are raised.

On entry to the context manager, a WarningRecorder instance is returned. The underlying warnings list from catch_warnings() is available via the recorder object’s warnings attribute. As a convenience, the attributes of the object representing the most recent warning can also be accessed directly through the recorder object (see example below). If no warning has been raised, then any of the attributes that would otherwise be expected on an object representing a warning will return None.

The recorder object also has a reset() method, which clears the warnings list.

The context manager is designed to be used like this:

    with check_warnings(("assertion is always true", SyntaxWarning),
                        "+", UserWarning)):
        exec('assert(False, "Hey!")')
        warnings.warn(UserWarning("Hide me!"))

In this case if either warning was not raised, or some other warning was raised, check_warnings() would raise an error.

When a test needs to look more deeply into the warnings, rather than just checking whether or not they occurred, code like this can be used:

    with check_warnings(quiet=True) as w:
        warnings.warn("foo")
        assert str(w.args[0]) == "foo"
        warnings.warn("bar")
        assert str(w.args[0]) == "bar"
        assert str(w.warnings[0].args[0]) == "foo"
        assert str(w.warnings[1].args[0]) == "bar"
        w.reset()
        assert len(w.warnings) == 0

Here all warnings will be caught, and the test code tests the captured warnings directly. Changed in version 3.2: New optional arguments filters and quiet.

test.support.captured_stdin()
test.support.captured_stdout()
test.support.captured_stderr()

A context managers that temporarily replaces the named stream with io.StringIO object.

Example use with output streams:

    with captured_stdout() as stdout, captured_stderr() as stderr:
        print("hello")
        print("error", file=sys.stderr)
assert stdout.getvalue() == "hello\n"
assert stderr.getvalue() == "error\n"

Example use with input stream:

```python
with captured_stdout() as stdin:
    stdin.write('hello\n')
    stdin.seek(0)
    # call test code that consumes from sys.stdin
    captured = input()
self.assertEqual(captured, "hello")
```

test.support.temp_dir(path=None, quiet=False)
A context manager that creates a temporary directory at `path` and yields the directory.

If `path` is None, the temporary directory is created using `tempfile.mkdtemp()`. If `quiet` is False, the context manager raises an exception on error. Otherwise, if `path` is specified and cannot be created, only a warning is issued.

test.support.change_cwd(path, quiet=False)
A context manager that temporarily changes the current working directory to `path` and yields the directory.

If `quiet` is False, the context manager raises an exception on error. Otherwise, it issues only a warning and keeps the current working directory the same.

test.support.temp_cwd(name='tempcwd', quiet=False)
A context manager that temporarily creates a new directory and changes the current working directory (CWD).

The context manager creates a temporary directory in the current directory with name `name` before temporarily changing the current working directory. If `name` is None, the temporary directory is created using `tempfile.mkdtemp()`.

If `quiet` is False and it is not possible to create or change the CWD, an error is raised. Otherwise, only a warning is raised and the original CWD is used.

test.support.temp_umask(umask)
A context manager that temporarily sets the process umask.

test.support.can_symlink()
Return True if the OS supports symbolic links, False otherwise.

@test.support.skip_unless_symlink
A decorator for running tests that require support for symbolic links.

test.support.suppress_crash_popup()
A context manager that disables Windows Error Reporting dialogs using `SetErrorMode`. On other platforms it’s a no-op.

@test.support.anticipate_failure(condition)
A decorator to conditionally mark tests with `unittest.expectedFailure()`. Any use of this decorator should have an associated comment identifying the relevant tracker issue.

@test.support.run_with_locale(catstr, *locales)
A decorator for running a function in a different locale, correctly resetting it after it has finished. `catstr` is the locale category as a string (for example "LC_ALL"). The `locales` passed will be tried sequentially, and the first valid locale will be used.

test.support.make_bad_fd()
Create an invalid file descriptor by opening and closing a temporary file, and returning its descriptor.

test.support.import_module(name, deprecated=False)
This function imports and returns the named module. Unlike a normal import, this function raises `unittest.SkipTest` if the module cannot be imported.
Module and package deprecation messages are suppressed during this import if `deprecated` is `True`. New in version 3.1.

```python
import_fresh_module(name, fresh=(), blocked=(), deprecated=False)
```

This function imports and returns a fresh copy of the named Python module by removing the named module from `sys.modules` before doing the import. Note that unlike `reload()`, the original module is not affected by this operation.

- `fresh` is an iterable of additional module names that are also removed from the `sys.modules` cache before doing the import.
- `blocked` is an iterable of module names that are replaced with `None` in the module cache during the import to ensure that attempts to import them raise `ImportError`.

The named module and any modules named in the `fresh` and `blocked` parameters are saved before starting the import and then reinserted into `sys.modules` when the fresh import is complete.

Module and package deprecation messages are suppressed during this import if `deprecated` is `True`.

This function will raise `ImportError` if the named module cannot be imported.

Example use:

```python
# Get copies of the warnings module for testing without affecting the
# version being used by the rest of the test suite. One copy uses the
# C implementation, the other is forced to use the pure Python fallback
# implementation
py_warnings = import_fresh_module('warnings', blocked=['_warnings'])
c_warnings = import_fresh_module('warnings', fresh=['_warnings'])
```

New in version 3.1.

```python
bind_port(sock, host=HOST)
```

Bind the socket to a free port and return the port number. Relies on ephemeral ports in order to ensure we are using an unbound port. This is important as many tests may be running simultaneously, especially in a buildbot environment. This method raises an exception if the `sock.family` is `AF_INET` and `sock.type` is `SOCK_STREAM`, and the socket has `SO_REUSEADDR` or `SO_REUSEPORT` set on it. Tests should never set these socket options for TCP/IP sockets. The only case for setting these options is testing multicasting via multiple UDP sockets.

Additionally, if the `SO_EXCLUSIVEADDRUSE` socket option is available (i.e. on Windows), it will be set on the socket. This will prevent anyone else from binding to our host/port for the duration of the test.

```python
find_unused_port(family=socket.AF_INET, socktype=socket.SOCK_STREAM)
```

Returns an unused port that should be suitable for binding. This is achieved by creating a temporary socket with the same family and type as the `sock` parameter (default is `AF_INET`, `SOCK_STREAM`), and binding it to the specified host address (defaults to `0.0.0.0`) with the port set to 0, eliciting an unused ephemeral port from the OS. The temporary socket is then closed and deleted, and the ephemeral port is returned.

Either this method or `bind_port()` should be used for any tests where a server socket needs to be bound to a particular port for the duration of the test. Which one to use depends on whether the calling code is creating a python socket, or if an unused port needs to be provided in a constructor or passed to an external program (i.e. the `-accept` argument to openssl’s s_server mode). Always prefer `bind_port()` over `find_unused_port()` where possible. Using a hard coded port is discouraged since it can makes multiple instances of the test impossible to run simultaneously, which is a problem for buildbots.

The `test.support` module defines the following classes:

```python
class TransientResource
```

Instances are a context manager that raises `ResourceDenied` if the specified exception type is raised. Any keyword arguments are treated as attribute/value pairs to be compared against any exception raised within the `with` statement. Only if all pairs match properly against attributes on the exception is `ResourceDenied` raised.
class test.support.EnvironmentVarGuard

Class used to temporarily set or unset environment variables. Instances can be used as a context manager and have a complete dictionary interface for querying/modifying the underlying os.environ. After exit from the context manager all changes to environment variables done through this instance will be rolled back. Changed in version 3.1: Added dictionary interface.

EnvironmentVarGuard.set(envvar, value)
Temporarily set the environment variable envvar to the value of value.

EnvironmentVarGuard.unset(envvar)
Temporarily unset the environment variable envvar.

class test.support.WarningsRecorder

Class used to record warnings for unit tests. See documentation of check_warnings() above for more details.

26.9 venv — Creation of virtual environments

New in version 3.3. Source code: Lib/venv

The venv module provides support for creating lightweight “virtual environments” with their own site directories, optionally isolated from system site directories. Each virtual environment has its own Python binary (allowing creation of environments with various Python versions) and can have its own independent set of installed Python packages in its site directories.

See PEP 405 for more information about Python virtual environments.

26.9.1 Creating virtual environments

Creation of virtual environments is done by executing the pyvenv script:

pyvenv /path/to/new/virtual/environment

Running this command creates the target directory (creating any parent directories that don’t exist already) and places a pyvenv.cfg file in it with a home key pointing to the Python installation the command was run from. It also creates a bin (or Scripts on Windows) subdirectory containing a copy of the python binary (or binaries, in the case of Windows). It also creates an (initially empty) lib/pythonX.Y/site-packages subdirectory (on Windows, this is Lib/site-packages).

On Windows, you may have to invoke the pyvenv script as follows, if you don’t have the relevant PATH and PATHEXT settings:

c:\Temp>c:\Python33\python c:\Python33\Tools\Scripts\pyvenv.py myenv

or equivalently:

c:\Temp>c:\Python33\python -m venv myenv

The command, if run with -h, will show the available options:


Creates virtual Python environments in one or more target directories.

positional arguments:
ENV_DIR A directory to create the environment in.

optional arguments:
-h, --help show this help message and exit
--system-site-packages Give access to the global site-packages dir to the
virtual environment.

--symlinks
Try to use symlinks rather than copies, when symlinks are not the default for the platform.

--clear
Delete the environment directory if it already exists. If not specified and the directory exists, an error is raised.

--upgrade
Upgrade the environment directory to use this version of Python, assuming Python has been upgraded in-place.

If the target directory already exists an error will be raised, unless the --clear or --upgrade option was provided.

The created pyvenv.cfg file also includes the include-system-site-packages key, set to true if venv is run with the --system-site-packages option, false otherwise.

Multiple paths can be given to pyvenv, in which case an identical virtualenv will be created, according to the given options, at each provided path.

Once a venv has been created, it can be “activated” using a script in the venv’s binary directory. The invocation of the script is platform-specific: on a Posix platform, you would typically do:

$ source <venv>/bin/activate

whereas on Windows, you might do:

C:\> <venv>/Scripts/activate

if you are using the cmd.exe shell, or perhaps:

PS C:\> <venv>/Scripts/Activate.ps1

if you use PowerShell.

You don’t specifically need to activate an environment; activation just prepends the venv’s binary directory to your path, so that “python” invokes the venv’s Python interpreter and you can run installed scripts without having to use their full path. However, all scripts installed in a venv should be runnable without activating it, and run with the venv’s Python automatically.

You can deactivate a venv by typing “deactivate” in your shell. The exact mechanism is platform-specific: for example, the Bash activation script defines a “deactivate” function, whereas on Windows there are separate scripts called deactivate.bat and Deactivate.ps1 which are installed when the venv is created.

Note: A virtual environment (also called a venv) is a Python environment such that the Python interpreter, libraries and scripts installed into it are isolated from those installed in other virtual environments, and (by default) any libraries installed in a “system” Python, i.e. one which is installed as part of your operating system.

A venv is a directory tree which contains Python executable files and other files which indicate that it is a venv.

Common installation tools such as Distribute and pip work as expected with venvs - i.e. when a venv is active, they install Python packages into the venv without needing to be told to do so explicitly. Of course, you need to install them into the venv first: this could be done by running distribute_setup.py with the venv activated, followed by running easy_install pip. Alternatively, you could download the source tarballs and run python setup.py install after unpacking, with the venv activated.

When a venv is active (i.e. the venv’s Python interpreter is running), the attributes sys.prefix and sys.exec_prefix point to the base directory of the venv, whereas sys.base_prefix and sys.base_exec_prefix point to the non-venv Python installation which was used to create the venv. If a venv is not active, then sys.prefix is the same as sys.base_prefix and sys.exec_prefix is the same as sys.base_exec_prefix (they all point to a non-venv Python installation).

When a venv is active, any options that change the installation path will be ignored from all distutils configuration files to prevent projects being inadvertently installed outside of the virtual environment.

When working in a command shell, users can make a venv active by running an activate script in the venv’s executables directory (the precise filename is shell-dependent), which prepends the venv’s directory for executables to the PATH environment variable for the running shell. There should be no need in other circumstances to
activate a venv – scripts installed into venvs have a shebang line which points to the venv’s Python interpreter. This means that the script will run with that interpreter regardless of the value of PATH. On Windows, shebang line processing is supported if you have the Python Launcher for Windows installed (this was added to Python in 3.3 - see PEP 397 for more details). Thus, double-clicking an installed script in a Windows Explorer window should run the script with the correct interpreter without there needing to be any reference to its venv in PATH.

26.9.2 API

The high-level method described above makes use of a simple API which provides mechanisms for third-party virtual environment creators to customize environment creation according to their needs, the EnvBuilder class.

class venv.EnvBuilder (system_site_packages=False, clear=False, symlinks=False, upgrade=False)

The EnvBuilder class accepts the following keyword arguments on instantiation:

• system_site_packages – a Boolean value indicating that the system Python site-packages should be available to the environment (defaults to False).
• clear – a Boolean value which, if True, will delete any existing target directory instead of raising an exception (defaults to False).
• symlinks – a Boolean value indicating whether to attempt to symlink the Python binary (and any necessary DLLs or other binaries, e.g. pythonw.exe), rather than copying. Defaults to True on Linux and Unix systems, but False on Windows.
• upgrade – a Boolean value which, if True, will upgrade an existing environment with the running Python - for use when that Python has been upgraded in-place (defaults to False).

Creators of third-party virtual environment tools will be free to use the provided EnvBuilder class as a base class.

The returned env-builder is an object which has a method, create:

create (env_dir)

This method takes as required argument the path (absolute or relative to the current directory) of the target directory which is to contain the virtual environment. The create method will either create the environment in the specified directory, or raise an appropriate exception.

The create method of the EnvBuilder class illustrates the hooks available for subclass customization:

def create (self, env_dir):
    """
    Create a virtualized Python environment in a directory.
    env_dir is the target directory to create an environment in.
    """
    env_dir = os.path.abspath(env_dir)
    context = self.ensure_directories(env_dir)
    self.create_configuration(context)
    self.setup_python(context)
    self.setup_scripts(context)
    self.post_setup(context)

    Each of the methods ensure_directories(), create_configuration(), setup_python(), setup_scripts() and post_setup() can be overridden.

ensure_directories (env_dir)

Creates the environment directory and all necessary directories, and returns a context object. This is just a holder for attributes (such as paths), for use by the other methods. The directories are allowed to exist already, as long as either clear or upgrade were specified to allow operating on an existing environment directory.
create_configuration(context)
Creates the pyvenv.cfg configuration file in the environment.

setup_python(context)
Creates a copy of the Python executable (and, under Windows, DLLs) in the environment. On a POSIX system, if a specific executable python3.x was used, symlinks to python and python3 will be created pointing to that executable, unless files with those names already exist.

setup_scripts(context)
Installs activation scripts appropriate to the platform into the virtual environment.

post_setup(context)
A placeholder method which can be overridden in third party implementations to pre-install packages in the virtual environment or perform other post-creation steps.

In addition, EnvBuilder provides this utility method that can be called from setup_scripts() or post_setup() in subclasses to assist in installing custom scripts into the virtual environment.

install_scripts(context, path)
path is the path to a directory that should contain subdirectories “common”, “posix”, “nt”, each containing scripts destined for the bin directory in the environment. The contents of “common” and the directory corresponding to os.name are copied after some text replacement of placeholders:

- __VENV_DIR__ is replaced with the absolute path of the environment directory.
- __VENV_NAME__ is replaced with the environment name (final path segment of environment directory).
- __VENV_BIN_NAME__ is replaced with the name of the bin directory (either bin or Scripts).
- __VENV_PYTHON__ is replaced with the absolute path of the environment’s executable.

The directories are allowed to exist (for when an existing environment is being upgraded).

There is also a module-level convenience function:

venv.create(env_dir, system_site_packages=False, clear=False, symlinks=False)
Create an EnvBuilder with the given keyword arguments, and call its create() method with the env_dir argument.

26.9.3 An example of extending EnvBuilder

The following script shows how to extend EnvBuilder by implementing a subclass which installs setuptools and pip into a created venv:

```python
import os
import os.path
from subprocess import Popen, PIPE
import sys
from threading import Thread
from urllib.parse import urlparse
from urllib.request import urlretrieve
import venv

class ExtendedEnvBuilder(venv.EnvBuilder):
    """
    This builder installs setuptools and pip so that you can pip or easy_install other packages into the created environment.
    :param nodist: If True, setuptools and pip are not installed into the created environment.
    :param nopip: If True, pip is not installed into the created environment.
    """
```

26.9. venv — Creation of virtual environments 1183
The Python Library Reference, Release 3.3.3

:param progress: If setuptools or pip are installed, the progress of the installation can be monitored by passing a progress callable. If specified, it is called with two arguments: a string indicating some progress, and a context indicating where the string is coming from. The context argument can have one of three values: 'main', indicating that it is called from virtualize() itself, and 'stdout' and 'stderr', which are obtained by reading lines from the output streams of a subprocess which is used to install the app.

If a callable is not specified, default progress information is output to sys.stderr.

```python
def __init__(self, *args, **kwargs):
    self.nodist = kwargs.pop('nodist', False)
    self.nopip = kwargs.pop('nopip', False)
    self.progress = kwargs.pop('progress', None)
    self.verbose = kwargs.pop('verbose', False)
    super().__init__(*args, **kwargs)

def post_setup(self, context):
    ##
    Set up any packages which need to be pre-installed into the environment being created.

    :param context: The information for the environment creation request being processed.
    ##
    os.environ['VIRTUAL_ENV'] = context.env_dir
    if not self.nodist:
        self.install_setuptools(context)
    # Can’t install pip without setuptools
    if not self.nopip and not self.nodist:
        self.install_pip(context)

def reader(self, stream, context):
    ##
    Read lines from a subprocess’ output stream and either pass to a progress callable (if specified) or write progress information to sys.stderr.
    ##
    progress = self.progress
    while True:
        s = stream.readline()
        if not s:
            break
        if progress is not None:
            progress(s, context)
        else:
            if not self.verbose:
                sys.stderr.write(‘.’)
            else:
                sys.stderr.write(s.decode(‘utf-8’))
                sys.stderr.flush()
    stream.close()

def install_script(self, context, name, url):
```

Chapter 26. Development Tools
The Python Library Reference, Release 3.3.3

_, _, path, _, _, _ = urlparse(url)
fn = os.path.split(path)[-1]
binpath = context.bin_path
distpath = os.path.join(binpath, fn)
# Download script into the env’s binaries folder
urlretrieve(url, distpath)
progress = self.progress
if self.verbose:
    term = '\n'
else:
    term = ''
if progress is not None:
    progress('Installing %s ...%s' % (name, term), 'main')
else:
    sys.stderr.write('Installing %s ...%s' % (name, term))
sys.stderr.flush()
# Install in the env
args = [context.env_exe, fn]
p = Popen(args, stdout=PIPE, stderr=PIPE, cwd=binpath)
t1 = Thread(target=self.reader, args=(p.stdout, 'stdout'))
t1.start()
t2 = Thread(target=self.reader, args=(p.stderr, 'stderr'))
t2.start()
p.wait()
t1.join()
t2.join()
if progress is not None:
    progress('done.', 'main')
else:
    sys.stderr.write('done.\n')
# Clean up - no longer needed
os.unlink(distpath)

def install_setup tools(self, context):
    
    Install setuptools in the environment.

    :param context: The information for the environment creation request being processed.
    
    url = 'https://bitbucket.org/pypa/setuptools/downloads/ez_setup.py'
    self.install_script(context, 'setuptools', url)
    # clear up the setuptools archive which gets downloaded
    pred = lambda o: o.startswith('setuptools-') and o.endswith('.tar.gz')
    files = filter(pred, os.listdir(context.bin_path))
    for f in files:
        f = os.path.join(context.bin_path, f)
        os.unlink(f)

def install_pip(self, context):
    
    Install pip in the environment.

    :param context: The information for the environment creation request being processed.
    
    url = 'https://raw.githubusercontent.com/pypa/pip/master/contrib/get-pip.py'
    self.install_script(context, 'pip', url)
```

```
```python
def main():
    # ... (code snippet)
```

This script is also available for download online.
DEBUGGING AND PROFILING

These libraries help you with Python development: the debugger enables you to step through code, analyze stack frames and set breakpoints etc., and the profilers run code and give you a detailed breakdown of execution times, allowing you to identify bottlenecks in your programs.

27.1 bdb — Debugger framework

Source code: Lib/bdb.py

The bdb module handles basic debugger functions, like setting breakpoints or managing execution via the debugger.

The following exception is defined:

exception bdb.BdbQuit
    Exception raised by the Bdb class for quitting the debugger.

The bdb module also defines two classes:

class bdb.Breakpoint (self, file, line, temporary=0, cond=None, funcname=None)
    This class implements temporary breakpoints, ignore counts, disabling and (re-)enabling, and conditionals.

    Breakpoints are indexed by number through a list called bpbynumber and by (file, line) pairs through bplist. The former points to a single instance of class Breakpoint. The latter points to a list of such instances since there may be more than one breakpoint per line.

    When creating a breakpoint, its associated filename should be in canonical form. If a funcname is defined, a breakpoint hit will be counted when the first line of that function is executed. A conditional breakpoint always counts a hit.

    Breakpoint instances have the following methods:

    deleteMe()
        Delete the breakpoint from the list associated to a file/line. If it is the last breakpoint in that position, it also deletes the entry for the file/line.

    enable()
        Mark the breakpoint as enabled.

    disable()
        Mark the breakpoint as disabled.

    bpformat()
        Return a string with all the information about the breakpoint, nicely formatted:

        • The breakpoint number.
        • If it is temporary or not.
The Python Library Reference, Release 3.3.3

- Its file, line position.
- The condition that causes a break.
- If it must be ignored the next N times.
- The breakpoint hit count.

New in version 3.2.

**bpprint** *(out=None)*

Print the output of **bpformat()** to the file **out**, or if it is **None**, to standard output.

**class** **bdb.Bdb** *(skip=None)*

The **Bdb** class acts as a generic Python debugger base class.

This class takes care of the details of the trace facility; a derived class should implement user interaction. The standard debugger class (**pdb.Pdb**) is an example.

The **skip** argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns. Whether a frame is considered to originate in a certain module is determined by the **__name__** in the frame globals. New in version 3.1: The **skip** argument. The following methods of **Bdb** normally don’t need to be overridden.

**canonic** *(filename)*

Auxiliary method for getting a filename in a canonical form, that is, as a case-normalized (on case-insensitive filesystems) absolute path, stripped of surrounding angle brackets.

**reset()**

Set the **botframe**, **stopframe**, **returnframe** and **quitting** attributes with values ready to start debugging.

**trace_dispatch** *(frame, event, arg)*

This function is installed as the trace function of debugged frames. Its return value is the new trace function (in most cases, that is, itself).

The default implementation decides how to dispatch a frame, depending on the type of event (passed as a string) that is about to be executed. **event** can be one of the following:

- "line": A new line of code is going to be executed.
- "call": A function is about to be called, or another code block entered.
- "return": A function or other code block is about to return.
- "exception": An exception has occurred.
- "c_call": A C function is about to be called.
- "c_return": A C function has returned.
- "c_exception": A C function has raised an exception.

For the Python events, specialized functions (see below) are called. For the C events, no action is taken.

The **arg** parameter depends on the previous event.

See the documentation for **sys.settrace()** for more information on the trace function. For more information on code and frame objects, refer to **types**.

**dispatch_line** *(frame)*

If the debugger should stop on the current line, invoke the **user_line()** method (which should be overridden in subclasses). Raise a **BdbQuit** exception if the **Bdb.quitting** flag is set (which can be set from **user_line()**). Return a reference to the **trace_dispatch()** method for further tracing in that scope.

**dispatch_call** *(frame, arg)*

If the debugger should stop on this function call, invoke the **user_call()** method (which should be overridden in subclasses). Raise a **BdbQuit** exception if the **Bdb.quitting** flag is set (which can
be set from \texttt{user\_call()}. Return a reference to the \texttt{trace\_dispatch()} method for further tracing in that scope.

\textbf{dispatch\_return (frame, arg)}

If the debugger should stop on this function return, invoke the \texttt{user\_return()} method (which should be overridden in subclasses). Raise a \texttt{BdbQuit} exception if the Bdb.quitting flag is set (which can be set from \texttt{user\_return()}). Return a reference to the \texttt{trace\_dispatch()} method for further tracing in that scope.

\textbf{dispatch\_exception (frame, arg)}

If the debugger should stop at this exception, invokes the \texttt{user\_exception()} method (which should be overridden in subclasses). Raise a \texttt{BdbQuit} exception if the Bdb.quitting flag is set (which can be set from \texttt{user\_exception()}). Return a reference to the \texttt{trace\_dispatch()} method for further tracing in that scope.

Normally derived classes don’t override the following methods, but they may if they want to redefine the definition of stopping and breakpoints.

\textbf{stop\_here (frame)}

This method checks if the \texttt{frame} is somewhere below \texttt{botframe} in the call stack. \texttt{botframe} is the frame in which debugging started.

\textbf{break\_here (frame)}

This method checks if there is a breakpoint in the filename and line belonging to \texttt{frame} or, at least, in the current function. If the breakpoint is a temporary one, this method deletes it.

\textbf{break\_anywhere (frame)}

This method checks if there is a breakpoint in the filename of the current frame.

Derived classes should override these methods to gain control over debugger operation.

\textbf{user\_call (frame, argument\_list)}

This method is called from \texttt{dispatch\_call()} when there is the possibility that a break might be necessary anywhere inside the called function.

\textbf{user\_line (frame)}

This method is called from \texttt{dispatch\_line()} when either \texttt{stop\_here()} or \texttt{break\_here()} yields True.

\textbf{user\_return (frame, return\_value)}

This method is called from \texttt{dispatch\_return()} when \texttt{stop\_here()} yields True.

\textbf{user\_exception (frame, exc\_info)}

This method is called from \texttt{dispatch\_exception()} when \texttt{stop\_here()} yields True.

\textbf{do\_clear (arg)}

Handle how a breakpoint must be removed when it is a temporary one.

This method must be implemented by derived classes.

Derived classes and clients can call the following methods to affect the stepping state.

\textbf{set\_step ()}

Stop after one line of code.

\textbf{set\_next (frame)}

Stop on the next line in or below the given frame.

\textbf{set\_return (frame)}

Stop when returning from the given frame.

\textbf{set\_until (frame)}

Stop when the line with the line no greater than the current one is reached or when returning from current frame

\textbf{set\_trace ([frame])}

Start debugging from \texttt{frame}. If \texttt{frame} is not specified, debugging starts from caller’s frame.
**set_continue()**
Stop only at breakpoints or when finished. If there are no breakpoints, set the system trace function to None.

**set_quit()**
Set the quitting attribute to True. This raises BdbQuit in the next call to one of the dispatch_*( ) methods.

Derived classes and clients can call the following methods to manipulate breakpoints. These methods return a string containing an error message if something went wrong, or None if all is well.

**set_break (filename, lineno, temporary=0, cond, funcname)**
Set a new breakpoint. If the lineno line doesn’t exist for the filename passed as argument, return an error message. The filename should be in canonical form, as described in the canon() method.

**clear_break (filename, lineno)**
Delete the breakpoints in filename and lineno. If none were set, an error message is returned.

**clear_bpbynumber (arg)**
Delete the breakpoint which has the index arg in the Breakpoint.bpbynumber. If arg is not numeric or out of range, return an error message.

**clear_all_file_breaks (filename)**
Delete all breakpoints in filename. If none were set, an error message is returned.

**clear_all_breaks ()**
Delete all existing breakpoints.

**get_bpbynumber (arg)**
Return a breakpoint specified by the given number. If arg is a string, it will be converted to a number. If arg is a non-numeric string, if the given breakpoint never existed or has been deleted, a ValueError is raised. New in version 3.2.

**get_break (filename, lineno)**
Check if there is a breakpoint for lineno of filename.

**get_breaks (filename, lineno)**
Return all breakpoints for lineno in filename, or an empty list if none are set.

**get_file_breaks (filename)**
Return all breakpoints in filename, or an empty list if none are set.

**get_all_breaks ()**
Return all breakpoints that are set.

Derived classes and clients can call the following methods to get a data structure representing a stack trace.

**get_stack (f, t)**
Get a list of records for a frame and all higher (calling) and lower frames, and the size of the higher part.

**format_stack_entry (frame_lineno, lprefix=':')**
Return a string with information about a stack entry, identified by a (frame, lineno) tuple:
- The canonical form of the filename which contains the frame.
- The function name, or "<lambda>".
- The input arguments.
- The return value.
- The line of code (if it exists).

The following two methods can be called by clients to use a debugger to debug a statement, given as a string.
run \((cmd, \text{globals}=\text{None}, \text{locals}=\text{None})\)
Debug a statement executed via the `exec()` function. `globals` defaults to `__main__.__dict__`, `locals` defaults to `globals`.

runeval \((expr, \text{globals}=\text{None}, \text{locals}=\text{None})\)
Debug an expression executed via the `eval()` function. `globals` and `locals` have the same meaning as in `run()`.

runcall \((\text{func}, *\text{args}, **\text{kwds})\)
Debug a single function call, and return its result.

Finally, the module defines the following functions:

\texttt{bdb.\_checkfuncname}\((b, \text{frame})\)
Check whether we should break here, depending on the way the breakpoint \(b\) was set.

\texttt{bdb.\_effective}\((\text{file}, \text{line}, \text{frame})\)
Determine if there is an effective (active) breakpoint at this line of code. Return a tuple of the breakpoint and a boolean that indicates if it is ok to delete a temporary breakpoint. Return \((\text{None}, \text{None})\) if there is no matching breakpoint.

\texttt{bdb.\_set\_trace()}\(
\)
Start debugging with a \texttt{Bdb} instance from caller’s frame.

### 27.2 faulthandler — Dump the Python traceback

This module contains functions to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal. Call \texttt{faulthandler.enable()} to install fault handlers for the \texttt{SIGSEGV}, \texttt{SIGFPE}, \texttt{SIGABRT}, \texttt{SIGBUS}, and \texttt{SIGILL} signals. You can also enable them at startup by setting the \texttt{PYTHONFAULTHANDLER} environment variable or by using \texttt{-X faulthandler} command line option.

The fault handler is compatible with system fault handlers like Apport or the Windows fault handler. The module uses an alternative stack for signal handlers if the \texttt{sigaltstack()} function is available. This allows it to dump the traceback even on a stack overflow.

The fault handler is called on catastrophic cases and therefore can only use signal-safe functions (e.g. it cannot allocate memory on the heap). Because of this limitation traceback dumping is minimal compared to normal Python tracebacks:

- Only ASCII is supported. The \texttt{backslashreplace} error handler is used on encoding.
- Each string is limited to 500 characters.
- Only the filename, the function name and the line number are displayed. (no source code)
- It is limited to 100 frames and 100 threads.

By default, the Python traceback is written to \texttt{sys.stderr}. To see tracebacks, applications must be run in the terminal. A log file can alternatively be passed to \texttt{faulthandler.enable()}.

The module is implemented in C, so tracebacks can be dumped on a crash or when Python is deadlocked. New in version 3.3.

#### 27.2.1 Dump the traceback

\texttt{faulthandler.dump\_traceback}\((\text{file}=\text{sys.stderr}, \text{all\_threads}=\text{True})\)
Dump the tracebacks of all threads into file. If \texttt{all\_threads} is \texttt{False}, dump only the current thread.
27.2.2 Fault handler state

faulthandler.enable (file=sys.stderr, all_threads=True)
   Enable the fault handler: install handlers for the SIGSEGV, SIGFPE, SIGABRT, SIGBUS and SIGILL signals to dump the Python traceback. If all_threads is True, produce tracebacks for every running thread. Otherwise, dump only the current thread.

faulthandler.disable()
   Disable the fault handler: uninstall the signal handlers installed by enable().

faulthandler.is_enabled()
   Check if the fault handler is enabled.

27.2.3 Dump the tracebacks after a timeout

faulthandler.dump_traceback_later (timeout, repeat=False, file=sys.stderr, exit=False)
   Dump the tracebacks of all threads, after a timeout of timeout seconds, or every timeout seconds if repeat is True. If exit is True, call _exit() with status=1 after dumping the tracebacks. (Note _exit() exits the process immediately, which means it doesn’t do any cleanup like flushing file buffers.) If the function is called twice, the new call replaces previous parameters and resets the timeout. The timer has a sub-second resolution.

   This function is implemented using a watchdog thread and therefore is not available if Python is compiled with threads disabled.

faulthandler.cancel_dump_traceback_later()
   Cancel the last call to dump_traceback_later().

27.2.4 Dump the traceback on a user signal

faulthandler.register (signum, file=sys.stderr, all_threads=True, chain=False)
   Register a user signal: install a handler for the signum signal to dump the traceback of all threads, or of the current thread if all_threads is False, into file. Call the previous handler if chain is True.

   Not available on Windows.

faulthandler.unregister (signum)
   Unregister a user signal: uninstall the handler of the signum signal installed by register(). Return True if the signal was registered, False otherwise.

   Not available on Windows.

27.2.5 File descriptor issue

enable(), dump_traceback_later() and register() keep the file descriptor of their file argument. If the file is closed and its file descriptor is reused by a new file, or if os.dup2() is used to replace the file descriptor, the traceback will be written into a different file. Call these functions again each time that the file is replaced.

27.2.6 Example

Example of a segmentation fault on Linux:

$ python -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault

Current thread 0x00007fb899f39700:
27.3 pdb — The Python Debugger

The module `pdb` defines an interactive source code debugger for Python programs. It supports setting (conditional) breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame. It also supports post-mortem debugging and can be called under program control.

The debugger is extensible – it is actually defined as the class `Pdb`. This is currently undocumented but easily understood by reading the source. The extension interface uses the modules `bdb` and `cmd`.

The debugger’s prompt is `(Pdb)`. Typical usage to run a program under control of the debugger is:

```python
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

Changed in version 3.3: Tab-completion via the `readline` module is available for commands and command arguments, e.g. the current global and local names are offered as arguments of the `print` command. `pdb.py` can also be invoked as a script to debug other scripts. For example:

```
python3 -m pdb myscript.py
```

When invoked as a script, `pdb` will automatically enter post-mortem debugging if the program being debugged exits abnormally. After post-mortem debugging (or after normal exit of the program), `pdb` will restart the program. Automatic restarting preserves `pdb`’s state (such as breakpoints) and in most cases is more useful than quitting the debugger upon program’s exit. New in version 3.2: `pdb.py` now accepts a `-c` option that executes commands as if given in a .pdbrc file, see Debugger Commands. The typical usage to break into the debugger from a running program is to insert

```
import pdb; pdb.set_trace()
```

at the location you want to break into the debugger. You can then step through the code following this statement, and continue running without the debugger using the `continue` command.

The typical usage to inspect a crashed program is:

```python
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "/mymodule.py", line 4, in test
    test2()
  File "/mymodule.py", line 3, in test2
    print(spam)
NameError: spam
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print(spam)
(Pdb)
```
The module defines the following functions; each enters the debugger in a slightly different way:

```python
definitions:
    run (statement, globals=None, locals=None)
    runeval (expression, globals=None, locals=None)
    runcall (function, *args, **kwds)
    set_trace ()
    post_mortem (traceback=None)
```

- `run` executes the `statement` (given as a string or a code object) under debugger control. The debugger prompt appears before any code is executed; you can set breakpoints and type `continue`, or you can step through the statement using `step` or `next` (all these commands are explained below). The optional `globals` and `locals` arguments specify the environment in which the code is executed; by default the dictionary of the module `__main__` is used. (See the explanation of the built-in `exec()` or `eval()` functions.)

- `runeval` evaluates the `expression` (given as a string or a code object) under debugger control. When `runeval()` returns, it returns the value of the expression. Otherwise this function is similar to `run()`.

- `runcall` calls the `function` (a function or method object, not a string) with the given arguments. When `runcall()` returns, it returns whatever the function call returned. The debugger prompt appears as soon as the function is entered.

- `set_trace` enters the debugger at the calling stack frame. This is useful to hard-code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails).

- `post_mortem` enters post-mortem debugging of the given `traceback` object. If no `traceback` is given, it uses the one of the exception that is currently being handled (an exception must be being handled if the default is to be used).

- `pm` enters post-mortem debugging of the traceback found in `sys.last_traceback`.

The `run*` functions and `set_trace()` are aliases for instantiating the `Pdb` class and calling the method of the same name. If you want to access further features, you have to do this yourself:

```python
class Pdb (completekey='tab', stdin=None, stdout=None, skip=None, nosigint=False)
```

`Pdb` is the debugger class.

- `completekey`, `stdin` and `stdout` arguments are passed to the underlying `cmd.Cmd` class; see the description there.

- The `skip` argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns. \(^1\)

By default, Pdb sets a handler for the SIGINT signal (which is sent when the user presses Ctrl-C on the console) when you give a `continue` command. This allows you to break into the debugger again by pressing Ctrl-C. If you want Pdb not to touch the SIGINT handler, set `nosigint` to true.

Example call to enable tracing with `skip`:

```python
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

New in version 3.1: The `skip` argument. New in version 3.2: The `nosigint` argument. Previously, a SIGINT handler was never set by Pdb.

```python
run (statement, globals=None, locals=None)
runeval (expression, globals=None, locals=None)
runcall (function, *args, **kwds)
set_trace ()
```

See the documentation for the functions explained above.

### 27.3.1 Debugger Commands

The commands recognized by the debugger are listed below. Most commands can be abbreviated to one or two letters as indicated; e.g. `h (elp)` means that either `h` or `help` can be used to enter the help command (but not

---

\(^1\) Whether a frame is considered to originate in a certain module is determined by the `__name__` in the frame globals.
he or hel, nor H or Help or HELP). Arguments to commands must be separated by whitespace (spaces or tabs). Optional arguments are enclosed in square brackets ([ ]) in the command syntax; the square brackets must not be typed. Alternatives in the command syntax are separated by a vertical bar (|).

Entering a blank line repeats the last command entered. Exception: if the last command was a list command, the next 11 lines are listed.

Commands that the debugger doesn’t recognize are assumed to be Python statements and are executed in the context of the program being debugged. Python statements can also be prefixed with an exclamation point (!). This is a powerful way to inspect the program being debugged; it is even possible to change a variable or call a function. When an exception occurs in such a statement, the exception name is printed but the debugger’s state is not changed.

The debugger supports aliases. Aliases can have parameters which allows one a certain level of adaptability to the context under examination.

Multiple commands may be entered on a single line, separated by ;;. (A single ; is not used as it is the separator for multiple commands in a line that is passed to the Python parser.) No intelligence is applied to separating the commands; the input is split at the first ;; pair, even if it is in the middle of a quoted string.

If a file .pdbrc exists in the user’s home directory or in the current directory, it is read in and executed as if it had been typed at the debugger prompt. This is particularly useful for aliases. If both files exist, the one in the home directory is read first and aliases defined there can be overridden by the local file. Changed in version 3.2: .pdbrc can now contain commands that continue debugging, such as continue or next. Previously, these commands had no effect.

h(help) [command]
Without argument, print the list of available commands. With a command as argument, print help about that command. help pdb displays the full documentation (the docstring of the pdb module). Since the command argument must be an identifier, help exec must be entered to get help on the ! command.

w(here)
Print a stack trace, with the most recent frame at the bottom. An arrow indicates the current frame, which determines the context of most commands.

d(own) [count]
Move the current frame count (default one) levels down in the stack trace (to a newer frame).

u(p) [count]
Move the current frame count (default one) levels up in the stack trace (to an older frame).

b(reak) [([filename:]lineno | function) [, condition]]
With a lineno argument, set a break there in the current file. With a function argument, set a break at the first executable statement within that function. The line number may be prefixed with a filename and a colon, to specify a breakpoint in another file (probably one that hasn’t been loaded yet). The file is searched on sys.path. Note that each breakpoint is assigned a number to which all the other breakpoint commands refer.

If a second argument is present, it is an expression which must evaluate to true before the breakpoint is honored.

Without argument, list all breaks, including for each breakpoint, the number of times that breakpoint has been hit, the current ignore count, and the associated condition if any.

tbreak [([filename:]lineno | function) [, condition]]
Temporary breakpoint, which is removed automatically when it is first hit. The arguments are the same as for break.

clear [filename:lineno | bpnumber [bpnumber ...]]
With a filename:lineno argument, clear all the breakpoints at this line. With a space separated list of breakpoint numbers, clear those breakpoints. Without argument, clear all breaks (but first ask confirmation).

disable [bpnumber [bpnumber ...]]
Disable the breakpoints given as a space separated list of breakpoint numbers. Disabling a breakpoint means it cannot cause the program to stop execution, but unlike clearing a breakpoint, it remains in the list of breakpoints and can be (re-)enabled.
enable [bpnumber [bpnumber ...]]
   Enable the breakpoints specified.

ignore bpnumber [count]
   Set the ignore count for the given breakpoint number. If count is omitted, the ignore count is set to 0. A breakpoint becomes active when the ignore count is zero. When non-zero, the count is decremented each time the breakpoint is reached and the breakpoint is not disabled and any associated condition evaluates to true.

condition bpnumber [condition]
   Set a new condition for the breakpoint, an expression which must evaluate to true before the breakpoint is honored. If condition is absent, any existing condition is removed; i.e., the breakpoint is made unconditional.

commands [bpnumber]
   Specify a list of commands for breakpoint number bpnumber. The commands themselves appear on the following lines. Type a line containing just end to terminate the commands. An example:

   (Pdb) commands 1
   (com) print some_variable
   (com) end
   (Pdb)

   To remove all commands from a breakpoint, type commands and follow it immediately with end; that is, give no commands.

   With no bpnumber argument, commands refers to the last breakpoint set.

   You can use breakpoint commands to start your program up again. Simply use the continue command, or step, or any other command that resumes execution.

   Specifying any command resuming execution (currently continue, step, next, return, jump, quit and their abbreviations) terminates the command list (as if that command was immediately followed by end). This is because any time you resume execution (even with a simple next or step), you may encounter another breakpoint–which could have its own command list, leading to ambiguities about which list to execute.

   If you use the 'silent' command in the command list, the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue. If none of the other commands print anything, you see no sign that the breakpoint was reached.

s(tep)
   Execute the current line, stop at the first possible occasion (either in a function that is called or on the next line in the current function).

n(ext)
   Continue execution until the next line in the current function is reached or it returns. (The difference between next and step is that step stops inside a called function, while next executes called functions at (nearly) full speed, only stopping at the next line in the current function.)

unt(ill) [lineno]
   Without argument, continue execution until the line with a number greater than the current one is reached.

   With a line number, continue execution until a line with a number greater or equal to that is reached. In both cases, also stop when the current frame returns. Changed in version 3.2: Allow giving an explicit line number.

r(eturn)
   Continue execution until the current function returns.

c(ont(inue))
   Continue execution, only stop when a breakpoint is encountered.

j(ump) lineno
   Set the next line that will be executed. Only available in the bottom-most frame. This lets you jump back and execute code again, or jump forward to skip code that you don’t want to run.
It should be noted that not all jumps are allowed – for instance it is not possible to jump into the middle of a for loop or out of a finally clause.

**l(is)** `[first[, last]]`
List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing. With . as argument, list 11 lines around the current line. With one argument, list 11 lines around that line. With two arguments, list the given range; if the second argument is less than the first, it is interpreted as a count.

The current line in the current frame is indicated by ->. If an exception is being debugged, the line where the exception was originally raised or propagated is indicated by >>, if it differs from the current line. New in version 3.2: The >> marker.

**ll | longlist**
List all source code for the current function or frame. Interesting lines are marked as for list. New in version 3.2.

**a(rgs)**
Print the argument list of the current function.

**p(rint) expression**
Evaluate the expression in the current context and print its value.

**PP expression**
Like the print command, except the value of the expression is pretty-printed using the pprint module.

**whatis expression**
Print the type of the expression.

**source expression**
Try to get source code for the given object and display it. New in version 3.2.

**display [expression]**
Display the value of the expression if it changed, each time execution stops in the current frame.
Without expression, list all display expressions for the current frame. New in version 3.2.

**undisplay [expression]**
Do not display the expression any more in the current frame. Without expression, clear all display expressions for the current frame. New in version 3.2.

**interact**
Start an interactive interpreter (using the code module) whose global namespace contains all the (global and local) names found in the current scope. New in version 3.2.

**alias [name [command]]**
Create an alias called name that executes command. The command must not be enclosed in quotes. Replaceable parameters can be indicated by %1, %2, and so on, while %* is replaced by all the parameters. If no command is given, the current alias for name is shown. If no arguments are given, all aliases are listed.

Aliases may be nested and can contain anything that can be legally typed at the pdb prompt. Note that internal pdb commands can be overridden by aliases. Such a command is then hidden until the alias is removed. Aliasing is recursively applied to the first word of the command line; all other words in the line are left alone.

As an example, here are two useful aliases (especially when placed in the .pdbrc file):

```python
# Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print("%1."+k,"=",%1.__dict__[k])
# Print instance variables in self
alias ps pi self
```

**unalias name**
Delete the specified alias.
! statement
   Execute the (one-line) statement in the context of the current stack frame. The exclamation point can be
   omitted unless the first word of the statement resembles a debugger command. To set a global variable, you
   can prefix the assignment command with a global statement on the same line, e.g.:

   (Pdb) global list_options; list_options = ['-l']
   (Pdb)

run [args ...]
restart [args ...]
   Restart the debugged Python program. If an argument is supplied, it is split with shlex and the result is
   used as the new sys.argv. History, breakpoints, actions and debugger options are preserved. restart
   is an alias for run.

q(uit)
   Quit from the debugger. The program being executed is aborted.

27.4 The Python Profilers

Source code: Lib/profile.py and Lib/pstats.py

27.4.1 Introduction to the profilers

cProfile and profile provide deterministic profiling of Python programs. A profile is a set of statistics that
describes how often and for how long various parts of the program executed. These statistics can be formatted
into reports via the pstats module.

The Python standard library provides two different implementations of the same profiling interface:

1. cProfile is recommended for most users; it’s a C extension with reasonable overhead that makes it
   suitable for profiling long-running programs. Based on lprof, contributed by Brett Rosen and Ted
   Czotter.

2. profile, a pure Python module whose interface is imitated by cProfile, but which adds significant
   overhead to profiled programs. If you’re trying to extend the profiler in some way, the task might be easier
   with this module.

Note: The profiler modules are designed to provide an execution profile for a given program, not for benchmark-
ning purposes (for that, there is timeit for reasonably accurate results). This particularly applies to benchmarking
Python code against C code: the profilers introduce overhead for Python code, but not for C-level functions, and
so the C code would seem faster than any Python one.

27.4.2 Instant User’s Manual

This section is provided for users that “don’t want to read the manual.” It provides a very brief overview, and
allows a user to rapidly perform profiling on an existing application.

To profile a function that takes a single argument, you can do:

    import cProfile
    import re
    cProfile.run('re.compile("foo|bar")')

(Use profile instead of cProfile if the latter is not available on your system.)

The above action would run re.compile() and print profile results like the following:
197 function calls (192 primitive calls) in 0.002 seconds

Ordered by: standard name

<table>
<thead>
<tr>
<th>ncalls</th>
<th>tottime</th>
<th>percall</th>
<th>cumtime</th>
<th>percall</th>
<th>filename:lineno(function)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.001</td>
<td>0.001</td>
<td>&lt;string&gt;:1(&lt;module&gt;)</td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.001</td>
<td>0.001</td>
<td>re.py:212(compile)</td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.001</td>
<td>0.001</td>
<td>re.py:268(_compile)</td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>sre_compile.py:172(_compile_charset)</td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>sre_compile.py:201(_optimize_charset)</td>
</tr>
<tr>
<td>4</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>sre_compile.py:25(_identityfunction)</td>
</tr>
<tr>
<td>3/1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>sre_compile.py:33(_compile)</td>
</tr>
</tbody>
</table>

The first line indicates that 197 calls were monitored. Of those calls, 192 were primitive, meaning that the call was not induced via recursion. The next line: Ordered by: standard name, indicates that the text string in the far right column was used to sort the output. The column headings include:

- ncalls for the number of calls,
- tottime for the total time spent in the given function (and excluding time made in calls to sub-functions)
- percall is the quotient of tottime divided by ncalls
- cumtime is the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate even for recursive functions.
- percall is the quotient of cumtime divided by primitive calls
- filename:lineno(function) provides the respective data of each function

When there are two numbers in the first column (for example 3/1), it means that the function recursed. The second value is the number of primitive calls and the former is the total number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

Instead of printing the output at the end of the profile run, you can save the results to a file by specifying a filename to the run() function:

```python
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

The pstats.Stats class reads profile results from a file and formats them in various ways.

The file cProfile can also be invoked as a script to profile another script. For example:

```bash
python -m cProfile [-o output_file] [-s sort_order] myscript.py
```

- o writes the profile results to a file instead of to stdout
- s specifies one of the sort_stats() sort values to sort the output by. This only applies when -o is not supplied.

The pstats module’s Stats class has a variety of methods for manipulating and printing the data saved into a profile results file:

```python
import pstats
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

The strip_dirs() method removed the extraneous path from all the module names. The sort_stats() method sorted all the entries according to the standard module/line/name string that is printed. The print_stats() method printed out all the statistics. You might try the following sort calls:

```python
p.sort_stats('name')
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:
p.sort_stats('cumulative').print_stats(10)

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

p.sort_stats('time').print_stats(10)

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try:

p.sort_stats('file').print_stats('__init__')

This will sort all the statistics by file name, and then print out statistics for only the class init methods (since they are spelled with __init__ in them). As one final example, you could try:

p.sort_stats('time', 'cum').print_stats(.5, 'init')

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: .5) of its original size, then only lines containing init are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (p is still sorted according to the last criteria) do:

p.print_callers(.5, 'init')

and you would get a list of callers for each of the listed functions.

If you want more functionality, you’re going to have to read the manual, or guess what the following functions do:

p.print_callees()
p.add('restats')

Invoked as a script, the pstats module is a statistics browser for reading and examining profile dumps. It has a simple line-oriented interface (implemented using cmd) and interactive help.

27.4.3 profile and cProfile Module Reference

Both the profile and cProfile modules provide the following functions:

profile.run(command, filename=None, sort=-1)

This function takes a single argument that can be passed to the exec() function, and an optional file name. In all cases this routine executes:

exec(command, __main__.__dict__, __main__.__dict__)

and gathers profiling statistics from the execution. If no file name is present, then this function automatically creates a Stats instance and prints a simple profiling report. If the sort value is specified it is passed to this Stats instance to control how the results are sorted.

profile.runctx(command, globals, locals, filename=None)

This function is similar to run(), with added arguments to supply the globals and locals dictionaries for the command string. This routine executes:

exec(command, globals, locals)

and gathers profiling statistics as in the run() function above.

class profile.Profile(timer=None, timeunit=0.0, subcalls=True, builtins=True)

This class is normally only used if more precise control over profiling is needed than what the cProfile.run() function provides.

A custom timer can be supplied for measuring how long code takes to run via the timer argument. This must be a function that returns a single number representing the current time. If the number is an integer,
the `timeunit` specifies a multiplier that specifies the duration of each unit of time. For example, if the timer
returns times measured in thousands of seconds, the time unit would be `.001`.

Directly using the `Profile` class allows formatting profile results without writing the profile data to a file:

```python
import cProfile, pstats, io
pr = cProfile.Profile()
pr.enable()
    # ... do something ...
pr.disable()
s = io.StringIO()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print(s.getvalue())
```

`enable()`  
Start collecting profiling data.

`disable()`  
Stop collecting profiling data.

`create_stats()`  
Stop collecting profiling data and record the results internally as the current profile.

`print_stats(sort=-1)`  
Create a `Stats` object based on the current profile and print the results to stdout.

`dump_stats(filename)`  
Write the results of the current profile to `filename`.

`run(cmd)`  
Profile the cmd via `exec()`.

`runcall(func, *args, **kwargs)`  
Profile `func(*args, **kwargs)`

### 27.4.4 The Stats Class

Analysis of the profiler data is done using the `Stats` class.

```python
class pstats.Stats(*filenames or profile, stream=sys.stdout)
```

This class constructor creates an instance of a “statistics object” from a `filename` (or list of filenames) or from a `Profile` instance. Output will be printed to the stream specified by `stream`.

The file selected by the above constructor must have been created by the corresponding version of `profile` or `cProfile`. To be specific, there is no file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers. If several files are provided, all the statistics for identical functions will be coalesced, so that an overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing `Stats` object, the `add()` method can be used.

Instead of reading the profile data from a file, a `cProfile.Profile` or `profile.Profile` object can be used as the profile data source.

`Stats` objects have the following methods:

`strip_dirs()`  
This method for the `Stats` class removes all leading path information from file names. It is very useful in reducing the size of the printout to fit within (close to) 80 columns. This method modifies
the object, and the stripped information is lost. After performing a strip operation, the object is considered to have its entries in a “random” order, as it was just after object initialization and loading. If strip_dirs() causes two function names to be indistinguishable (they are on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

add(*filenames)
This method of the Stats class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of profile.run() or cProfile.run(). Statistics for identically named (re: file, line, name) functions are automatically accumulated into single function statistics.

dump_stats(filename)
Save the data loaded into the Stats object to a file named filename. The file is created if it does not exist, and is overwritten if it already exists. This is equivalent to the method of the same name on the profile.Profile and cProfile.Profile classes.

sort_stats(*keys)
This method modifies the Stats object by sorting it according to the supplied criteria. The argument is typically a string identifying the basis of a sort (example: ‘time’ or ‘name’).

When more than one key is provided, then additional keys are used as secondary criteria when there is equality in all keys selected before them. For example, sort_stats(‘name’, ‘file’) will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.

Abbreviations can be used for any key names, as long as the abbreviation is unambiguous. The following are the keys currently defined:

<table>
<thead>
<tr>
<th>Valid Arg</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘calls’</td>
<td>call count</td>
</tr>
<tr>
<td>‘cumulative’</td>
<td>cumulative time</td>
</tr>
<tr>
<td>‘cumtime’</td>
<td>cumulative time</td>
</tr>
<tr>
<td>‘file’</td>
<td>file name</td>
</tr>
<tr>
<td>‘filename’</td>
<td>file name</td>
</tr>
<tr>
<td>‘module’</td>
<td>file name</td>
</tr>
<tr>
<td>‘ncalls’</td>
<td>call count</td>
</tr>
<tr>
<td>‘pcalls’</td>
<td>primitive call count</td>
</tr>
<tr>
<td>‘line’</td>
<td>line number</td>
</tr>
<tr>
<td>‘name’</td>
<td>function name</td>
</tr>
<tr>
<td>‘nfl’</td>
<td>name/file linea</td>
</tr>
<tr>
<td>‘stdname’</td>
<td>standard name</td>
</tr>
<tr>
<td>‘time’</td>
<td>internal time</td>
</tr>
<tr>
<td>‘tottime’</td>
<td>internal time</td>
</tr>
</tbody>
</table>

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (alphabetical). The subtle distinction between ‘nfl’ and ‘stdname’ is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, ‘nfl’ does a numeric compare of the line numbers. In fact, sort_stats(‘nfl’) is the same as sort_stats(‘name’, ‘file’, ‘line’).

For backward-compatibility reasons, the numeric arguments -1, 0, 1, and 2 are permitted. They are interpreted as ‘stdname’, ‘calls’, ‘time’, and ‘cumulative’ respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

reverse_order()
This method for the Stats class reverses the ordering of the basic list within the object. Note that by default ascending vs descending order is properly selected based on the sort key of choice.
The Python Library Reference, Release 3.3.3

print_stats(*restrictions)

This method for the Stats class prints out a report as described in the profile.run() definition. The order of the printing is based on the last sort_stats() operation done on the object (subject to caveats in add() and strip_dirs()). The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a regular expression (to pattern match the standard name that is printed. If several restrictions are provided, then they are applied sequentially. For example:

print_stats(.1, 'foo:')

would first limit the printing to first 10% of list, and then only print functions that were part of filename .*foo:. In contrast, the command:

print_stats('foo:', .1)

would limit the list to all functions having file names .*foo:, and then proceed to only print the first 10% of them.

print_callers(*restrictions)

This method for the Stats class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by print_stats(), and the definition of the restricting argument is also identical. Each caller is reported on its own line. The format differs slightly depending on the profiler that produced the stats:

• With profile, a number is shown in parentheses after each caller to show how many times this specific call was made. For convenience, a second non-parenthesized number repeats the cumulative time spent in the function at the right.

• With cProfile, each caller is preceded by three numbers: the number of times this specific call was made, and the total and cumulative times spent in the current function while it was invoked by this specific caller.

print_callees(*restrictions)

This method for the Stats class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re: called vs was called by), the arguments and ordering are identical to the print_callers() method.

27.4.5 What Is Deterministic Profiling?

Deterministic profiling is meant to reflect the fact that all function call, function return, and exception events are monitored, and precise timings are made for the intervals between these events (during which time the user’s code is executing). In contrast, statistical profiling (which is not done by this module) randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required to do deterministic profiling. Python automatically provides a hook (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead in typical applications. The result is that deterministic profiling is not that expensive, yet provides extensive run time statistics about the execution of a Python program.

Call count statistics can be used to identify bugs in code (surprising counts), and to identify possible inline-expansion points (high call counts). Internal time statistics can be used to identify “hot loops” that should be carefully optimized. Cumulative time statistics should be used to identify high level errors in the selection of algorithms. Note that the unusual handling of cumulative times in this profiler allows statistics for recursive implementations of algorithms to be directly compared to iterative implementations.
27.4.6 Limitations

One limitation has to do with accuracy of timing information. There is a fundamental problem with deterministic profilers involving accuracy. The most obvious restriction is that the underlying “clock” is only ticking at a rate (typically) of about .001 seconds. Hence no measurements will be more accurate than the underlying clock. If enough measurements are taken, then the “error” will tend to average out. Unfortunately, removing this first error induces a second source of error.

The second problem is that it “takes a while” from when an event is dispatched until the profiler’s call to get the time actually gets the state of the clock. Similarly, there is a certain lag when exiting the profiler event handler from the time that the clock’s value was obtained (and then squirreled away), until the user’s code is once again executing. As a result, functions that are called many times, or call many functions, will typically accumulate this error. The error that accumulates in this fashion is typically less than the accuracy of the clock (less than one clock tick), but it can accumulate and become very significant.

The problem is more important with profile than with the lower-overhead cProfile. For this reason, profile provides a means of calibrating itself for a given platform so that this error can be probabilistically (on the average) removed. After the profiler is calibrated, it will be more accurate (in a least square sense), but it will sometimes produce negative numbers (when call counts are exceptionally low, and the gods of probability work against you :-). ) Do not be alarmed by negative numbers in the profile. They should only appear if you have calibrated your profiler, and the results are actually better than without calibration.

27.4.7 Calibration

The profiler of the profile module subtracts a constant from each event handling time to compensate for the overhead of calling the time function, and socking away the results. By default, the constant is 0. The following procedure can be used to obtain a better constant for a given platform (see Limitations).

```python
import profile
pr = profile.Profile()
for i in range(5):
    print(pr.calibrate(10000))
```

The method executes the number of Python calls given by the argument, directly and again under the profiler, measuring the time for both. It then computes the hidden overhead per profiler event, and returns that as a float. For example, on a 1.8Ghz Intel Core i5 running Mac OS X, and using Python’s time.clock() as the timer, the magical number is about 4.04e-6.

The object of this exercise is to get a fairly consistent result. If your computer is very fast, or your timer function has poor resolution, you might have to pass 100000, or even 1000000, to get consistent results.

When you have a consistent answer, there are three ways you can use it:

```python
import profile

# 1. Apply computed bias to all Profile instances created hereafter.
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

If you have a choice, you are better off choosing a smaller constant, and then your results will “less often” show up as negative in profile statistics.
27.4.8 Using a custom timer

If you want to change how current time is determined (for example, to force use of wall-clock time or elapsed process time), pass the timing function you want to the Profile class constructor:

\[
pr = profile.Profile(your_time_func)
\]

The resulting profiler will then call your_time_func. Depending on whether you are using profile.Profile or cProfile.Profile, your_time_func's return value will be interpreted differently:

**profile.Profile** your_time_func should return a single number, or a list of numbers whose sum is the current time (like what `os.times()` returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you should calibrate the profiler class for the timer function that you choose (see Calibration). For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (`os.times()` is pretty bad, as it returns a tuple of floating point values). If you want to substitute a better timer in the cleanest fashion, derive a class and hardwire a replacement dispatch method that best handles your timer call, along with the appropriate calibration constant.

**cProfile.Profile** your_time_func should return a single number. If it returns integers, you can also invoke the class constructor with a second argument specifying the real duration of one unit of time. For example, if your_integer_time_func returns times measured in thousands of seconds, you would construct the Profile instance as follows:

\[
pr = cProfile.Profile(your_integer_time_func, 0.001)
\]

As the cProfile.Profile class cannot be calibrated, custom timer functions should be used with care and should be as fast as possible. For the best results with a custom timer, it might be necessary to hard-code it in the C source of the internal _lsprof module.

Python 3.3 adds several new functions in time that can be used to make precise measurements of process or wall-clock time. For example, see `time.perf_counter()`.

27.5 timeit — Measure execution time of small code snippets

Source code: Lib/timeit.py

This module provides a simple way to time small bits of Python code. It has both a Command-Line Interface as well as a callable one. It avoids a number of common traps for measuring execution times. See also Tim Peters’ introduction to the “Algorithms” chapter in the Python Cookbook, published by O’Reilly.

27.5.1 Basic Examples

The following example shows how the Command-Line Interface can be used to compare three different expressions:

\[
\begin{align*}
\texttt{\$ python -m timeit "\"-\".join(str(n) for n in range(100))\""} \\
10000 \text{ loops, best of 3: 40.3 usec per loop} \\
\texttt{\$ python -m timeit "\"-\".join([str(n) for n in range(100)])\""} \\
10000 \text{ loops, best of 3: 33.4 usec per loop} \\
\texttt{\$ python -m timeit "\"-\".join(map(str, range(100)))\""} \\
10000 \text{ loops, best of 3: 25.2 usec per loop}
\end{align*}
\]

This can be achieved from the Python Interface with:

```python
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.8187260627746582
```
>>> timeit.timeit("""-""".join([str(n) for n in range(100)])", number=10000)
0.7288308143615723
>>> timeit.timeit("""-""".join(map(str, range(100)))", number=10000)
0.5858950614929199

Note however that `timeit` will automatically determine the number of repetitions only when the command-line interface is used. In the `Examples` section you can find more advanced examples.

### 27.5.2 Python Interface

The module defines three convenience functions and a public class:

```python
timeit.timeit(stmt='pass', setup='pass', timer=<default timer>, number=1000000)
    Create a `Timer` instance with the given statement, setup code and timer function and run its `timeit()` method with number executions.
```

```python
timeit.repeat(stmt='pass', setup='pass', timer=<default timer>, repeat=3, number=1000000)
    Create a `Timer` instance with the given statement, setup code and timer function and run its `repeat()` method with the given repeat count and number executions.
```

```python
timeit.default_timer()
    The default timer, which is always `time.perf_counter()`. Changed in version 3.3: `time.perf_counter()` is now the default timer.
```

```python
class timeit.Timer(stmt='pass', setup='pass', timer=<timer function>)
    Class for timing execution speed of small code snippets.
    The constructor takes a statement to be timed, an additional statement used for setup, and a timer function. Both statements default to 'pass'; the timer function is platform-dependent (see the module doc string). `stmt` and `setup` may also contain multiple statements separated by ; or newlines, as long as they don’t contain multi-line string literals.

    To measure the execution time of the first statement, use the `timeit()` method. The `repeat()` method is a convenience to call `timeit()` multiple times and return a list of results.

    The `stmt` and `setup` parameters can also take objects that are callable without arguments. This will embed calls to them in a timer function that will then be executed by `timeit()`. Note that the timing overhead is a little larger in this case because of the extra function calls.
```

```python
timeit(time=number=1000000)
    Time number executions of the main statement. This executes the setup statement once, and then returns the time it takes to execute the main statement a number of times, measured in seconds as a float. The argument is the number of times through the loop, defaulting to one million. The main statement, the setup statement and the timer function to be used are passed to the constructor.
```

**Note:** By default, `timeit()` temporarily turns off `garbage collection` during the timing. The advantage of this approach is that it makes independent timings more comparable. This disadvantage is that GC may be an important component of the performance of the function being measured. If so, GC can be re-enabled as the first statement in the `setup` string. For example:

```python
timeit.Timer('for i in range(10): oct(i)', 'gc.enable()').timeit()
```

```python
repeat(repeat=3, number=1000000)
    Call `timeit()` a few times.
    This is a convenience function that calls the `timeit()` repeatedly, returning a list of results. The first argument specifies how many times to call `timeit()`. The second argument specifies the `number` argument for `timeit()`.
```

**Note:** It’s tempting to calculate mean and standard deviation from the result vector and report these. However, this is not very useful. In a typical case, the lowest value gives a lower bound for how fast
your machine can run the given code snippet; higher values in the result vector are typically not caused by variability in Python’s speed, but by other processes interfering with your timing accuracy. So the \texttt{min()} of the result is probably the only number you should be interested in. After that, you should look at the entire vector and apply common sense rather than statistics.

\begin{verbatim}
print_exc(file=None)
\end{verbatim}

Helper to print a traceback from the timed code.

Typical use:

\begin{verbatim}
t = Timer(...) # outside the try/except
try:
    t.timeit(...) # or t.repeat(...)
except:
    t.print_exc()
\end{verbatim}

The advantage over the standard traceback is that source lines in the compiled template will be displayed. The optional \texttt{file} argument directs where the traceback is sent; it defaults to \texttt{sys.stderr}.

### 27.5.3 Command-Line Interface

When called as a program from the command line, the following form is used:

\begin{verbatim}
python -m timeit [-n N] [-r N] [-s S] [-t] [-c] [-h] [statement ...]
\end{verbatim}

Where the following options are understood:

- \texttt{-n N}, \texttt{--number=N}  
  how many times to execute 'statement'

- \texttt{-r N}, \texttt{--repeat=N}  
  how many times to repeat the timer (default 3)

- \texttt{-s S}, \texttt{--setup=S}  
  statement to be executed once initially (default \texttt{pass})

- \texttt{-p}, \texttt{--process}  
  measure process time, not wallclock time, using \texttt{time.process_time()} instead of \texttt{time.perf_counter()}, which is the default New in version 3.3.

- \texttt{-t}, \texttt{--time}  
  use \texttt{time.time()} (deprecated)

- \texttt{-c}, \texttt{--clock}  
  use \texttt{time.clock()} (deprecated)

- \texttt{-v}, \texttt{--verbose}  
  print raw timing results; repeat for more digits precision

- \texttt{-h}, \texttt{--help}  
  print a short usage message and exit

A multi-line statement may be given by specifying each line as a separate statement argument; indented lines are possible by enclosing an argument in quotes and using leading spaces. Multiple \texttt{-s} options are treated similarly.

If \texttt{-n} is not given, a suitable number of loops is calculated by trying successive powers of 10 until the total time is at least 0.2 seconds.

\texttt{default_timer()} measurements can be affected by other programs running on the same machine, so the best thing to do when accurate timing is necessary is to repeat the timing a few times and use the best time. The \texttt{-r} option is good for this; the default of 3 repetitions is probably enough in most cases. You can use \texttt{time.process_time()} to measure CPU time.
Note: There is a certain baseline overhead associated with executing a pass statement. The code here doesn’t try to hide it, but you should be aware of it. The baseline overhead can be measured by invoking the program without arguments, and it might differ between Python versions.

27.5.4 Examples

It is possible to provide a setup statement that is executed only once at the beginning:

```
$ python -m timeit -s 'text = "sample string"; char = "g"' 'char in text'
1000000 loops, best of 3: 0.0877 usec per loop
$ python -m timeit -s 'text = "sample string"; char = "g"' 'text.find(char)'
1000000 loops, best of 3: 0.342 usec per loop
``` 

```
>>> import timeit

>>> timeit.timeit('char in text', setup='text = "sample string"; char = "g"')
0.41440500499993504
>>> timeit.timeit('text.find(char)', setup='text = "sample string"; char = "g"')
1.7246671520006203
``` 

The same can be done using the Timer class and its methods:

```
>>> import timeit

>>> t = timeit.Timer('char in text', setup='text = "sample string"; char = "g"')
>>> t.timeit()
0.395516149999312
>>> t.repeat()
[0.40193588800002544, 0.3960157959998014, 0.39594301399984033]
``` 

The following examples show how to time expressions that contain multiple lines. Here we compare the cost of using `hasattr()` vs. `try/except` to test for missing and present object attributes:

```
$ python -m timeit 'try:' ' str.__bool__' 'except AttributeError:' ' pass'
100000 loops, best of 3: 15.7 usec per loop
$ python -m timeit 'if hasattr(str, '__bool__'): pass'
100000 loops, best of 3: 4.26 usec per loop
```

```
$ python -m timeit 'try:' ' int.__bool__' 'except AttributeError:' ' pass'
1000000 loops, best of 3: 1.43 usec per loop
$ python -m timeit 'if hasattr(int, '__bool__'): pass'
100000 loops, best of 3: 2.23 usec per loop
``` 

```
>>> import timeit

>>> # attribute is missing

>>> s = """\n... try:
...    str.__bool__
... except AttributeError:
...    pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.9138244460009446
>>> s = "if hasattr(str, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.5829014980008651
>>> 
>>> # attribute is present

>>> s = """\n... try:
...    int.__bool__
... except AttributeError:
... """
```
...    pass
...""
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603

To give the timeit module access to functions you define, you can pass a setup parameter which contains an import statement:

def test():
    """Stupid test function""
    L = [i for i in range(100)]
    if __name__ == '__main__':
        import timeit
        print(timeit.timeit("test()", setup="from __main__ import test"))

27.6 trace — Trace or track Python statement execution

Source code: Lib/trace.py

The trace module allows you to trace program execution, generate annotated statement coverage listings, print caller/callee relationships and list functions executed during a program run. It can be used in another program or from the command line.

27.6.1 Command-Line Usage

The trace module can be invoked from the command line. It can be as simple as

```
python -m trace --count -C . somefile.py ...
```

The above will execute somefile.py and generate annotated listings of all Python modules imported during the execution into the current directory.

- **-help**
  Display usage and exit.

- **-version**
  Display the version of the module and exit.

Main options

At least one of the following options must be specified when invoking trace. The --listfuncs option is mutually exclusive with the --trace and --counts options. When --listfuncs is provided, neither --counts nor --trace are accepted, and vice versa.

- **-c, -count**
  Produce a set of annotated listing files upon program completion that shows how many times each statement was executed. See also --coverdir, --file and --no-report below.

- **-t, -trace**
  Display lines as they are executed.

- **-l, -listfuncs**
  Display the functions executed by running the program.
-r, --report
   Produce an annotated list from an earlier program run that used the --count and --file option. This
does not execute any code.

-T, --trackcalls
   Display the calling relationships exposed by running the program.

Modifiers

-f, --file=<file>
   Name of a file to accumulate counts over several tracing runs. Should be used with the --count option.

-C, --coverdir=<dir>
   Directory where the report files go. The coverage report for package.module is written to file
dir/package/module.cover.

-m, --missing
   When generating annotated listings, mark lines which were not executed with >>>>>>>.

-s, --summary
   When using --count or --report, write a brief summary to stdout for each file processed.

-R, --no-report
   Do not generate annotated listings. This is useful if you intend to make several runs with --count, and
then produce a single set of annotated listings at the end.

-g, --timing
   Prefix each line with the time since the program started. Only used while tracing.

Filters

These options may be repeated multiple times.

-ignore-module=<mod>
   Ignore each of the given module names and its submodules (if it is a package). The argument can be a list
of names separated by a comma.

-ignore-dir=<dir>
   Ignore all modules and packages in the named directory and subdirectories. The argument can be a list
of directories separated by os.pathsep.

27.6.2 Programmatic Interface

class trace.Trace(count=1, trace=1, countfuncs=0, countcallers=0, ignoremods=(), ignoredirs=(),
infile=None, outfile=None, timing=False)
   Create an object to trace execution of a single statement or expression. All parameters are optional. count
enables counting of line numbers. trace enables line execution tracing. countfuncs enables listing of the
functions called during the run. countcallers enables call relationship tracking. ignoremods is a list of
modules or packages to ignore. ignoredirs is a list of directories whose modules or packages should be
ignored. infile is the name of the file from which to read stored count information. outfile is the name of the
file in which to write updated count information. timing enables a timestamp relative to when tracing was
started to be displayed.

   run(cmd)
      Execute the command and gather statistics from the execution with the current tracing pa-
terms. cmd must be a string or code object, suitable for passing into exec() .

   runctx(cmd, globals=None, locals=None)
      Execute the command and gather statistics from the execution with the current tracing pa-
terms, in the defined global and local environments. If not defined, globals and locals
default to empty dictionaries.
runfunc (func, *args, **kwds)
Call func with the given arguments under control of the Trace object with the current tracing parameters.

results()
Return a CoverageResults object that contains the cumulative results of all previous calls to run, runctx and runfunc for the given Trace instance. Does not reset the accumulated trace results.

class trace.CoverageResults
A container for coverage results, created by Trace.results(). Should not be created directly by the user.

update(other)
Merge in data from another CoverageResults object.

write_results(show_missing=True, summary=False, coverdir=None)
Write coverage results. Set show_missing to show lines that had no hits. Set summary to include in the output the coverage summary per module. coverdir specifies the directory into which the coverage result files will be output. If None, the results for each source file are placed in its directory.

A simple example demonstrating the use of the programmatic interface:

```python
import sys
import trace

# create a Trace object, telling it what to ignore, and whether to do tracing or line-counting or both.
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)

# run the new command using the given tracer
tracer.run('main()')

# make a report, placing output in the current directory
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")
```
The modules described in this chapter provide a wide range of services related to the Python interpreter and its interaction with its environment. Here’s an overview:

28.1 sys — System-specific parameters and functions

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available.

**sys.abiflags**

On POSIX systems where Python is build with the standard configure script, this contains the ABI flags as specified by PEP 3149. New in version 3.2.

**sys.argv**

The list of command line arguments passed to a Python script. argv[0] is the script name (it is operating system dependent whether this is a full pathname or not). If the command was executed using the `-c` command line option to the interpreter, argv[0] is set to the string `-c`. If no script name was passed to the Python interpreter, argv[0] is the empty string.

To loop over the standard input, or the list of files given on the command line, see the fileinput module.

**sys.base_exec_prefix**

Set during Python startup, before site.py is run, to the same value as exec_prefix. If not running in a virtual environment, the values will stay the same; if site.py finds that a virtual environment is in use, the values of prefix and exec_prefix will be changed to point to the virtual environment, whereas base_prefix and base_exec_prefix will remain pointing to the base Python installation (the one which the virtual environment was created from). New in version 3.3.

**sys.base_prefix**

Set during Python startup, before site.py is run, to the same value as prefix. If not running in a virtual environment, the values will stay the same; if site.py finds that a virtual environment is in use, the values of prefix and exec_prefix will be changed to point to the virtual environment, whereas base_prefix and base_exec_prefix will remain pointing to the base Python installation (the one which the virtual environment was created from). New in version 3.3.

**sys.byteorder**

An indicator of the native byte order. This will have the value ‘big’ on big-endian (most-significant byte first) platforms, and ‘little’ on little-endian (least-significant byte first) platforms.

**sys.builtin_module_names**

A tuple of strings giving the names of all modules that are compiled into this Python interpreter. (This information is not available in any other way — modules.keys() only lists the imported modules.)

**sys.call_tracing(func, args)**

Call func(*args), while tracing is enabled. The tracing state is saved, and restored afterwards. This is intended to be called from a debugger from a checkpoint, to recursively debug some other code.
**sys.copyright**
A string containing the copyright pertaining to the Python interpreter.

**sys._clear_type_cache()**
Clear the internal type cache. The type cache is used to speed up attribute and method lookups. Use the function *only* to drop unnecessary references during reference leak debugging.

This function should be used for internal and specialized purposes only.

**sys._current_frames()**
Return a dictionary mapping each thread’s identifier to the topmost stack frame currently active in that thread at the time the function is called. Note that functions in the `traceback` module can build the call stack given such a frame.

This is most useful for debugging deadlock: this function does not require the deadlocked threads’ cooperation, and such threads’ call stacks are frozen for as long as they remain deadlocked. The frame returned for a non-deadlocked thread may bear no relationship to that thread’s current activity by the time calling code examines the frame.

This function should be used for internal and specialized purposes only.

**sys._debugmallocstats()**
Print low-level information to stderr about the state of CPython’s memory allocator.

If Python is configured `--with-pydebug`, it also performs some expensive internal consistency checks. New in version 3.3. **CPython implementation detail:** This function is specific to CPython. The exact output format is not defined here, and may change.

**sys.dllhandle**
Integer specifying the handle of the Python DLL. Availability: Windows.

**sys.displayhook(value)**
If `value` is not `None`, this function prints `repr(value)` to `sys.stdout`, and saves `value` in `builtins._`. If `repr(value)` is not encodable to `sys.stdout.encoding` with `sys.stdout.errors` error handler (which is probably ‘strict’), encode it to `sys.stdout.encoding` with ‘backslashreplace’ error handler.

`sys.displayhook` is called on the result of evaluating an `expression` entered in an interactive Python session. The display of these values can be customized by assigning another one-argument function to `sys.displayhook`.

Pseudo-code:

```python
def displayhook(value):
    if value is None:
        return
    # Set ‘_’ to None to avoid recursion
    builtins._ = None
    text = repr(value)
    try:
        sys.stdout.write(text)
    except UnicodeEncodeError:
        bytes = text.encode(sys.stdout.encoding, 'backslashreplace')
        if hasattr(sys.stdout, 'buffer'):
            sys.stdout.buffer.write(bytes)
        else:
            text = bytes.decode(sys.stdout.encoding, 'strict')
        sys.stdout.write("\n")
        builtins._ = value
```

Changed in version 3.2: Use ‘backslashreplace’ error handler on `UnicodeEncodeError`. 

---

1216 Chapter 28. Python Runtime Services
**sys.dont_write_bytecode**

If this is true, Python won’t try to write `.pyc` or `.pyo` files on the import of source modules. This value is initially set to `True` or `False` depending on the `-B` command line option and the `PYTHONDONTHWRITEBYTECODE` environment variable, but you can set it yourself to control bytecode file generation.

**sys.excepthook**(type, value, traceback)

This function prints out a given traceback and exception to `sys.stderr`.

When an exception is raised and uncaught, the interpreter calls `sys.excepthook` with three arguments, the exception class, exception instance, and a traceback object. In an interactive session this happens just before control is returned to the prompt; in a Python program this happens just before the program exits. The handling of such top-level exceptions can be customized by assigning another three-argument function to `sys.excepthook`.

**sys._displayhook__**

**sys._excepthook__**

These objects contain the original values of `displayhook` and `excepthook` at the start of the program. They are saved so that `displayhook` and `excepthook` can be restored in case they happen to get replaced with broken objects.

**sys.exc_info()**

This function returns a tuple of three values that give information about the exception that is currently being handled. The information returned is specific both to the current thread and to the current stack frame. If the current stack frame is not handling an exception, the information is taken from the calling stack frame, or its caller, and so on until a stack frame is found that is handling an exception. Here, “handling an exception” is defined as “executing an except clause.” For any stack frame, only information about the exception being currently handled is accessible.

If no exception is being handled anywhere on the stack, a tuple containing three `None` values is returned. Otherwise, the values returned are `(type, value, traceback)`. Their meaning is: `type` gets the type of the exception being handled (a subclass of `BaseException`); `value` gets the exception instance (an instance of the exception type); `traceback` gets a traceback object (see the Reference Manual) which encapsulates the call stack at the point where the exception originally occurred.

**sys.exec_prefix**

A string giving the site-specific directory prefix where the platform-dependent Python files are installed; by default, this is also `/usr/local`. This can be set at build time with the `--exec-prefix` argument to the `configure` script. Specifically, all configuration files (e.g. the `pyconfig.h` header file) are installed in the directory `exec_prefix/lib/pythonX.Y/config`, and shared library modules are installed in `exec_prefix/lib/pythonX.Y/lib-dynload`, where `X.Y` is the version number of Python, for example 3.2.

**Note:** If a virtual environment is in effect, this value will be changed in `site.py` to point to the virtual environment. The value for the Python installation will still be available, via `base_exec_prefix`.

**sys.executable**

A string giving the absolute path of the executable binary for the Python interpreter, on systems where this makes sense. If Python is unable to retrieve the real path to its executable, `sys.executable` will be an empty string or `None`.

**sys.exit([arg])**

Exit from Python. This is implemented by raising the `SystemExit` exception, so cleanup actions specified by finally clauses of `try` statements are honored, and it is possible to intercept the exit attempt at an outer level.

The optional argument `arg` can be an integer giving the exit status (defaulting to zero), or another type of object. If it is an integer, zero is considered “successful termination” and any nonzero value is considered “abnormal termination” by shells and the like. Most systems require it to be in the range 0-127, and produce undefined results otherwise. Some systems have a convention for assigning specific meanings to specific exit codes, but these are generally underdeveloped; Unix programs generally use 2 for command
line syntax errors and 1 for all other kind of errors. If another type of object is passed, None is equivalent to passing zero, and any other object is printed to stderr and results in an exit code of 1. In particular, sys.exit("some error message") is a quick way to exit a program when an error occurs.

Since exit() ultimately “only” raises an exception, it will only exit the process when called from the main thread, and the exception is not intercepted.

**sys.flags**

The struct sequence flags exposes the status of command line flags. The attributes are read only.

<table>
<thead>
<tr>
<th>attribute</th>
<th>flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug</td>
<td>-d</td>
</tr>
<tr>
<td>inspect</td>
<td>-i</td>
</tr>
<tr>
<td>interactive</td>
<td>-i</td>
</tr>
<tr>
<td>optimize</td>
<td>-O or -OO</td>
</tr>
<tr>
<td>dont_write_bytecode</td>
<td>-B</td>
</tr>
<tr>
<td>no_user_site</td>
<td>-s</td>
</tr>
<tr>
<td>no_site</td>
<td>-S</td>
</tr>
<tr>
<td>ignore_environment</td>
<td>-E</td>
</tr>
<tr>
<td>verbose</td>
<td>-v</td>
</tr>
<tr>
<td>bytes_warning</td>
<td>-b</td>
</tr>
<tr>
<td>quiet</td>
<td>-q</td>
</tr>
<tr>
<td>hash_randomization</td>
<td>-R</td>
</tr>
</tbody>
</table>


**sys.float_info**

A struct sequence holding information about the float type. It contains low level information about the precision and internal representation. The values correspond to the various floating-point constants defined in the standard header file float.h for the ‘C’ programming language; see section 5.2.4.2.2 of the 1999 ISO/IEC C standard [C99], ‘Characteristics of floating types’, for details.

<table>
<thead>
<tr>
<th>attribute</th>
<th>float.h macro</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>epsilon</td>
<td>DBL_EPSILON</td>
<td>difference between 1 and the least value greater than 1 that is representable as a float</td>
</tr>
<tr>
<td>dig</td>
<td>DBL_DIG</td>
<td>maximum number of decimal digits that can be faithfully represented in a float; see below</td>
</tr>
<tr>
<td>mant_dig</td>
<td>DBL_MANT_DIG</td>
<td>float precision: the number of base-radix digits in the significand of a float</td>
</tr>
<tr>
<td>max</td>
<td>DBL_MAX</td>
<td>maximum representable finite float</td>
</tr>
<tr>
<td>max_exp</td>
<td>DBL_MAX_EXP</td>
<td>maximum integer e such that (\text{radix}^{(e-1)}) is a representable finite float</td>
</tr>
<tr>
<td>max_10_exp</td>
<td>DBL_MAX_10_EXP</td>
<td>maximum integer e such that (10^{e}) is in the range of representable finite floats</td>
</tr>
<tr>
<td>min</td>
<td>DBL_MIN</td>
<td>minimum positive normalized float</td>
</tr>
<tr>
<td>min_exp</td>
<td>DBL_MIN_EXP</td>
<td>minimum integer e such that (\text{radix}^{(e-1)}) is a normalized float</td>
</tr>
<tr>
<td>min_10_exp</td>
<td>DBL_MIN_10_EXP</td>
<td>minimum integer e such that (10^{e}) is a normalized float</td>
</tr>
<tr>
<td>radix</td>
<td>FLT_RADIX</td>
<td>integer constant representing the rounding mode used for arithmetic operations. This reflects the value of the system FLT_Rounds macro at interpreter startup time. See section 5.2.4.2.2 of the C99 standard for an explanation of the possible values and their meanings.</td>
</tr>
<tr>
<td>rounds</td>
<td>FLT_ROUNDS</td>
<td></td>
</tr>
</tbody>
</table>

The attribute sys.float_info.dig needs further explanation. If \(s\) is any string representing a decimal number with at most sys.float_info.dig significant digits, then converting \(s\) to a float and back again will recover a string representing the same decimal value:

```python
>>> import sys
```
>>> sys.float_info.dig
15
>>> s = '3.14159265358979'  # decimal string with 15 significant digits
>>> format(float(s), '.15g')  # convert to float and back -> same value
'3.14159265358979'

But for strings with more than sys.float_info.dig significant digits, this isn’t always true:

>>> s = '9876543211234567'  # 16 significant digits is too many!
>>> format(float(s), '.16g')  # conversion changes value
'9876543211234568'

**sys.float_repr_style**
A string indicating how the `repr()` function behaves for floats. If the string has value ‘short’ then for a finite float \(x\), `repr(x)` aims to produce a short string with the property that `float(repr(x)) == x`. This is the usual behaviour in Python 3.1 and later. Otherwise, `float_repr_style` has value ‘legacy’ and `repr(x)` behaves in the same way as it did in versions of Python prior to 3.1. New in version 3.1.

**sys.getcheckinterval()**
Return the interpreter’s “check interval”; see `setcheckinterval()`. Deprecated since version 3.2: Use `getswitchinterval()` instead.

**sys.getdefaultencoding()**
Return the name of the current default string encoding used by the Unicode implementation.

**sys.getdlopenflags()**
Return the current value of the flags that are used for `dlopen()` calls. The flag constants are defined in the `ctypes` and `DLFCN` modules. Availability: Unix.

**sys.getfilesystemencoding()**
Return the name of the encoding used to convert Unicode filenames into system file names. The result value depends on the operating system:

- On Mac OS X, the encoding is ’utf-8’.
- On Unix, the encoding is the user’s preference according to the result of `nl_langinfo(CODESET)`, or ’utf-8’ if `nl_langinfo(CODESET)` failed.
- On Windows NT+, file names are Unicode natively, so no conversion is performed. `getfilesystemencoding()` still returns ’mbcs’, as this is the encoding that applications should use when they explicitly want to convert Unicode strings to byte strings that are equivalent when used as file names.
- On Windows 9x, the encoding is ’mbcs’.

Changed in version 3.2: On Unix, use ’utf-8’ instead of None if `nl_langinfo(CODESET)` failed. `getfilesystemencoding()` result cannot be None.

**sys.getrefcount**(object)
Return the reference count of the `object`. The count returned is generally one higher than you might expect, because it includes the (temporary) reference as an argument to `getrefcount()`.

**sys.getrecursionlimit()**
Return the current value of the recursion limit, the maximum depth of the Python interpreter stack. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python. It can be set by `setrecursionlimit()`.

**sys.getsizeof**(object[, default])
Return the size of an object in bytes. The object can be any type of object. All built-in objects will return correct results, but this does not have to hold true for third-party extensions as it is implementation specific. Only the memory consumption directly attributed to the object is accounted for, not the memory consumption of objects it refers to.
If given, `default` will be returned if the object does not provide means to retrieve the size. Otherwise a `TypeError` will be raised.

`getsizeof()` calls the object’s `__sizeof__` method and adds an additional garbage collector overhead if the object is managed by the garbage collector.

See recursive sizeof recipe for an example of using `getsizeof()` recursively to find the size of containers and all their contents.

```python
sys.getswitchinterval()
```

Return the interpreter’s “thread switch interval”; see `setswitchinterval()`. New in version 3.2.

```python
sys._getframe([depth])
```

Return a frame object from the call stack. If optional integer `depth` is given, return the frame object that many calls below the top of the stack. If that is deeper than the call stack, `ValueError` is raised. The default for `depth` is zero, returning the frame at the top of the call stack.

**CPython implementation detail:** This function should be used for internal and specialized purposes only. It is not guaranteed to exist in all implementations of Python.

```python
sys.getprofile()
```

Get the profiler function as set by `setprofile()`.

```python
sys.gettrace()
```

Get the trace function as set by `settrace()`.

**CPython implementation detail:** The `gettrace()` function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

```python
sys.getwindowsversion()
```

Return a named tuple describing the Windows version currently running. The named elements are `major`, `minor`, `build`, `platform`, `service_pack`, `service_pack_minor`, `service_pack_major`, `suite_mask`, and `product_type`. `service_pack` contains a string while all other values are integers. The components can also be accessed by name, so `sys.getwindowsversion()[0]` is equivalent to `sys.getwindowsversion().major`. For compatibility with prior versions, only the first 5 elements are retrievable by indexing.

`platform` may be one of the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (VER_PLATFORM_WIN32s)</td>
<td>Win32s on Windows 3.1</td>
</tr>
<tr>
<td>1 (VER_PLATFORM_WIN32_WINDOWS)</td>
<td>Windows 95/98/ME</td>
</tr>
<tr>
<td>2 (VER_PLATFORM_WIN32_NT)</td>
<td>Windows NT/2000/XP/x64</td>
</tr>
<tr>
<td>3 (VER_PLATFORM_WIN32_CE)</td>
<td>Windows CE</td>
</tr>
</tbody>
</table>

`product_type` may be one of the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (VER_NT_WORKSTATION)</td>
<td>The system is a workstation.</td>
</tr>
<tr>
<td>2 (VER_NT_DOMAIN_CONTROLLER)</td>
<td>The system is a domain controller.</td>
</tr>
<tr>
<td>3 (VER_NT_SERVER)</td>
<td>The system is a server, but not a domain controller.</td>
</tr>
</tbody>
</table>

This function wraps the Win32 `GetVersionEx()` function; see the Microsoft documentation on `OSVERSIONINFOEX()` for more information about these fields.

Availability: Windows. Changed in version 3.2: Changed to a named tuple and added `service_pack_minor`, `service_pack_major`, `suite_mask`, and `product_type`.

```python
sys.hash_info
```

A `struct sequence` giving parameters of the numeric hash implementation. For more details about hashing of numeric types, see Hashing of numeric types.
The Python Library Reference, Release 3.3.3

**sys.hexversion**

The version number encoded as a single integer. This is guaranteed to increase with each version, including proper support for non-production releases. For example, to test that the Python interpreter is at least version 1.5.2, use:

```python
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
...
else:
    # use an alternative implementation or warn the user
...
```

This is called `hexversion` since it only really looks meaningful when viewed as the result of passing it to the built-in `hex()` function. The `struct sequence` `sys.version_info` may be used for a more human-friendly encoding of the same information.

More details of `hexversion` can be found at `apiabiversion`

**sys.implementation**

An object containing information about the implementation of the currently running Python interpreter. The following attributes are required to exist in all Python implementations.

- **name** is the implementation’s identifier, e.g. ‘cpython’. The actual string is defined by the Python implementation, but it is guaranteed to be lower case.

- **version** is a named tuple, in the same format as `sys.version_info`. It represents the version of the Python implementation. This has a distinct meaning from the specific version of the Python language to which the currently running interpreter conforms, which `sys.version_info` represents. For example, for PyPy 1.8 `sys.implementation.version` might be `sys.version_info(1, 8, 0, 'final', 0)`, whereas `sys.version_info` would be `sys.version_info(2, 7, 2, 'final', 0)`. For CPython they are the same value, since it is the reference implementation.

`hexversion` is the implementation version in hexadecimal format, like `sys.hexversion`.

`cache_tag` is the tag used by the import machinery in the filenames of cached modules. By convention, it would be a composite of the implementation’s name and version, like ‘cpython-33’. However, a Python implementation may use some other value if appropriate. If `cache_tag` is set to `None`, it indicates that module caching should be disabled.

`sys.implementation` may contain additional attributes specific to the Python implementation. These non-standard attributes must start with an underscore, and are not described here. Regardless of its contents, `sys.implementation` will not change during a run of the interpreter, nor between implementation versions. (It may change between Python language versions, however.) See `PEP 421` for more information.

New in version 3.3.

**sys.int_info**

A `struct sequence` that holds information about Python’s internal representation of integers. The attributes are read only.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits_per_digit</td>
<td>number of bits held in each digit. Python integers are stored internally in base $2^{\text{int_info.bits_per_digit}}$</td>
</tr>
<tr>
<td>sizeof_digit</td>
<td>size in bytes of the C type used to represent a digit</td>
</tr>
</tbody>
</table>

New in version 3.1.
sys.intern(string)
Enter string in the table of “interned” strings and return the interned string – which is string itself or a copy. Interning strings is useful to gain a little performance on dictionary lookup – if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare. Normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys.

Interned strings are not immortal; you must keep a reference to the return value of intern() around to benefit from it.

sys.last_type
sys.last_value
sys.last_traceback
These three variables are not always defined; they are set when an exception is not handled and the interpreter prints an error message and a stack traceback. Their intended use is to allow an interactive user to import a debugger module and engage in post-mortem debugging without having to re-execute the command that caused the error. (Typical use is import pdb; pdb.pm() to enter the post-mortem debugger; see pdb module for more information.)

The meaning of the variables is the same as that of the return values from exc_info() above.

sys.maxsize
An integer giving the maximum value a variable of type Py_ssize_t can take. It’s usually $2^{31} - 1$ on a 32-bit platform and $2^{63} - 1$ on a 64-bit platform.

sys.maxunicode
An integer giving the value of the largest Unicode code point, i.e. 1114111 (0x10FFFF in hexadecimal). Changed in version 3.3: Before PEP 393, sys.maxunicode used to be either 0xFFFF or 0x10FFFF, depending on the configuration option that specified whether Unicode characters were stored as UCS-2 or UCS-4.

sys.meta_path
A list of finder objects that have their find_module() methods called to see if one of the objects can find the module to be imported. The find_module() method is called at least with the absolute name of the module being imported. If the module to be imported is contained in a package then the parent package’s __path__ attribute is passed in as a second argument. The method returns None if the module cannot be found, else returns a loader.

sys.meta_path is searched before any implicit default finders or sys.path.

See PEP 302 for the original specification.

sys.modules
This is a dictionary that maps module names to modules which have already been loaded. This can be manipulated to force reloading of modules and other tricks. However, replacing the dictionary will not necessarily work as expected and deleting essential items from the dictionary may cause Python to fail.

sys.path
A list of strings that specifies the search path for modules. Initialized from the environment variable PYTHONPATH, plus an installation-dependent default.

As initialized upon program startup, the first item of this list, path[0], is the directory containing the script that was used to invoke the Python interpreter. If the script directory is not available (e.g. if the interpreter is invoked interactively or if the script is read from standard input), path[0] is the empty string, which directs Python to search modules in the current directory first. Notice that the script directory is inserted before the entries inserted as a result of PYTHONPATH.

A program is free to modify this list for its own purposes. Only strings and bytes should be added to sys.path; all other data types are ignored during import.

See Also:
Module site This describes how to use .pth files to extend sys.path.
The Python Library Reference, Release 3.3.3

**sys.path_hooks**
A list of callables that take a path argument to try to create a *finder* for the path. If a finder can be created, it is to be returned by the callable, else raise *ImportError*.

Originally specified in **PEP 302**.

**sys.path_importer_cache**
A dictionary acting as a cache for *finder* objects. The keys are paths that have been passed to *sys.path_hooks* and the values are the finders that are found. If a path is a valid file system path but no finder is found on *sys.path_hooks* then *None* is stored.

Originally specified in **PEP 302**. Changed in version 3.3: *None* is stored instead of *Imp.NullImporter* when no finder is found.

**sys.platform**
This string contains a platform identifier that can be used to append platform-specific components to *sys.path*, for instance.

For Unix systems, except on Linux, this is the lowercased OS name as returned by *uname* -s with the first part of the version as returned by *uname* -r appended, e.g. ‘sunos5’ or ‘freebsd8’, *at the time when Python was built*. Unless you want to test for a specific system version, it is therefore recommended to use the following idiom:

```python
if sys.platform.startswith('freebsd'):
    # FreeBSD-specific code here...
elif sys.platform.startswith('linux'):
    # Linux-specific code here...
```

For other systems, the values are:

<table>
<thead>
<tr>
<th>System</th>
<th>platform value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>‘linux’</td>
</tr>
<tr>
<td>Windows</td>
<td>‘win32’</td>
</tr>
<tr>
<td>Windows/Cygwin</td>
<td>‘cygwin’</td>
</tr>
<tr>
<td>Mac OS X</td>
<td>‘darwin’</td>
</tr>
<tr>
<td>OS/2</td>
<td>‘os2’</td>
</tr>
<tr>
<td>OS/2 EMX</td>
<td>‘os2emx’</td>
</tr>
</tbody>
</table>

Changed in version 3.3: On Linux, *sys.platform* doesn’t contain the major version anymore. It is always ‘linux’, instead of ‘linux2’ or ‘linux3’. Since older Python versions include the version number, it is recommended to always use the *startswith* idiom presented above.

**See Also:**
*os.name* has a coarser granularity. *os.uname()* gives system-dependent version information.

The *platform* module provides detailed checks for the system’s identity.

**sys.prefix**
A string giving the site-specific directory prefix where the platform independent Python files are installed; by default, this is the string ‘/usr/local’. This can be set at build time with the *--prefix* argument to the *configure* script. The main collection of Python library modules is installed in the directory *prefix/lib/pythonX.Y* while the platform independent header files (all except *pyconfig.h*) are stored in *prefix/include/pythonX.Y*, where *X.Y* is the version number of Python, for example 3.2.

**Note:** If a virtual environment is in effect, this value will be changed in *site.py* to point to the virtual environment. The value for the Python installation will still be available, via *base_prefix*.

**sys.ps1**
**sys.ps2**
Strings specifying the primary and secondary prompt of the interpreter. These are only defined if the

28.1. *sys* — System-specific parameters and functions 1223
The Python Library Reference, Release 3.3.3

The Python interpreter is in interactive mode. Their initial values in this case are ‘>>>’ and ‘...’. If a non-string object is assigned to either variable, its str() is re-evaluated each time the interpreter prepares to read a new interactive command; this can be used to implement a dynamic prompt.

sys.setcheckinterval(interval)
Set the interpreter’s “check interval”. This integer value determines how often the interpreter checks for periodic things such as thread switches and signal handlers. The default is 100, meaning the check is performed every 100 Python virtual instructions. Setting it to a larger value may increase performance for programs using threads. Setting it to a value <= 0 checks every virtual instruction, maximizing responsiveness as well as overhead. Deprecated since version 3.2: This function doesn’t have an effect anymore, as the internal logic for thread switching and asynchronous tasks has been rewritten. Use setswitchinterval() instead.

sys.setdlopenflags(n)
Set the flags used by the interpreter for dlopen() calls, such as when the interpreter loads extension modules. Among other things, this will enable a lazy resolving of symbols when importing a module, if called as sys.setdlopenflags(0). To share symbols across extension modules, call as sys.setdlopenflags(os.RTLD_GLOBAL). Symbolic names for the flag modules can be found in the os module (RTLD_xxx constants, e.g. os.RTLD_LAZY).

Availability: Unix.

sys.setprofile(profilefunc)
Set the system’s profile function, which allows you to implement a Python source code profiler in Python. See chapter The Python Profilers for more information on the Python profiler. The system’s profile function is called similarly to the system’s trace function (see settrace()), but it isn’t called for each executed line of code (only on call and return, but the return event is reported even when an exception has been set). The function is thread-specific, but there is no way for the profiler to know about context switches between threads, so it does not make sense to use this in the presence of multiple threads. Also, its return value is not used, so it can simply return None.

sys.setrecursionlimit(limit)
Set the maximum depth of the Python interpreter stack to limit. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python.

The highest possible limit is platform-dependent. A user may need to set the limit higher when they have a program that requires deep recursion and a platform that supports a higher limit. This should be done with care, because a too-high limit can lead to a crash.

sys.setswitchinterval(interval)
Set the interpreter’s thread switch interval (in seconds). This floating-point value determines the ideal duration of the “timeslices” allocated to concurrently running Python threads. Please note that the actual value can be higher, especially if long-running internal functions or methods are used. Also, which thread becomes scheduled at the end of the interval is the operating system’s decision. The interpreter doesn’t have its own scheduler. New in version 3.2.

sys.settrace(tracefunc)
Set the system’s trace function, which allows you to implement a Python source code debugger in Python. The function is thread-specific; for a debugger to support multiple threads, it must be registered using settrace() for each thread being debugged.

Trace functions should have three arguments: frame, event, and arg. frame is the current stack frame. event is a string: ‘call’, ‘line’, ‘return’, ‘exception’, ‘c_call’, ‘c_return’, or ‘c_exception’. arg depends on the event type.

The trace function is invoked (with event set to ‘call’) whenever a new local scope is entered; it should return a reference to a local trace function to be used that scope, or None if the scope shouldn’t be traced.

The local trace function should return a reference to itself (or to another function for further tracing in that scope), or None to turn off tracing in that scope.

The events have the following meaning:

‘call’ A function is called (or some other code block entered). The global trace function is called; arg is None; the return value specifies the local trace function.
'line' The interpreter is about to execute a new line of code or re-execute the condition of a loop. The local trace function is called; \texttt{arg} is \texttt{None}; the return value specifies the new local trace function. See \texttt{Objects/inotab_notes.txt} for a detailed explanation of how this works.

'\texttt{return}' A function (or other code block) is about to return. The local trace function is called; \texttt{arg} is the value that will be returned, or \texttt{None} if the event is caused by an exception being raised. The trace function’s return value is ignored.

'\texttt{exception}' An exception has occurred. The local trace function is called; \texttt{arg} is a tuple \texttt{(exception, value, traceback)}; the return value specifies the new local trace function.

'\texttt{c\_call}' A C function is about to be called. This may be an extension function or a built-in. \texttt{arg} is the C function object.

'\texttt{c\_return}' A C function has returned. \texttt{arg} is the C function object.

'\texttt{c\_exception}' A C function has raised an exception. \texttt{arg} is the C function object.

Note that as an exception is propagated down the chain of callers, an ‘\texttt{exception}’ event is generated at each level.

For more information on code and frame objects, refer to \texttt{types}.

\textbf{CPython implementation detail:} The \texttt{settrace()} function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

\texttt{sys.settscdump(on\_flag)}

Activate dumping of VM measurements using the Pentium timestamp counter, if \texttt{on\_flag} is true. Deactivate these dumps if \texttt{on\_flag} is off. The function is available only if Python was compiled with \texttt{--with-tsc}.

To understand the output of this dump, read \texttt{Python/ceval.c} in the Python sources.

\textbf{CPython implementation detail:} This function is intimately bound to CPython implementation details and thus not likely to be implemented elsewhere.

\texttt{sys.stdin} \quad \texttt{sys.stdout} \quad \texttt{sys.stderr}

\textit{File objects} used by the interpreter for standard input, output and errors:

- \texttt{stdin} is used for all interactive input (including calls to \texttt{input()});
- \texttt{stdout} is used for the output of \texttt{print()} and \texttt{expression} statements and for the prompts of \texttt{input()};
- The interpreter’s own prompts and its error messages go to \texttt{stderr}.

By default, these streams are regular text streams as returned by the \texttt{open()} function. Their parameters are chosen as follows:

- The character encoding is platform-dependent. Under Windows, if the stream is interactive (that is, if its \texttt{isatty()} method returns True), the console codepage is used, otherwise the ANSI code page. Under other platforms, the locale encoding is used (see \texttt{locale.getpreferredencoding()}).

  Under all platforms though, you can override this value by setting the \texttt{PYTHONIOENCODING} environment variable.

- When interactive, standard streams are line-buffered. Otherwise, they are block-buffered like regular text files. You can override this value with the \texttt{-u} command-line option.

To write or read binary data from/to the standard streams, use the underlying binary \texttt{buffer}. For example, to write bytes to \texttt{stdout}, use \texttt{sys.stdout.buffer.write(b’abc’)}. Using \texttt{io.TextIOWrapper.detach()}, streams can be made binary by default. This function sets \texttt{stdin} and \texttt{stdout} to binary:
```python
def make_streams_binary():
    sys.stdin = sys.stdin.detach()
    sys.stdout = sys.stdout.detach()
```

Note that the streams may be replaced with objects (like `io.StringIO`) that do not support
the `buffer` attribute or the `detach()` method and can raise `AttributeError` or
`io.UnsupportedOperation`.

```python
sys.__stdin__
sys.__stdout__
sys.__stderr__
```

These objects contain the original values of `stdin`, `stderr` and `stdout` at the start of the program.
They are used during finalization, and could be useful to print to the actual standard stream no matter if the
`sys.std*` object has been redirected.

It can also be used to restore the actual files to known working file objects in case they have been overwritten
with a broken object. However, the preferred way to do this is to explicitly save the previous stream before
replacing it, and restore the saved object.

**Note:** Under some conditions `stdin`, `stdout` and `stderr` as well as the original values `__stdin__`,
`__stdout__` and `__stderr__` can be `None`. It is usually the case for Windows GUI apps that aren’t
connected to a console and Python apps started with `pythonw`.

```python
sys.thread_info
```
A `struct sequence` holding information about the thread implementation.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| name      | Name of the thread implementation:
|           | • ‘nt’: Windows threads
|           | • ‘os2’: OS/2 threads
|           | • ‘pthread’: POSIX threads
|           | • ‘solaris’: Solaris threads
| lock      | Name of the lock implementation:
|           | • ‘semaphore’: a lock uses a semaphore
|           | • ‘mutex+cond’: a lock uses a mutex and a condition variable
|           | • `None` if this information is unknown
| version   | Name and version of the thread library. It is a string, or `None` if these
|           | informations are unknown.

New in version 3.3.

```python
sys.tracebacklimit
```
When this variable is set to an integer value, it determines the maximum number of levels of traceback
information printed when an unhandled exception occurs. The default is `1000`. When set to `0` or less, all
traceback information is suppressed and only the exception type and value are printed.

```python
sys.version
```
A string containing the version number of the Python interpreter plus additional information on the build
number and compiler used. This string is displayed when the interactive interpreter is started. Do not extract
version information out of it, rather, use `version_info` and the functions provided by the `platform`
module.

```python
sys.api_version
```
The C API version for this interpreter. Programmers may find this useful when debugging version conflicts
between Python and extension modules.

```python
sys.version_info
```
A tuple containing the five components of the version number: `major`, `minor`, `micro`, `releaselevel`, and
`serial`. All values except `releaselevel` are integers; the release level is ‘alpha’, ‘beta’, ‘candidate’,
or ‘final’. The version_info value corresponding to the Python version 2.0 is (2, 0, 0, ‘final’, 0). The components can also be accessed by name, so sys.version_info[0] is equivalent to sys.version_info.major and so on. Changed in version 3.1: Added named component attributes.

sys.warnoptions
This is an implementation detail of the warnings framework; do not modify this value. Refer to the warnings module for more information on the warnings framework.

sys.winver
The version number used to form registry keys on Windows platforms. This is stored as string resource 1000 in the Python DLL. The value is normally the first three characters of version. It is provided in the sys module for informational purposes; modifying this value has no effect on the registry keys used by Python. Availability: Windows.

sys._xoptions
A dictionary of the various implementation-specific flags passed through the -X command-line option. Option names are either mapped to their values, if given explicitly, or to True. Example:

```bash
$ ./python -Xa=b -Xc
Python 3.2a3+ (py3k, Oct 16 2010, 20:14:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys._xoptions
{'a': 'b', 'c': True}
```

CPython implementation detail: This is a CPython-specific way of accessing options passed through -X. Other implementations may export them through other means, or not at all. New in version 3.2.

Citations

28.2 sysconfig — Provide access to Python’s configuration information

New in version 3.2. Source code: Lib/sysconfig.py

The sysconfig module provides access to Python’s configuration information like the list of installation paths and the configuration variables relevant for the current platform.

28.2.1 Configuration variables

A Python distribution contains a Makefile and a pyconfig.h header file that are necessary to build both the Python binary itself and third-party C extensions compiled using distutils.

sysconfig puts all variables found in these files in a dictionary that can be accessed using get_config_vars() or get_config_var().

Notice that on Windows, it’s a much smaller set.

```python
sysconfig.get_config_vars(*args)
```

With no arguments, return a dictionary of all configuration variables relevant for the current platform.

With arguments, return a list of values that result from looking up each argument in the configuration variable dictionary.

For each argument, if the value is not found, return None.
sysconfig.get_config_var(name)

Return the value of a single variable name. Equivalent to get_config_vars().get(name).

If name is not found, return None.

Example of usage:

>>> import sysconfig
>>> sysconfig.get_config_var('Py_ENABLE_SHARED')
0
>>> sysconfig.get_config_var('LIBDIR')
'/usr/local/lib'
>>> sysconfig.get_config_vars('AR', 'CXX')
['ar', 'g++']

28.2.2 Installation paths

Python uses an installation scheme that differs depending on the platform and on the installation options. These schemes are stored in sysconfig under unique identifiers based on the value returned by os.name.

Every new component that is installed using distutils or a Distutils-based system will follow the same scheme to copy its file in the right places.

Python currently supports seven schemes:

- **posix_prefix**: scheme for Posix platforms like Linux or Mac OS X. This is the default scheme used when Python or a component is installed.
- **posix_home**: scheme for Posix platforms used when a home option is used upon installation. This scheme is used when a component is installed through Distutils with a specific home prefix.
- **posix_user**: scheme for Posix platforms used when a component is installed through Distutils and the user option is used. This scheme defines paths located under the user home directory.
- **nt**: scheme for NT platforms like Windows.
- **nt_user**: scheme for NT platforms, when the user option is used.
- **os2**: scheme for OS/2 platforms.
- **os2_home**: scheme for OS/2 platforms, when the user option is used.

Each scheme is itself composed of a series of paths and each path has a unique identifier. Python currently uses eight paths:

- **stdlib**: directory containing the standard Python library files that are not platform-specific.
- **platstdlib**: directory containing the standard Python library files that are platform-specific.
- **platlib**: directory for site-specific, platform-specific files.
- **purelib**: directory for site-specific, non-platform-specific files.
- **include**: directory for non-platform-specific header files.
- **platinclude**: directory for platform-specific header files.
- **scripts**: directory for script files.
- **data**: directory for data files.

sysconfig provides some functions to determine these paths.

sysconfig.get_scheme_names()

Return a tuple containing all schemes currently supported in sysconfig.

sysconfig.get_path_names()

Return a tuple containing all path names currently supported in sysconfig.
sysconfig.get_path(name[, scheme[, vars[, expand]]]])
Return an installation path corresponding to the path name, from the install scheme named scheme.

name has to be a value from the list returned by get_path_names().
sysconfig stores installation paths corresponding to each path name, for each platform, with variables to be expanded. For instance the stdlib path for the nt scheme is: {base}/Lib.
get_path() will use the variables returned by get_config_vars() to expand the path. All variables have default values for each platform so one may call this function and get the default value.

If scheme is provided, it must be a value from the list returned by get_scheme_names(). Otherwise, the default scheme for the current platform is used.

If vars is provided, it must be a dictionary of variables that will update the dictionary return by get_config_vars().

If expand is set to False, the path will not be expanded using the variables.

If name is not found, return None.

sysconfig.get_paths([scheme[, vars[, expand]]]])
Return a dictionary containing all installation paths corresponding to an installation scheme. See get_path() for more information.

If scheme is not provided, will use the default scheme for the current platform.

If vars is provided, it must be a dictionary of variables that will update the dictionary used to expand the paths.

If expand is set to False, the paths will not be expanded.

If scheme is not an existing scheme, get_paths() will raise a KeyError.

28.2.3 Other functions

sysconfig.get_python_version()
Return the MAJOR.MINOR Python version number as a string. Similar to sys.version[:3].

sysconfig.get_platform()
Return a string that identifies the current platform.

This is used mainly to distinguish platform-specific build directories and platform-specific built distributions. Typically includes the OS name and version and the architecture (as supplied by os.uname()), although the exact information included depends on the OS; e.g. for IRIX the architecture isn’t particularly important (IRIX only runs on SGI hardware), but for Linux the kernel version isn’t particularly important.

Examples of returned values:
- linux-i586
- linux-alpha (?)
- solaris-2.6-sun4u
- irix-5.3
- irix64-6.2

Windows will return one of:
- win-amd64 (64bit Windows on AMD64 (aka x86_64, Intel64, EM64T, etc)
- win-ia64 (64bit Windows on Itanium)
- win32 (all others - specifically, sys.platform is returned)

Mac OS X can return:
- macosx-10.6-ppc
For other non-POSIX platforms, currently just returns `sys.platform`.

**sysconfig.**

*.is_python_build()*

Return True if the current Python installation was built from source.

**sysconfig.**

*.parse_config_h(fp[, vars])*

Parse a config.h-style file.

*fp* is a file-like object pointing to the config.h-like file.

A dictionary containing name/value pairs is returned. If an optional dictionary is passed in as the second argument, it is used instead of a new dictionary, and updated with the values read in the file.

**sysconfig.**

*.get_config_h_filename()*

Return the path of pyconfig.h.

**sysconfig.**

*.get_makefile_filename()*

Return the path of Makefile.

### 28.2.4 Using sysconfig as a script

You can use `sysconfig` as a script with Python’s `-m` option:

```bash
$ python -m sysconfig
Platform: "macosx-10.4-i386"
Python version: "3.2"
Current installation scheme: "posix_prefix"
```

Paths:

```text
data = "/usr/local"
include = "/Users/tarek/Dev/svn.python.org/py3k/Include"
platinclude = "." 
platlib = "/usr/local/lib/python3.2/site-packages"
platstdlib = "/usr/local/lib/python3.2"
purelib = "/usr/local/lib/python3.2/site-packages"
scripts = "/usr/local/bin"
stdlib = "/usr/local/lib/python3.2"
```

Variables:

- AC_APPLE_UNIVERSAL_BUILD = "0"
- AIX_GENUINE_CPLUSPLUS = "0"
- AR = "ar"
- ARFLAGS = "rc"
- ASDLGEN = "./Parser/asdl_c.py"
- ...

This call will print in the standard output the information returned by `get_platform()`, `get_python_version()`, `get_path()` and `get_config_vars()`.

### 28.3 builtins — Built-in objects

This module provides direct access to all ‘built-in’ identifiers of Python; for example, `builtins.open` is the full name for the built-in function `open()`. See `Built-in Functions` and `Built-in Constants` for documentation.

This module is not normally accessed explicitly by most applications, but can be useful in modules that provide objects with the same name as a built-in value, but in which the built-in of that name is also needed. For example,
in a module that wants to implement an `open()` function that wraps the built-in `open()`, this module can be used directly:

```python
import builtins

def open(path):
    f = builtins.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''Wrapper around a file that converts output to upper-case.''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()
```

As an implementation detail, most modules have the name `__builtins__` made available as part of their globals. The value of `__builtins__` is normally either this module or the value of this module's `__dict__` attribute. Since this is an implementation detail, it may not be used by alternate implementations of Python.

### 28.4 __main__ — Top-level script environment

This module represents the (otherwise anonymous) scope in which the interpreter’s main program executes — commands read either from standard input, from a script file, or from an interactive prompt. It is this environment in which the idiomatic “conditional script” stanza causes a script to run:

```python
if __name__ == '__main__':
    main()
```

### 28.5 warnings — Warning control

**Source code:** Lib/warnings.py

Warning messages are typically issued in situations where it is useful to alert the user of some condition in a program, where that condition (normally) doesn’t warrant raising an exception and terminating the program. For example, one might want to issue a warning when a program uses an obsolete module.

Python programmers issue warnings by calling the `warn()` function defined in this module. (C programmers use `PyErr_WarnEx()`; see `exceptionhandling` for details).

Warning messages are normally written to `sys.stderr`, but their disposition can be changed flexibly, from ignoring all warnings to turning them into exceptions. The disposition of warnings can vary based on the warning category (see below), the text of the warning message, and the source location where it is issued. Repetitions of a particular warning for the same source location are typically suppressed.

There are two stages in warning control: first, each time a warning is issued, a determination is made whether a message should be issued or not; next, if a message is to be issued, it is formatted and printed using a user-settable hook.

The determination whether to issue a warning message is controlled by the warning filter, which is a sequence of matching rules and actions. Rules can be added to the filter by calling `filterwarnings()` and reset to its default state by calling `resetwarnings()`.
The printing of warning messages is done by calling \texttt{showwarning()}, which may be overridden; the default implementation of this function formats the message by calling \texttt{formatwarning()}, which is also available for use by custom implementations.

\textbf{See Also:}

\texttt{logging.captureWarnings()} allows you to handle all warnings with the standard logging infrastructure.

\subsection*{28.5.1 Warning Categories}

There are a number of built-in exceptions that represent warning categories. This categorization is useful to be able to filter out groups of warnings. The following warnings category classes are currently defined:

\begin{table}[h]
\begin{tabular}{|l|l|}
\hline
Class & Description \\
\hline
Warning & This is the base class of all warning category classes. It is a subclass of \texttt{Exception}. \\
UserWarning & The default category for \texttt{warn()}. \\
DeprecationWarning & Base category for warnings about deprecated features (ignored by default). \\
SyntaxWarning & Base category for warnings about dubious syntactic features. \\
RuntimeWarning & Base category for warnings about dubious runtime features. \\
FutureWarning & Base category for warnings about constructs that will change semantically in the future. \\
PendingDeprecationWarning & Base category for warnings about features that will be deprecated in the future (ignored by default). \\
ImportWarning & Base category for warnings triggered during the process of importing a module (ignored by default). \\
UnicodeWarning & Base category for warnings related to Unicode. \\
BytesWarning & Base category for warnings related to \texttt{bytes} and \texttt{bytearray}. \\
ResourceWarning & Base category for warnings related to resource usage. \\
\hline
\end{tabular}
\end{table}

While these are technically built-in exceptions, they are documented here, because conceptually they belong to the warnings mechanism.

User code can define additional warning categories by subclassing one of the standard warning categories. A warning category must always be a subclass of the \texttt{Warning} class.

\subsection*{28.5.2 The Warnings Filter}

The warnings filter controls whether warnings are ignored, displayed, or turned into errors (raising an exception).

Conceptually, the warnings filter maintains an ordered list of filter specifications; any specific warning is matched against each filter specification in the list in turn until a match is found; the match determines the disposition of the match. Each entry is a tuple of the form (\texttt{action}, \texttt{message}, \texttt{category}, \texttt{module}, \texttt{lineno}), where:

- \texttt{action} is one of the following strings:

\begin{table}[h]
\begin{tabular}{|l|l|}
\hline
Value & Disposition \\
\hline
"error" & turn matching warnings into exceptions \\
"ignore" & never print matching warnings \\
"always" & always print matching warnings \\
"default" & print the first occurrence of matching warnings for each location where the warning is issued \\
"module" & print the first occurrence of matching warnings for each module where the warning is issued \\
"once" & print only the first occurrence of matching warnings, regardless of location \\
\hline
\end{tabular}
\end{table}

- \texttt{message} is a string containing a regular expression that the warning message must match (the match is compiled to always be case-insensitive).

- \texttt{category} is a class (a subclass of \texttt{Warning}) of which the warning category must be a subclass in order to match.
• **module** is a string containing a regular expression that the module name must match (the match is compiled to be case-sensitive).

• **lineno** is an integer that the line number where the warning occurred must match, or 0 to match all line numbers.

Since the `Warning` class is derived from the built-in `Exception` class, to turn a warning into an error we simply raise `category(message)`.

The warnings filter is initialized by `-W` options passed to the Python interpreter command line. The interpreter saves the arguments for all `-W` options without interpretation in `sys.warnoptions`; the `warnings` module parses these when it is first imported (invalid options are ignored, after printing a message to `sys.stderr`).

## Default Warning Filters

By default, Python installs several warning filters, which can be overridden by the command-line options passed to `-W` and calls to `filterwarnings()`.

- `DeprecationWarning` and `PendingDeprecationWarning`, and `ImportWarning` are ignored.
- `BytesWarning` is ignored unless the `-b` option is given once or twice; in this case this warning is either printed (`-b`) or turned into an exception (`-bb`).
- `ResourceWarning` is ignored unless Python was built in debug mode.

Changed in version 3.2: `DeprecationWarning` is now ignored by default in addition to `PendingDeprecationWarning`.

### 28.5.3 Temporarily Suppressing Warnings

If you are using code that you know will raise a warning, such as a deprecated function, but do not want to see the warning, then it is possible to suppress the warning using the `catch_warnings` context manager:

```python
import warnings
def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings()
    warnings.simplefilter("ignore")
    fxn()
```

While within the context manager all warnings will simply be ignored. This allows you to use known-deprecated code without having to see the warning while not suppressing the warning for other code that might not be aware of its use of deprecated code. Note: this can only be guaranteed in a single-threaded application. If two or more threads use the `catch_warnings` context manager at the same time, the behavior is undefined.

### 28.5.4 Testing Warnings

To test warnings raised by code, use the `catch_warnings` context manager. With it you can temporarily mutate the warnings filter to facilitate your testing. For instance, do the following to capture all raised warnings to check:

```python
import warnings
def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
    # Cause all warnings to always be triggered.
    warnings.simplefilter("always")
    # Trigger a warning.
```
fxn()
# Verify some things
assert len(w) == 1
assert issubclass(w[-1].category, DeprecationWarning)
assert "deprecated" in str(w[-1].message)

One can also cause all warnings to be exceptions by using `error` instead of `always`. One thing to be aware of is that if a warning has already been raised because of a `once/default` rule, then no matter what filters are set the warning will not be seen again unless the warnings registry related to the warning has been cleared.

Once the context manager exits, the warnings filter is restored to its state when the context was entered. This prevents tests from changing the warnings filter in unexpected ways between tests and leading to indeterminate test results. The `showwarning()` function in the module is also restored to its original value. Note: this can only be guaranteed in a single-threaded application. If two or more threads use the `catch_warnings` context manager at the same time, the behavior is undefined.

When testing multiple operations that raise the same kind of warning, it is important to test them in a manner that confirms each operation is raising a new warning (e.g. set warnings to be raised as exceptions and check the operations raise exceptions, check that the length of the warning list continues to increase after each operation, or else delete the previous entries from the warnings list before each new operation).

### 28.5.5 Updating Code For New Versions of Python

Warnings that are only of interest to the developer are ignored by default. As such you should make sure to test your code with typically ignored warnings made visible. You can do this from the command-line by passing `-Wd` to the interpreter (this is shorthand for `^- ^default`). This enables default handling for all warnings, including those that are ignored by default. To change what action is taken for encountered warnings you simply change what argument is passed to `-W`, e.g. `^- ^error`. See the `-W` flag for more details on what is possible.

To programmatically do the same as `-Wd`, use:

```
warnings.simplefilter('default')
```

Make sure to execute this code as soon as possible. This prevents the registering of what warnings have been raised from unexpectedly influencing how future warnings are treated.

Having certain warnings ignored by default is done to prevent a user from seeing warnings that are only of interest to the developer. As you do not necessarily have control over what interpreter a user uses to run their code, it is possible that a new version of Python will be released between your release cycles. The new interpreter release could trigger new warnings in your code that were not there in an older interpreter, e.g. `DeprecationWarning` for a module that you are using. While you as a developer want to be notified that your code is using a deprecated module, to a user this information is essentially noise and provides no benefit to them.

The `unittest` module has been also updated to use the `default` filter while running tests.

### 28.5.6 Available Functions

```
warnings.warn(message, category=None, stacklevel=1)
```

Issue a warning, or maybe ignore it or raise an exception. The `category` argument, if given, must be a warning category class (see above); it defaults to `UserWarning`. Alternatively `message` can be a `Warning` instance, in which case `category` will be ignored and `message.__class__` will be used. In this case the message text will be `str(message)`. This function raises an exception if the particular warning issued is changed into an error by the warnings filter see above. The `stacklevel` argument can be used by wrapper functions written in Python, like this:

```
def deprecation(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

This makes the warning refer to `deprecation()`'s caller, rather than to the source of `deprecation()` itself (since the latter would defeat the purpose of the warning message).
**warnings.warn_explicit**(message, category, filename, lineno, module=None, registry=None, module_globals=None)

This is a low-level interface to the functionality of warn(), passing in explicitly the message, category, filename and line number, and optionally the module name and the registry (which should be the __warningregistry__ dictionary of the module). The module name defaults to the filename with .py stripped; if no registry is passed, the warning is never suppressed. message must be a string and category a subclass of Warning or message may be a Warning instance, in which case category will be ignored.

module_globals, if supplied, should be the global namespace in use by the code for which the warning is issued. (This argument is used to support displaying source for modules found in zipfiles or other non-filesystem import sources).

**warnings.showwarning**(message, category, filename, lineno, file=None, line=None)

Write a warning to a file. The default implementation calls formatwarning(message, category, filename, lineno, line) and writes the resulting string to file, which defaults to sys.stderr. You may replace this function with any callable by assigning to warnings.showwarning. line is a line of source code to be included in the warning message; if line is not supplied, showwarning() will try to read the line specified by filename and lineno.

**warnings.formatwarning**(message, category, filename, lineno, line=None)

Format a warning the standard way. This returns a string which may contain embedded newlines and ends in a newline. line is a line of source code to be included in the warning message; if line is not supplied, formatwarning() will try to read the line specified by filename and lineno.

**warnings.filterwarnings**(action, message=’,’, category=Warning, module=’,’, lineno=0, append=False)

Insert an entry into the list of warnings filter specifications. The entry is inserted at the front by default; if append is true, it is inserted at the end. This checks the types of the arguments, compiles the message and module regular expressions, and inserts them as a tuple in the list of warnings filters. Entries closer to the front of the list override entries later in the list, if both match a particular warning. Omitted arguments default to a value that matches everything.

**warnings.simplefilter**(action, category=Warning, lineno=0, append=False)

Insert a simple entry into the list of warnings filter specifications. The meaning of the function parameters is as for filterwarnings(), but regular expressions are not needed as the filter inserted always matches any message in any module as long as the category and line number match.

**warnings.resetwarnings()**

Reset the warnings filter. This discards the effect of all previous calls to filterwarnings(), including that of the -W command line options and calls to simplefilter().

### 28.5.7 Available Context Managers

**class warnings.catch_warnings**(*, record=False, module=None)

A context manager that copies and, upon exit, restores the warnings filter and the showwarning() function. If the record argument is False (the default) the context manager returns None on entry. If record is True, a list is returned that is progressively populated with objects as seen by a custom showwarning() function (which also suppresses output to sys.stdout). Each object in the list has attributes with the same names as the arguments to showwarning().

The module argument takes a module that will be used instead of the module returned when you import warnings whose filter will be protected. This argument exists primarily for testing the warnings module itself.

**Note:** The catch_warnings manager works by replacing and then later restoring the module’s showwarning() function and internal list of filter specifications. This means the context manager is modifying global state and therefore is not thread-safe.
This module provides utilities for common tasks involving the `with` statement. For more information see also `Context Manager Types` and `context-managers`.

### 28.6.1 Utilities

Functions and classes provided:

`@contextlib.contextmanager`

This function is a decorator that can be used to define a factory function for `with` statement context managers, without needing to create a class or separate `__enter__()` and `__exit__()` methods.

A simple example (this is not recommended as a real way of generating HTML!):

```python
from contextlib import contextmanager

@contextmanager
def tag(name):
    print("<%s>" % name)
    yield
    print("</%s>" % name)

>>> with tag("h1"):
...    print("foo")
...<h1>
foo</h1>
```

The function being decorated must return a generator-iterator when called. This iterator must yield exactly one value, which will be bound to the targets in the `with` statement's `as` clause, if any.

At the point where the generator yields, the block nested in the `with` statement is executed. The generator is then resumed after the block is exited. If an unhandled exception occurs in the block, it is reraised inside the generator at the point where the yield occurred. Thus, you can use a `try...except...finally` statement to trap the error (if any), or ensure that some cleanup takes place. If an exception is trapped merely in order to log it or to perform some action (rather than to suppress it entirely), the generator must reraise that exception. Otherwise the generator context manager will indicate to the `with` statement that the exception has been handled, and execution will resume with the statement immediately following the `with` statement.

`contextmanager()` uses `ContextDecorator` so the context managers it creates can be used as decorators as well as in `with` statements. When used as a decorator, a new generator instance is implicitly created on each function call (this allows the otherwise “one-shot” context managers created by `contextmanager()` to meet the requirement that context managers support multiple invocations in order to be used as decorators). Changed in version 3.2: Use of `ContextDecorator`.

`contextlib.closing(thing)`

Return a context manager that closes `thing` upon completion of the block. This is basically equivalent to:

```python
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
```
And lets you write code like this:

```python
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('http://www.python.org')) as page:
    for line in page:
        print(line)
```

without needing to explicitly close `page`. Even if an error occurs, `page.close()` will be called when the `with` block is exited.

### class `contextlib.ContextDecorator`

A base class that enables a context manager to also be used as a decorator.

Context managers inheriting from `ContextDecorator` have to implement `__enter__` and `__exit__` as normal. `__exit__` retains its optional exception handling even when used as a decorator.

`ContextDecorator` is used by `contextmanager()`, so you get this functionality automatically.

Example of `ContextDecorator`:

```python
from contextlib import ContextDecorator

class mycontext(ContextDecorator):
    def __enter__(self):
        print('Starting')
        return self

    def __exit__(self, *exc):
        print('Finishing')
        return False

>>> @mycontext()
... def function():
...     print('The bit in the middle')
... >>> function()
Starting
The bit in the middle
Finishing
```

This change is just syntactic sugar for any construct of the following form:

```python
def f():
    with cm():
        # Do stuff
```
@cm()
def f():
    # Do stuff

It makes it clear that the cm applies to the whole function, rather than just a piece of it (and saving an indentation level is nice, too).

Existing context managers that already have a base class can be extended by using ContextDecorator as a mixin class:

```python
from contextlib import ContextDecorator
class mycontext(ContextBaseClass, ContextDecorator):
    def __enter__(self):
        return self
    def __exit__(self, *exc):
        return False
```

**Note:** As the decorated function must be able to be called multiple times, the underlying context manager must support use in multiple `with` statements. If this is not the case, then the original construct with the explicit `with` statement inside the function should be used.

New in version 3.2.

```python
class contextlib.ExitStack
A context manager that is designed to make it easy to programmatically combine other context managers and cleanup functions, especially those that are optional or otherwise driven by input data.

For example, a set of files may easily be handled in a single `with` statement as follows:

```python
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
# All opened files will automatically be closed at the end of
# the with statement, even if attempts to open files later
# in the list raise an exception
```

Each instance maintains a stack of registered callbacks that are called in reverse order when the instance is closed (either explicitly or implicitly at the end of a `with` statement). Note that callbacks are not invoked implicitly when the context stack instance is garbage collected.

This stack model is used so that context managers that acquire their resources in their `__init__` method (such as file objects) can be handled correctly.

Since registered callbacks are invoked in the reverse order of registration, this ends up behaving as if multiple nested `with` statements had been used with the registered set of callbacks. This even extends to exception handling - if an inner callback suppresses or replaces an exception, then outer callbacks will be passed arguments based on that updated state.

This is a relatively low level API that takes care of the details of correctly unwinding the stack of exit callbacks. It provides a suitable foundation for higher level context managers that manipulate the exit stack in application specific ways. New in version 3.3.

```python
enter_context(cm)
```

Enters a new context manager and adds its `__exit__()` method to the callback stack. The return value is the result of the context manager’s own `__enter__()` method.

These context managers may suppress exceptions just as they normally would if used directly as part of a `with` statement.
**push**(exit)

Adds a context manager’s **__exit__()** method to the callback stack.

As **__enter__** is not invoked, this method can be used to cover part of an **__enter__()** implementation with a context manager’s own **__exit__()** method.

If passed an object that is not a context manager, this method assumes it is a callback with the same signature as a context manager’s **__exit__()** method and adds it directly to the callback stack.

By returning true values, these callbacks can suppress exceptions the same way context manager **__exit__()** methods can.

The passed in object is returned from the function, allowing this method to be used as a function decorator.

**callback**(callback, *args, **kwds)

Accepts an arbitrary callback function and arguments and adds it to the callback stack.

Unlike the other methods, callbacks added this way cannot suppress exceptions (as they are never passed the exception details).

The passed in callback is returned from the function, allowing this method to be used as a function decorator.

**pop_all()**

Transfers the callback stack to a fresh ExitStack instance and returns it. No callbacks are invoked by this operation - instead, they will now be invoked when the new stack is closed (either explicitly or implicitly at the end of a with statement).

For example, a group of files can be opened as an “all or nothing” operation as follows:

```python
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # Hold onto the close method, but don’t call it yet.
    close_files = stack.pop_all().close
    # If opening any file fails, all previously opened files will be closed automatically. If all files are opened successfully, they will remain open even after the with statement ends. # close_files() can then be invoked explicitly to close them all.
```

**close()**

Immediately unwinds the callback stack, invoking callbacks in the reverse order of registration. For any context managers and exit callbacks registered, the arguments passed in will indicate that no exception occurred.

### 28.6.2 Examples and Recipes

This section describes some examples and recipes for making effective use of the tools provided by contextlib.

#### Supporting a variable number of context managers

The primary use case for ExitStack is the one given in the class documentation: supporting a variable number of context managers and other cleanup operations in a single with statement. The variability may come from the number of context managers needed being driven by user input (such as opening a user specified collection of files), or from some of the context managers being optional:

```python
with ExitStack() as stack:
    for resource in resources:
        stack.enter_context(resource)
    if need_special_resource:
        special = acquire_special_resource()
```
stack.callback(release_special_resource, special)
# Perform operations that use the acquired resources

As shown, ExitStack also makes it quite easy to use with statements to manage arbitrary resources that don’t natively support the context management protocol.

Simplifying support for single optional context managers

In the specific case of a single optional context manager, ExitStack instances can be used as a “do nothing” context manager, allowing a context manager to easily be omitted without affecting the overall structure of the source code:

def debug_trace(details):
    if __debug__:
        return
        # Don’t do anything special with the context in release mode
    return ExitStack()

with debug_trace():
    # Suite is traced in debug mode, but runs normally otherwise

Catching exceptions from __enter__ methods

It is occasionally desirable to catch exceptions from an __enter__ method implementation, without inadvertently catching exceptions from the with statement body or the context manager’s __exit__ method. By using ExitStack the steps in the context management protocol can be separated slightly in order to allow this:

    stack = ExitStack()
try:
    x = stack.enter_context(cm)
except Exception:
    # handle __enter__ exception
else:
    with stack:
        # Handle normal case

Actually needing to do this is likely to indicate that the underlying API should be providing a direct resource management interface for use with try/except/finally statements, but not all APIs are well designed in that regard. When a context manager is the only resource management API provided, then ExitStack can make it easier to handle various situations that can’t be handled directly in a with statement.

Cleaning up in an __enter__ implementation

As noted in the documentation of ExitStack.push(), this method can be useful in cleaning up an already allocated resource if later steps in the __enter__() implementation fail.

Here’s an example of doing this for a context manager that accepts resource acquisition and release functions, along with an optional validation function, and maps them to the context management protocol:

    from contextlib import contextmanager, ExitStack

class ResourceManager:
    def __init__(self, acquire_resource, release_resource, check_resource_ok=None):
        self.acquire_resource = acquire_resource
        self.release_resource = release_resource
        if check_resource_ok is None:
            def check_resource_ok(resource):
                return True
self.check_resource_ok = check_resource_ok

@contextmanager
def _cleanup_on_error(self):
    with ExitStack() as stack:
        stack.push(self)
        yield
        # The validation check passed and didn’t raise an exception
        # Accordingly, we want to keep the resource, and pass it
        # back to our caller
        stack.pop_all()

def __enter__(self):
    resource = self.acquire_resource()
    with self._cleanup_on_error():
        if not self.check_resource_ok(resource):
            msg = "Failed validation for {!r}"
            raise RuntimeError(msg.format(resource))
    return resource

def __exit__(self, *exc_details):
    # We don’t need to duplicate any of our resource release logic
    self.release_resource()

Replacing any use of try-finally and flag variables

A pattern you will sometimes see is a try-finally statement with a flag variable to indicate whether or not
the body of the finally clause should be executed. In its simplest form (that can’t already be handled just by
using an except clause instead), it looks something like this:

cleanup_needed = True
try:
    result = perform_operation()
    if result:
        cleanup_needed = False
finally:
    if cleanup_needed:
        cleanup_resources()

As with any try statement based code, this can cause problems for development and review, because the setup
code and the cleanup code can end up being separated by arbitrarily long sections of code.

ExitStack makes it possible to instead register a callback for execution at the end of a with statement, and
then later decide to skip executing that callback:

from contextlib import ExitStack

with ExitStack() as stack:
    stack.callback(cleanup_resources)
    result = perform_operation()
    if result:
        stack.pop_all()

This allows the intended cleanup up behaviour to be made explicit up front, rather than requiring a separate flag
variable.

If a particular application uses this pattern a lot, it can be simplified even further by means of a small helper class:

from contextlib import ExitStack

class Callback(ExitStack):

def __init__(self, callback, *args, **kwds):
    super(Callback, self).__init__()
    self.callback(callback, *args, **kwds)

def cancel(self):
    self.pop_all()

with Callback(cleanup_resources) as cb:
    result = perform_operation()
    if result:
        cb.cancel()

If the resource cleanup isn’t already neatly bundled into a standalone function, then it is still possible to use the
decorator form of ExitStack.callback() to declare the resource cleanup in advance:

from contextlib import ExitStack

with ExitStack() as stack:
    @stack.callback
    def cleanup_resources():
        ...
    result = perform_operation()
    if result:
        stack.pop_all()

Due to the way the decorator protocol works, a callback function declared this way cannot take any parameters.
Instead, any resources to be released must be accessed as closure variables

Using a context manager as a function decorator

ContextDecorator makes it possible to use a context manager in both an ordinary with statement and also
as a function decorator.

For example, it is sometimes useful to wrap functions or groups of statements with a logger that can track the
time of entry and time of exit. Rather than writing both a function decorator and a context manager for the task,
inheriting from ContextDecorator provides both capabilities in a single definition:

from contextlib import ContextDecorator
import logging

logging.basicConfig(level=logging.INFO)

class track_entry_and_exit(ContextDecorator):
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        logging.info('Entering: {}'.format(self.name))

    def __exit__(self, exc_type, exc, exc_tb):
        logging.info('Exiting: {}'.format(self.name))

Instances of this class can be used as both a context manager:

with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()

And also as a function decorator:

@track_entry_and_exit('widget loader')
def activity():
print('Some time consuming activity goes here')
load_widget()

Note that there is one additional limitation when using context managers as function decorators: there’s no way to access the return value of \_enter\_(\_\_). If that value is needed, then it is still necessary to use an explicit with statement.

See Also:

PEP 0343 - The “with” statement The specification, background, and examples for the Python with statement.

28.7 abc — Abstract Base Classes

Source code: Lib/abc.py

This module provides the infrastructure for defining abstract base classes (ABCs) in Python, as outlined in PEP 3119; see the PEP for why this was added to Python. (See also PEP 3141 and the numbers module regarding a type hierarchy for numbers based on ABCs.)

The collections module has some concrete classes that derive from ABCs; these can, of course, be further derived. In addition the collections.abc submodule has some ABCs that can be used to test whether a class or instance provides a particular interface, for example, is it hashable or a mapping.

This module provides the following class:

class abc.ABCMeta

Metaclass for defining Abstract Base Classes (ABCs).

Use this metaclass to create an ABC. An ABC can be subclassed directly, and then acts as a mix-in class. You can also register unrelated concrete classes (even built-in classes) and unrelated ABCs as “virtual subclasses” – these and their descendants will be considered subclasses of the registering ABC by the built-in \_issubclass\_(\_\_) function, but the registering ABC won’t show up in their MRO (Method Resolution Order) nor will method implementations defined by the registering ABC be callable (not even via super\_\_).\_\_.

Classes created with a metaclass of ABCMeta have the following method:

\_register\_(subclass)

Register subclass as a “virtual subclass” of this ABC. For example:

\from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass

MyABC.register(tuple)

assert \_issubclass\_(tuple, MyABC)
assert \_isinstance\_((), MyABC)

Changed in version 3.3: Returns the registered subclass, to allow usage as a class decorator.

You can also override this method in an abstract base class:

\_\_subclasshook\_\_(subclass)

(Must be defined as a class method.)

Check whether subclass is considered a subclass of this ABC. This means that you can customize the behavior of \_issubclass\_ further without the need to call register\_(\_\_) on every class you want

\^1 C++ programmers should note that Python’s virtual base class concept is not the same as C++’s.
to consider a subclass of the ABC. (This class method is called from the __subclasscheck__() method of the ABC.)

This method should return True, False or NotImplemented. If it returns True, the subclass is considered a subclass of this ABC. If it returns False, the subclass is not considered a subclass of this ABC, even if it would normally be one. If it returns NotImplemented, the subclass check is continued with the usual mechanism.

For a demonstration of these concepts, look at this example ABC definition:

```python
class Foo:
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
        return iter(self)

class MyIterable(metaclass=ABCMeta):
    @abstractmethod
    def __iter__(self):
        while False:
            yield None
    def get_iterator(self):
        return self.__iter__()

    @classmethod
    def __subclasshook__(cls, C):
        if cls is MyIterable:
            if any("__iter__" in B.__dict__ for B in C.__mro__):
                return True
            return NotImplemented

MyIterable.register(Foo)
```

The ABC MyIterable defines the standard iterable method, __iter__(), as an abstract method. The implementation given here can still be called from subclasses. The get_iterator() method is also part of the MyIterable abstract base class, but it does not have to be overridden in non-abstract derived classes.

The __subclasshook__() class method defined here says that any class that has an __iter__() method in its __dict__ (or in that of one of its base classes, accessed via the __mro__ list) is considered a MyIterable too.

Finally, the last line makes Foo a virtual subclass of MyIterable, even though it does not define an __iter__() method (it uses the old-style iterable protocol, defined in terms of __len__() and __getitem__()). Note that this will not make get_iterator available as a method of Foo, so it is provided separately.

The abc module also provides the following decorators:

@abc.abstractmethod
A decorator indicating abstract methods.

Using this decorator requires that the class’s metaclass is ABCMeta or is derived from it. A class that has a metaclass derived from ABCMeta cannot be instantiated unless all of its abstract methods and properties are overridden. The abstract methods can be called using any of the normal ‘super’ call mechanisms. abstractmethod() may be used to declare abstract methods for properties and descriptors.
Dynamically adding abstract methods to a class, or attempting to modify the abstraction status of a method or class once it is created, are not supported. The `abstractmethod()` only affects subclasses derived using regular inheritance; “virtual subclasses” registered with the ABC’s `register()` method are not affected.

When `abstractmethod()` is applied in combination with other method descriptors, it should be applied as the innermost decorator, as shown in the following usage examples:

```python
class C(metaclass=ABCMeta):
    @abstractmethod
    def my_abstract_method(self, ...):
        ...

    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, ...):
        ...

    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(...):
        ...

    @property
    @abstractmethod
    def my_abstract_property(self):
        ...

    @my_abstract_property.setter
    @abstractmethod
    def my_abstract_property(self, val):
        ...

    @abstractmethod
    def _get_x(self):
        ...

    @abstractmethod
    def _set_x(self, val):
        ...

    x = property(_get_x, _set_x)
```

In order to correctly interoperate with the abstract base class machinery, the descriptor must identify itself as abstract using `__isabstractmethod__`. In general, this attribute should be `True` if any of the methods used to compose the descriptor are abstract. For example, Python’s built-in property does the equivalent of:

```python
class Descriptor:
    ...

    @property
    def __isabstractmethod__(self):
        return any(getattr(f, '__isabstractmethod__', False) for f in (self._fget, self._fset, self._fdel))
```

**Note:** Unlike Java abstract methods, these abstract methods may have an implementation. This implementation can be called via the `super()` mechanism from the class that overrides it. This could be useful as an end-point for a super-call in a framework that uses cooperative multiple-inheritance.

```python
@abc.abstractclassmethod
A subclass of the built-in `classmethod()`, indicating an abstract classmethod. Otherwise it is similar to `abstractmethod()`.
```
This special case is deprecated, as the `classmethod()` decorator is now correctly identified as abstract when applied to an abstract method:

```python
class C(metaclass=ABCMeta):
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, ...):
    ...
```

New in version 3.2. Deprecated since version 3.3: It is now possible to use `classmethod` with `abstractmethod()`, making this decorator redundant.

```python
@abc.abstractstaticmethod
```

A subclass of the built-in `staticmethod()`, indicating an abstract staticmethod. Otherwise it is similar to `abstractmethod()`.

This special case is deprecated, as the `staticmethod()` decorator is now correctly identified as abstract when applied to an abstract method:

```python
class C(metaclass=ABCMeta):
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(...):
    ...
```

New in version 3.2. Deprecated since version 3.3: It is now possible to use `staticmethod` with `abstractmethod()`, making this decorator redundant.

```python
@abc.abstractproperty
```

A subclass of the built-in `property()`, indicating an abstract property.

Using this function requires that the class’s metaclass is `ABCMeta` or is derived from it. A class that has a metaclass derived from `ABCMeta` cannot be instantiated unless all of its abstract methods and properties are overridden. The abstract properties can be called using any of the normal ‘super’ call mechanisms.

This special case is deprecated, as the `property()` decorator is now correctly identified as abstract when applied to an abstract method:

```python
class C(metaclass=ABCMeta):
    @property
    @abstractmethod
    def my_abstract_property(self):
    ...
```

The above example defines a read-only property; you can also define a read-write abstract property by appropriately marking one or more of the underlying methods as abstract:

```python
class C(metaclass=ABCMeta):
    @property
    def x(self):
    ...

    @x.setter
    @abstractmethod
    def x(self, val):
    ...
```

If only some components are abstract, only those components need to be updated to create a concrete property in a subclass:
class D(C):
    @C.x.setter
def x(self, val):
...

Deprecated since version 3.3: It is now possible to use property, property.getter(), property.setter() and property.deleter() with abstractmethod(), making this decorator redundant.

28.8 atexit — Exit handlers

The atexit module defines functions to register and unregister cleanup functions. Functions thus registered are automatically executed upon normal interpreter termination. atexit runs these functions in the reverse order in which they were registered; if you register A, B, and C, at interpreter termination time they will be run in the order C, B, A.

Note: The functions registered via this module are not called when the program is killed by a signal not handled by Python, when a Python fatal internal error is detected, or when os._exit() is called.

atexit.register(func, *args, **kargs)
Register func as a function to be executed at termination. Any optional arguments that are to be passed to func must be passed as arguments to register(). It is possible to register the same function and arguments more than once.

At normal program termination (for instance, if sys.exit() is called or the main module’s execution completes), all functions registered are called in last in, first out order. The assumption is that lower level modules will normally be imported before higher level modules and thus must be cleaned up later.

If an exception is raised during execution of the exit handlers, a traceback is printed (unless SystemExit is raised) and the exception information is saved. After all exit handlers have had a chance to run the last exception to be raised is re-raised.

This function returns func, which makes it possible to use it as a decorator.

atexit.unregister(func)
Remove func from the list of functions to be run at interpreter shutdown. After calling unregister(), func is guaranteed not to be called when the interpreter shuts down, even if it was registered more than once.
unregister() silently does nothing if func was not previously registered.

See Also:
Module readline Useful example of atexit to read and write readline history files.

28.8.1 atexit Example

The following simple example demonstrates how a module can initialize a counter from a file when it is imported and save the counter’s updated value automatically when the program terminates without relying on the application making an explicit call into this module at termination.

try:
    with open("counterfile") as infile:
        _count = int(infile.read())
except FileNotFoundError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
with open("counterfile", "w") as outfile:
    outfile.write("%d" % _count)

import atexit
atexit.register(savecounter)

Positional and keyword arguments may also be passed to register() to be passed along to the registered function when it is called:

```python
def goodbye(name, adjective):
    print('Goodbye, %s, it was %s to meet you.' % (name, adjective))
```

```python
import atexit
atexit.register(goodbye, 'Donny', 'nice')
```

# or:
```python
atexit.register(goodbye, adjective='nice', name='Donny')
```

Usage as a decorator:

```python
import atexit

@atexit.register
def goodbye():
    print("You are now leaving the Python sector.")
```

This only works with functions that can be called without arguments.

## 28.9 traceback — Print or retrieve a stack traceback

This module provides a standard interface to extract, format and print stack traces of Python programs. It exactly mimics the behavior of the Python interpreter when it prints a stack trace. This is useful when you want to print stack traces under program control, such as in a “wrapper” around the interpreter.

The module uses traceback objects — this is the object type that is stored in the `sys.last_traceback` variable and returned as the third item from `sys.exc_info()`.

The module defines the following functions:

- `traceback.print_tb(traceback, limit=None, file=None)`
  - Print up to `limit` stack trace entries from `traceback`. If `limit` is omitted or `None`, all entries are printed. If `file` is omitted or `None`, the output goes to `sys.stderr`; otherwise it should be an open file or file-like object to receive the output.

- `traceback.print_exception(type, value, traceback, limit=None, file=None, chain=True)`
  - Print exception information and up to `limit` stack trace entries from `traceback` to `file`. This differs from `print_tb()` in the following ways:
    - If `traceback` is not `None`, it prints a header `Traceback (most recent call last):`
    - It prints the exception `type` and `value` after the stack trace
    - If `type` is `SyntaxError` and `value` has the appropriate format, it prints the line where the syntax error occurred with a caret indicating the approximate position of the error.

    If `chain` is true (the default), then chained exceptions (the `__cause__` or `__context__` attributes of the exception) will be printed as well, like the interpreter itself does when printing an unhandled exception.

- `traceback.print_exc(limit=None, file=None, chain=True)`
  - This is a shorthand for `print_exception(*sys.exc_info())`.

- `traceback.print_last(limit=None, file=None, chain=True)`
  - This is a shorthand for `print_exception(sys.last_type, sys.last_value, sys.last_traceback)`.

This module is intended primarily for programmatic situations where you want to print stack traces under program control, rather than as a fallback when the interpreter itself is unable to print a stack trace. It is useful for debugging, but should generally not be used to replace the interpreter's own stack trace printing, as the two can differ in unexpected ways.
sys.last_traceback, limit, file). In general it will work only after an exception has reached an interactive prompt (see sys.last_type).

traceback.print_stack (f=None, limit=None, file=None)
This function prints a stack trace from its invocation point. The optional f argument can be used to specify an alternate stack frame to start. The optional limit and file arguments have the same meaning as for print_exception().

traceback.extract_tb (traceback, limit=None)
Return a list of up to limit “pre-processed” stack trace entries extracted from the traceback object traceback. It is useful for alternate formatting of stack traces. If limit is omitted or None, all entries are extracted. A “pre-processed” stack trace entry is a quadruple (filename, line number, function name, text) representing the information that is usually printed for a stack trace. The text is a string with leading and trailing whitespace stripped; if the source is not available it is None.

traceback.extract_stack (f=None, limit=None)
Extract the raw traceback from the current stack frame. The return value has the same format as for extract_tb(). The optional f and limit arguments have the same meaning as for print_stack().

traceback.format_list (list)
Given a list of tuples as returned by extract_tb() or extract_stack(), return a list of strings ready for printing. Each string in the resulting list corresponds to the item with the same index in the argument list. Each string ends in a newline; the strings may contain internal newlines as well, for those items whose source text line is not None.

traceback.format_exception_only (type, value)
Format the exception part of a traceback. The arguments are the exception type and value such as given by sys.last_type and sys.last_value. The return value is a list of strings, each ending in a newline. Normally, the list contains a single string; however, for SyntaxError exceptions, it contains several lines that (when printed) display detailed information about where the syntax error occurred. The message indicating which exception occurred is the always last string in the list.

traceback.format_exception (type, value, tb, limit=None, chain=True)
Format a stack trace and the exception information. The arguments have the same meaning as the corresponding arguments to print_exception(). The return value is a list of strings, each ending in a newline and some containing internal newlines. When these lines are concatenated and printed, exactly the same text is printed as does print_exception().

traceback.format_exc (limit=None, chain=True)
This is like print_exc(limit) but returns a string instead of printing to a file.

traceback.format_tb (tb, limit=None)
A shorthand for format_list(extract_tb(tb, limit)).

traceback.format_stack (f=None, limit=None)
A shorthand for format_list(extract_stack(f, limit)).

28.9.1 Traceback Examples

This simple example implements a basic read-eval-print loop, similar to (but less useful than) the standard Python interactive interpreter loop. For a more complete implementation of the interpreter loop, refer to the code module.

import sys, traceback

def run_user_code(envdir):
    source = input(">>> ")
    try:
        exec (source, envdir)
    except:
        print("Exception in user code:")
        print("-"*60)
        traceback.print_exc(file=sys.stdout)

28.9. traceback — Print or retrieve a stack traceback
print("\-"*60)

envdir = {}
while True:
    run_user_code(envdir)

The following example demonstrates the different ways to print and format the exception and traceback:

```python
import sys, traceback

def lumberjack():
    bright_side_of_death()

def bright_side_of_death():
    return tuple()[0]

try:
    lumberjack()
except IndexError:
    exc_type, exc_value, exc_traceback = sys.exc_info()
    print("*** print_tb:")
    traceback.print_tb(exc_traceback, limit=1, file=sys.stdout)
    print("*** print_exception:")
    traceback.print_exception(exc_type, exc_value, exc_traceback, limit=2, file=sys.stdout)
    print("*** print_exc:")
    traceback.print_exc()
    print("*** format_exc, first and last line:")
    formatted_lines = traceback.format_exc().splitlines()
    print(formatted_lines[0])
    print(formatted_lines[-1])
    print("*** format_exception:")
    print(repr(traceback.format_exception(exc_type, exc_value, exc_traceback)))
    print("*** extract_tb:")
    print(repr(traceback.extract_tb(exc_traceback)))
    print("*** format_tb:")
    print(repr(traceback.format_tb(exc_traceback)))
    print("*** tb_lineno:")
    print(exc_traceback.tb_lineno)
```

The output for the example would look similar to this:

```plaintext
*** print_tb:
  File "<doctest...>", line 10, in <module>
    lumberjack()
*** print_exception:
  Traceback (most recent call last):
    File "<doctest...>", line 10, in <module>
      lumberjack()
    File "<doctest...>", line 4, in lumberjack
      bright_side_of_death()
  IndexError: tuple index out of range
*** print_exc:
  Traceback (most recent call last):
    File "<doctest...>", line 10, in <module>
      lumberjack()
    File "<doctest...>", line 4, in lumberjack
      bright_side_of_death()
  IndexError: tuple index out of range
*** format_exc, first and last line:
```

1250 Chapter 28. Python Runtime Services
Traceback (most recent call last):
IndexError: tuple index out of range

The following example shows the different ways to print and format the stack:

```python
>>> import traceback

>>> def another_function():
...   lumberstack()
...

>>> def lumberstack():
...   traceback.print_stack()
...
>>> another_function()

```

This last example demonstrates the final few formatting functions:

```python
>>> import traceback

>>> traceback.format_list([('spam.py', 3, '<module>', 'spam.eggs()'),
                         ('eggs.py', 42, 'eggs', 'return "bacon"')])

```

28.10 __future__ — Future statement definitions

Source code: Lib/__future__.py
__future__ is a real module, and serves three purposes:

- To avoid confusing existing tools that analyze import statements and expect to find the modules they’re importing.
- To ensure that future statements run under releases prior to 2.1 at least yield runtime exceptions (the import of __future__ will fail, because there was no module of that name prior to 2.1).
- To document when incompatible changes were introduced, and when they will be — or were — made mandatory. This is a form of executable documentation, and can be inspected programmatically via importing __future__ and examining its contents.

Each statement in __future__.py is of the form:

FeatureName = __Feature(OptionalRelease, MandatoryRelease, CompilerFlag)

where, normally, OptionalRelease is less than MandatoryRelease, and both are 5-tuples of the same form as sys.version_info:

(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int
PY_MINOR_VERSION, # the 1; an int
PY_MICRO_VERSION, # the 0; an int
PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; string
PY_RELEASE_SERIAL # the 3; an int
)

OptionalRelease records the first release in which the feature was accepted.

In the case of a MandatoryRelease that has not yet occurred, MandatoryRelease predicts the release in which the feature will become part of the language.

Else MandatoryRelease records when the feature became part of the language; in releases at or after that, modules no longer need a future statement to use the feature in question, but may continue to use such imports.

MandatoryRelease may also be None, meaning that a planned feature got dropped.

Instances of class __Feature have two corresponding methods, getOptionalRelease() and getMandatoryRelease().

CompilerFlag is the (bitfield) flag that should be passed in the fourth argument to the built-in function compile() to enable the feature in dynamically compiled code. This flag is stored in the compiler_flag attribute on __Feature instances.

No feature description will ever be deleted from __future__. Since its introduction in Python 2.1 the following features have found their way into the language using this mechanism:

<table>
<thead>
<tr>
<th>feature</th>
<th>optional in</th>
<th>mandatory in</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>nested_scopes</td>
<td>2.1.0b1</td>
<td>2.2</td>
<td>PEP 227: Statically Nested Scopes</td>
</tr>
<tr>
<td>generators</td>
<td>2.2.0a1</td>
<td>2.3</td>
<td>PEP 255: Simple Generators</td>
</tr>
<tr>
<td>division</td>
<td>2.2.0a2</td>
<td>3.0</td>
<td>PEP 238: Changing the Division Operator</td>
</tr>
<tr>
<td>absolute_import</td>
<td>2.5.0a1</td>
<td>3.0</td>
<td>PEP 328: Imports: Multi-Line and Absolute/Relative</td>
</tr>
<tr>
<td>with_statement</td>
<td>2.5.0a1</td>
<td>2.6</td>
<td>PEP 343: The “with” Statement</td>
</tr>
<tr>
<td>print_function</td>
<td>2.6.0a2</td>
<td>3.0</td>
<td>PEP 3105: Make print a function</td>
</tr>
<tr>
<td>unicode_literals</td>
<td>2.6.0a2</td>
<td>3.0</td>
<td>PEP 3112: Bytes literals in Python 3000</td>
</tr>
</tbody>
</table>

See Also:

future How the compiler treats future imports.
28.11 gc — Garbage Collector interface

This module provides an interface to the optional garbage collector. It provides the ability to disable the collector, tune the collection frequency, and set debugging options. It also provides access to unreachable objects that the collector found but cannot free. Since the collector supplements the reference counting already used in Python, you can disable the collector if you are sure your program does not create reference cycles. Automatic collection can be disabled by calling `gc.disable()`. To debug a leaking program call `gc.set_debug(gc.DEBUG_LEAK)`. Notice that this includes `gc.DEBUG_SAVEALL`, causing garbage-collected objects to be saved in `gc.garbage` for inspection.

The `gc` module provides the following functions:

- `gc.enable()`  
  Enable automatic garbage collection.

- `gc.disable()`  
  Disable automatic garbage collection.

- `gc.isenabled()`  
  Returns true if automatic collection is enabled.

- `gc.collect(generations=2)`  
  With no arguments, run a full collection. The optional argument `generation` may be an integer specifying which generation to collect (from 0 to 2). A `ValueError` is raised if the generation number is invalid. The number of unreachable objects found is returned.

  The free lists maintained for a number of built-in types are cleared whenever a full collection or collection of the highest generation (2) is run. Not all items in some free lists may be freed due to the particular implementation, in particular `float`.

- `gc.set_debug(flags)`  
  Set the garbage collection debugging flags. Debugging information will be written to `sys.stderr`. See below for a list of debugging flags which can be combined using bit operations to control debugging.

- `gc.get_debug()`  
  Return the debugging flags currently set.

- `gc.get_objects()`  
  Returns a list of all objects tracked by the collector, excluding the list returned.

- `gc.set_threshold(threshold0[, threshold1[, threshold2 ]])`  
  Set the garbage collection thresholds (the collection frequency). Setting `threshold0` to zero disables collection.

  The GC classifies objects into three generations depending on how many collection sweeps they have survived. New objects are placed in the youngest generation (generation 0). If an object survives a collection it is moved into the next older generation. Since generation 2 is the oldest generation, objects in that generation remain there after a collection. In order to decide when to run, the collector keeps track of the number of object allocations and deallocations since the last collection. When the number of allocations minus the number of deallocations exceeds `threshold0`, collection starts. Initially only generation 0 is examined. If generation 0 has been examined more than `threshold1` times since generation 1 has been examined, then generation 1 is examined as well. Similarly, `threshold2` controls the number of collections of generation 1 before collecting generation 2.

- `gc.get_count()`  
  Return the current collection counts as a tuple of `(count0, count1, count2)`.

- `gc.get_threshold()`  
  Return the current collection thresholds as a tuple of `(threshold0, threshold1, threshold2)`.

- `gc.get_referrers(*objs)`  
  Return the list of objects that directly refer to any of `objs`. This function will only locate those containers which support garbage collection; extension types which do refer to other objects but do not support garbage collection will not be found.
Note that objects which have already been dereferenced, but which live in cycles and have not yet been collected by the garbage collector can be listed among the resulting referrers. To get only currently live objects, call `collect()` before calling `get_referrers()`.

Care must be taken when using objects returned by `get_referrers()` because some of them could still be under construction and hence in a temporarily invalid state. Avoid using `get_referrers()` for any purpose other than debugging.

```python
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
False
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
False
>>> gc.is_tracked({"a": []})
True
```

New in version 3.1.

The following variables are provided for read-only access (you can mutate the values but should not rebind them):

```python
>>> gc.garbage
tuple()
```

A list of objects which the collector found to be unreachable but could not be freed (uncollectable objects). By default, this list contains only objects with `__del__()` methods. Objects that have `__del__()` methods and are part of a reference cycle cause the entire reference cycle to be uncollectable, including objects not necessarily in the cycle but reachable only from it. Python doesn’t collect such cycles automatically because, in general, it isn’t possible for Python to guess a safe order in which to run the `__del__()` methods. If you know a safe order, you can force the issue by examining the `garbage` list, and explicitly breaking cycles due to your objects within the list. Note that these objects are kept alive even so by virtue of being in the `garbage` list, so they should be removed from `garbage` too. For example, after breaking cycles, do `del gc.garbage[:]` to empty the list. It’s generally better to avoid the issue by not creating cycles containing objects with `__del__()` methods, and `garbage` can be examined in that case to verify that no such cycles are being created.

If `DEBUG_SAVEALL` is set, then all unreachable objects will be added to this list rather than freed. Changed in version 3.2: If this list is non-empty at interpreter shutdown, a `ResourceWarning` is emitted, which is silent by default. If `DEBUG_UNCOLLECTABLE` is set, in addition all uncollectable objects are printed.

```python
>>> gc.callbacks
<list object at 0x100000000>
```

A list of callbacks that will be invoked by the garbage collector before and after collection. The callbacks will be called with two arguments, `phase` and `info`.

`phase` can be one of two values:

- “start”: The garbage collection is about to start.
“stop”: The garbage collection has finished.

`info` is a dict providing more information for the callback. The following keys are currently defined:

“generation”: The oldest generation being collected.
“collected”: When `phase` is “stop”, the number of objects successfully collected.
“uncollectable”: When `phase` is “stop”, the number of objects that could not be collected and were put in `garbage`.

Applications can add their own callbacks to this list. The primary use cases are:

Gathering statistics about garbage collection, such as how often various generations are collected, and how long the collection takes.

Allowing applications to identify and clear their own uncollectable types when they appear in `garbage`.

New in version 3.3.

The following constants are provided for use with `set_debug()`:

- `gc.DEBUG_STATS`
  Print statistics during collection. This information can be useful when tuning the collection frequency.

- `gc.DEBUG_COLLECTABLE`
  Print information on collectable objects found.

- `gc.DEBUG_UNCOLLECTABLE`
  Print information of uncollectable objects found (objects which are not reachable but cannot be freed by the collector). These objects will be added to the `garbage` list. Changed in version 3.2: Also print the contents of the `garbage` list at interpreter shutdown, if it isn’t empty.

- `gc.DEBUG_SAVEALL`
  When set, all unreachable objects found will be appended to `garbage` rather than being freed. This can be useful for debugging a leaking program.

- `gc.DEBUG_LEAK`
  The debugging flags necessary for the collector to print information about a leaking program (equal to `DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_SAVEALL`).

### 28.12 `inspect` — Inspect live objects

**Source code:** `Lib/inspect.py`

The `inspect` module provides several useful functions to help get information about live objects such as modules, classes, methods, functions, tracebacks, frame objects, and code objects. For example, it can help you examine the contents of a class, retrieve the source code of a method, extract and format the argument list for a function, or get all the information you need to display a detailed traceback.

There are four main kinds of services provided by this module: type checking, getting source code, inspecting classes and functions, and examining the interpreter stack.

#### 28.12.1 Types and members

The `getmembers()` function retrieves the members of an object such as a class or module. The sixteen functions whose names begin with “is” are mainly provided as convenient choices for the second argument to `getmembers()`. They also help you determine when you can expect to find the following special attributes:

<table>
<thead>
<tr>
<th>Type</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
</table>

Continued on next page
### Table 28.1 – continued from previous page

<table>
<thead>
<tr>
<th>Type</th>
<th><strong>doc</strong></th>
<th><strong>file</strong></th>
<th><strong>module</strong></th>
<th><strong>doc</strong></th>
<th><strong>name</strong></th>
<th><strong>func</strong></th>
<th><strong>self</strong></th>
<th><strong>name</strong></th>
<th><strong>code</strong></th>
<th><strong>defaults</strong></th>
<th><strong>globals</strong></th>
<th><strong>doc</strong></th>
<th><strong>name</strong></th>
<th><strong>func</strong></th>
<th><strong>self</strong></th>
<th><strong>name</strong></th>
<th><strong>code</strong></th>
<th><strong>defaults</strong></th>
<th><strong>globals</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>module</td>
<td>documentation string</td>
<td>filename (missing for built-in modules)</td>
<td>name of module in which this class was defined</td>
<td>documentation string</td>
<td>name with which this method was defined</td>
<td>function object containing implementation of method</td>
<td>instance to which this method is bound, or None</td>
<td>documentation string</td>
<td>name with which this function was defined</td>
<td>code object containing compiled function bytecode</td>
<td>tuple of any default values for arguments</td>
<td>global namespace in which this function was defined</td>
<td>frame object at this level</td>
<td>index of last attempted instruction in bytecode</td>
<td>current line number in Python source code</td>
<td>next inner traceback object (called by this level)</td>
<td>bultins namespace seen by this frame</td>
<td>code object object being executed in this frame</td>
<td>global namespace seen by this frame</td>
</tr>
</tbody>
</table>

**inspect.getmembers**(*object*, [, *predicate]*)

Return all the members of an object in a list of (name, value) pairs sorted by name. If the optional *predicate* argument is supplied, only members for which the predicate returns a true value are included.

**Note:** *getmembers()* does not return metaclass attributes when the argument is a class (this behavior is inherited from the *dir()* function).

**inspect.getmoduleinfo**(*path*)

Returns a named tuple ModuleInfo(name, suffix, mode, module_type) of values that describe how Python will interpret the file identified by *path* if it is a module, or None if it would not be identified as a module. In that tuple, name is the name of the module without the name of any enclosing package, suffix is the trailing part of the file name (which may not be a dot-delimited extension), mode is the open() mode that would be used (‘r’ or ‘rb’), and module_type is an integer giving the type of the module. module_type will have a value which can be compared to the constants defined in the *imp* module;
see the documentation for that module for more information on module types. Deprecated since version 3.3:
You may check the file path’s suffix against the supported suffixes listed in `importlib.machinery` to infer the same information.

`inspect.getmodulename(path)`
Return the name of the module named by the file `path`, without including the names of enclosing packages. The file extension is checked against all of the entries in `importlib.machinery.all_suffixes()`. If it matches, the final path component is returned with the extension removed. Otherwise, None is returned.

Note that this function only returns a meaningful name for actual Python modules - paths that potentially refer to Python packages will still return None. Changed in version 3.3: This function is now based directly on `importlib` rather than the deprecated `getmoduleinfo()`.

`inspect.ismodule(object)`
Return true if the object is a module.

`inspect.isclass(object)`
Return true if the object is a class, whether built-in or created in Python code.

`inspect.ismethod(object)`
Return true if the object is a bound method written in Python.

`inspect.isfunction(object)`
Return true if the object is a Python function, which includes functions created by a `lambda` expression.

`inspect.isgeneratorfunction(object)`
Return true if the object is a Python generator function.

`inspect.isgenerator(object)`
Return true if the object is a generator.

`inspect.istraceback(object)`
Return true if the object is a traceback.

`inspect.isframe(object)`
Return true if the object is a frame.

`inspect.iscode(object)`
Return true if the object is a code.

`inspect.isbuiltin(object)`
Return true if the object is a built-in function or a bound built-in method.

`inspect.isroutine(object)`
Return true if the object is a user-defined or built-in function or method.

`inspect.isabstract(object)`
Return true if the object is an abstract base class.

`inspect.ismethoddescriptor(object)`
Return true if the object is a method descriptor, but not if `ismethod()`, `isclass()`, `isfunction()` or `isbuiltin()` are true.

This, for example, is true of `int.__add__`. An object passing this test has a `__get__` attribute but not a `__set__` attribute, but beyond that the set of attributes varies. `__name__` is usually sensible, and `__doc__` often is.

Methods implemented via descriptors that also pass one of the other tests return false from the `ismethoddescriptor()` test, simply because the other tests promise more – you can, e.g., count on having the `__func__` attribute (etc) when an object passes `ismethod()`.

`inspect.isdatadescriptor(object)`
Return true if the object is a data descriptor.

Data descriptors have both a `__get__` and a `__set__` attribute. Examples are properties (defined in Python), getsets, and members. The latter two are defined in C and there are more specific tests available for those types, which is robust across Python implementations. Typically, data descriptors will also have
__name__ and __doc__ attributes (properties, getsets, and members have both of these attributes), but this is not guaranteed.

inspect.isgetsetdescriptor(object)
Return true if the object is a getset descriptor.

**CPython implementation detail**: getsets are attributes defined in extension modules via PyGetSetDef structures. For Python implementations without such types, this method will always return False.

inspect.ismemberdescriptor(object)
Return true if the object is a member descriptor.

**CPython implementation detail**: Member descriptors are attributes defined in extension modules via PyMemberDef structures. For Python implementations without such types, this method will always return False.

### 28.12.2 Retrieving source code

inspect.getdoc(object)
Get the documentation string for an object, cleaned up with cleandoc().

inspect.getcomments(object)
Return in a single string any lines of comments immediately preceding the object’s source code (for a class, function, or method), or at the top of the Python source file (if the object is a module).

inspect.getfile(object)
Return the name of the (text or binary) file in which an object was defined. This will fail with a TypeError if the object is a built-in module, class, or function.

inspect.getmodule(object)
Try to guess which module an object was defined in.

inspect.getsourcefile(object)
Return the name of the Python source file in which an object was defined. This will fail with a TypeError if the object is a built-in module, class, or function.

inspect.getsourcelines(object)
Return a list of source lines and starting line number for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of the lines corresponding to the object and the line number indicates where in the original source file the first line of code was found. An OSError is raised if the source code cannot be retrieved. Changed in version 3.3: OSError is raised instead of IOError, now an alias of the former.

inspect.getsource(object)
Return the text of the source code for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An OSError is raised if the source code cannot be retrieved. Changed in version 3.3: OSError is raised instead of IOError, now an alias of the former.

inspect.cleandoc(doc)
Clean up indentation from docstrings that are indented to line up with blocks of code. Any whitespace that can be uniformly removed from the second line onwards is removed. Also, all tabs are expanded to spaces.

### 28.12.3 Introspecting callables with the Signature object

New in version 3.3. The Signature object represents the call signature of a callable object and its return annotation. To retrieve a Signature object, use the signature() function.

inspect.signature(callable)
Return a Signature object for the given callable:
>>> from inspect import signature
>>> def foo(a, *, b:int, **kwargs):
...    pass

>>> sig = signature(foo)

>>> str(sig)
'(a, *, b:int, **kwargs)'

>>> str(sig.parameters['b'])
'b:int'

>>> sig.parameters['b'].annotation
<class 'int'>

Accepts a wide range of python callables, from plain functions and classes to `functools.partial()` objects.

**Note:** Some callables may not be introspectable in certain implementations of Python. For example, in CPython, built-in functions defined in C provide no metadata about their arguments.

```python
class inspect.Signature
A Signature object represents the call signature of a function and its return annotation. For each parameter accepted by the function it stores a `Parameter` object in its `parameters` collection.

Signature objects are immutable. Use `Signature.replace()` to make a modified copy.

`empty`
A special class-level marker to specify absence of a return annotation.

`parameters`
An ordered mapping of parameters’ names to the corresponding `Parameter` objects.

`return_annotation`
The “return” annotation for the callable. If the callable has no “return” annotation, this attribute is set to `Signature.empty`.

`bind(*args, **kwargs)`
Create a mapping from positional and keyword arguments to parameters. Returns `BoundArguments` if `*args` and `**kwargs` match the signature, or raises a `TypeError`.

`bind_partial(*args, **kwargs)`
Works the same way as `Signature.bind()`, but allows the omission of some required arguments (mimics `functools.partial()` behavior.) Returns `BoundArguments`, or raises a `TypeError` if the passed arguments do not match the signature.

`replace(*[, parameters][, return_annotation])`
Create a new Signature instance based on the instance replace was invoked on. It is possible to pass different parameters and/or `return_annotation` to override the corresponding properties of the base signature. To remove `return_annotation` from the copied Signature, pass in `Signature.empty`.

```python
>>> def test(a, b):
...    pass

>>> sig = signature(test)

>>> new_sig = sig.replace(return_annotation="new return anno")

>>> str(new_sig)
"(a, b) -> 'new return anno'"
```
class inspect.Parameter

Parameter objects are immutable. Instead of modifying a Parameter object, you can use Parameter.replace() to create a modified copy.

empty

A special class-level marker to specify absence of default values and annotations.

name

The name of the parameter as a string. Must be a valid python identifier name (with the exception of POSITIONAL_ONLY parameters, which can have it set to None).

default

The default value for the parameter. If the parameter has no default value, this attribute is set to Parameter.empty.

annotation

The annotation for the parameter. If the parameter has no annotation, this attribute is set to Parameter.empty.

kind

Describes how argument values are bound to the parameter. Possible values (accessible via Parameter, like Parameter.KEYWORD_ONLY):

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSITIONAL_ONLY</td>
<td>Value must be supplied as a positional argument. Python has no explicit</td>
</tr>
<tr>
<td></td>
<td>syntax for defining positional-only parameters, but many built-in and</td>
</tr>
<tr>
<td></td>
<td>extension module functions (especially those that accept only one or</td>
</tr>
<tr>
<td></td>
<td>two parameters) accept them.</td>
</tr>
<tr>
<td>POSITIONAL_OR_KEYWORD</td>
<td>Value may be supplied as either a keyword or positional argument</td>
</tr>
<tr>
<td></td>
<td>(this is the standard binding behaviour for functions implemented in</td>
</tr>
<tr>
<td></td>
<td>Python.)</td>
</tr>
<tr>
<td>VAR_POSITIONAL</td>
<td>A tuple of positional arguments that aren’t bound to any other parameter.</td>
</tr>
<tr>
<td></td>
<td>This corresponds to a *args parameter in a Python function definition.</td>
</tr>
<tr>
<td>KEYWORD_ONLY</td>
<td>Value must be supplied as a keyword argument. Keyword only parameters</td>
</tr>
<tr>
<td></td>
<td>are those which appear after a * or *args entry in a Python function</td>
</tr>
<tr>
<td></td>
<td>definition.</td>
</tr>
<tr>
<td>VAR_KEYWORD</td>
<td>A dict of keyword arguments that aren’t bound to any other parameter.</td>
</tr>
<tr>
<td></td>
<td>This corresponds to a **kwargs parameter in a Python function definition.</td>
</tr>
</tbody>
</table>

Example: print all keyword-only arguments without default values:

```python
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     if (param.kind == param.KEYWORD_ONLY and
...         param.default is param.empty):
...         print('Parameter:', param)
Parameter: c
```

replace(*[name]*, kind*, default*, annotation*)

Create a new Parameter instance based on the instance replaced was invoked on. To override a Parameter attribute, pass the corresponding argument. To remove a default value or/and an annotation from a Parameter, pass Parameter.empty.

```python
>>> from inspect import Parameter
>>> param = Parameter('foo', Parameter.KEYWORD_ONLY, default=42)
>>> str(param)
```
class inspect.BoundArguments

Result of a Signature.bind() or Signature.bind_partial() call. Holds the mapping of arguments to the function’s parameters.

arguments

An ordered, mutable mapping (collections.OrderedDict) of parameters’ names to arguments’ values. Contains only explicitly bound arguments. Changes in arguments will reflect in args and kwargs.

Should be used in conjunction with Signature.parameters for any argument processing purposes.

Note: Arguments for which Signature.bind() or Signature.bind_partial() relied on a default value are skipped. However, if needed, it is easy to include them.

>>> def foo(a, b=10):
...     pass

...     sig = signature(foo)
...     ba = sig.bind(5)

...     for param in sig.parameters.values():
...         if param.name not in ba.arguments:
...             ba.arguments[param.name] = param.default

...     ba.args, bakwargs
((5,), {})

args

A tuple of positional arguments values. Dynamically computed from the arguments attribute.

kwargs

A dict of keyword arguments values. Dynamically computed from the arguments attribute.

The args and kwargs properties can be used to invoke functions:

def test(a, *, b):
...      sig = signature(test)
ba = sig.bind(10, b=20)
test(*ba.args, **bakwargs)

See Also:

PEP 362 - Function Signature Object. The detailed specification, implementation details and examples.
28.12.4 Classes and functions

`inspect.getclasstree(classes, unique=False)`

Arrange the given list of classes into a hierarchy of nested lists. Where a nested list appears, it contains classes derived from the class whose entry immediately precedes the list. Each entry is a 2-tuple containing a class and a tuple of its base classes. If the `unique` argument is true, exactly one entry appears in the returned structure for each class in the given list. Otherwise, classes using multiple inheritance and their descendants will appear multiple times.

`inspect.getargspec(func)`

Get the names and default values of a Python function’s arguments. A `named tuple` `ArgSpec(args, varargs, keywords, defaults)` is returned. `args` is a list of the argument names. `varargs` and `keywords` are the names of the `*` and `**` arguments or `None`. `defaults` is a tuple of default argument values or `None` if there are no default arguments; if this tuple has `n` elements, they correspond to the last `n` elements listed in `args`. Deprecated since version 3.0: Use `getfullargspec()` instead, which provides information about keyword-only arguments and annotations.

`inspect.getfullargspec(func)`

Get the names and default values of a Python function’s arguments. A `named tuple` is returned:

`FullArgSpec(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults, annotations)`

`args` is a list of the argument names. `varargs` and `varkw` are the names of the `*` and `**` arguments or `None`. `defaults` is an `n`-tuple of the default values of the last `n` arguments, or `None` if there are no default arguments. `kwonlyargs` is a list of keyword-only argument names. `kwonlydefaults` is a dictionary mapping names from `kwonlyargs` to defaults. `annotations` is a dictionary mapping argument names to annotations.

The first four items in the tuple correspond to `getargspec()`.

**Note:** Consider using the new `Signature Object` interface, which provides a better way of introspecting functions.

`inspect.getargvalues(frame)`

Get information about arguments passed into a particular frame. A `named tuple` `ArgInfo(args, varargs, keywords, locals)` is returned. `args` is a list of the argument names. `varargs` and `keywords` are the names of the `*` and `**` arguments or `None`. `locals` is the locals dictionary of the given frame.

`inspect.formatargspec(*getfullargspec(f))`  
'`(a: int, b: float)'
**inspect.getmro**(cls)

Return a tuple of class cls’s base classes, including cls, in method resolution order. No class appears more than once in this tuple. Note that the method resolution order depends on cls’s type. Unless a very peculiar user-defined metatype is in use, cls will be the first element of the tuple.

**inspect.getcallargs**(func[, *args][, **kwds])

Bind the args and kwds to the argument names of the Python function or method func, as if it was called with them. For bound methods, bind also the first argument (typically named self) to the associated instance. A dict is returned, mapping the argument names (including the names of the * and ** arguments, if any) to their values from args and kwds. In case of invoking func incorrectly, i.e. whenever func(*args, **kwds) would raise an exception because of incompatible signature, an exception of the same type and the same or similar message is raised. For example:

```python
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...   pass
>>> getcallargs(f, 1, 2, 3) == {'a': 1, 'named': {}, 'b': 2, 'pos': (3,)}
True
>>> getcallargs(f, a=2, x=4) == {'a': 2, 'named': {'x': 4}, 'b': 1, 'pos': ()}
True
>>> getcallargs(f)
Traceback (most recent call last):
...  TypeError: f() missing 1 required positional argument: 'a'
```

New in version 3.2.

**Note:** Consider using the new Signature.bind() instead.

**inspect.getclosurevars**(func)

Get the mapping of external name references in a Python function or method func to their current values. A named tuple ClosureVars(nonlocals, globals, builtins, unbound) is returned. nonlocals maps referenced names to lexical closure variables, globals to the function’s module globals and builtins to the builtins visible from the function body. unbound is the set of names referenced in the function that could not be resolved at all given the current module globals and builtins.

TypeError is raised if func is not a Python function or method. New in version 3.3.

### 28.12.5 The interpreter stack

When the following functions return “frame records,” each record is a tuple of six items: the frame object, the filename, the line number of the current line, the function name, a list of lines of context from the source code, and the index of the current line within that list.

**Note:** Keeping references to frame objects, as found in the first element of the frame records these functions return, can cause your program to create reference cycles. Once a reference cycle has been created, the lifespan of all objects which can be accessed from the objects which form the cycle can become much longer even if Python’s optional cycle detector is enabled. If such cycles must be created, it is important to ensure they are explicitly broken to avoid the delayed destruction of objects and increased memory consumption which occurs.

Though the cycle detector will catch these, destruction of the frames (and local variables) can be made deterministic by removing the cycle in a finally clause. This is also important if the cycle detector was disabled when Python was compiled or using gc.disable(). For example:

```python
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
```
finally:
    del frame

The optional context argument supported by most of these functions specifies the number of lines of context to return, which are centered around the current line.

inspect.getframeinfo(frame, context=1)

Get information about a frame or traceback object. A named tuple Traceback(filename, lineno, function, code_context, index) is returned.

inspect.getouterframes(frame, context=1)

Get a list of frame records for a frame and all outer frames. These frames represent the calls that lead to the creation of frame. The first entry in the returned list represents frame; the last entry represents the outermost call on frame’s stack.

inspect.getinnerframes(traceback, context=1)

Get a list of frame records for a traceback’s frame and all inner frames. These frames represent calls made as a consequence of frame. The first entry in the list represents traceback; the last entry represents where the exception was raised.

inspect.currentframe()

Return the frame object for the caller’s stack frame.

CPython implementation detail: This function relies on Python stack frame support in the interpreter, which isn’t guaranteed to exist in all implementations of Python. If running in an implementation without Python stack frame support this function returns None.

inspect.stack(context=1)

Return a list of frame records for the caller’s stack. The first entry in the returned list represents the caller; the last entry represents the outermost call on the stack.

inspect.trace(context=1)

Return a list of frame records for the stack between the current frame and the frame in which an exception currently being handled was raised in. The first entry in the list represents the caller; the last entry represents where the exception was raised.

28.12.6 Fetching attributes statically

Both getattr() and hasattr() can trigger code execution when fetching or checking for the existence of attributes. Descriptors, like properties, will be invoked and __getattr__() and __getattribute__() may be called.

For cases where you want passive introspection, like documentation tools, this can be inconvenient. getattr_static() has the same signature as getattr() but avoids executing code when it fetches attributes.

inspect.getattr_static(obj, attr, default=None)

Retrieve attributes without triggering dynamic lookup via the descriptor protocol, __getattr__() or __getattribute__().

Note: this function may not be able to retrieve all attributes that getattr can fetch (like dynamically created attributes) and may find attributes that getattr can’t (like descriptors that raise AttributeError). It can also return descriptors objects instead of instance members.

If the instance __dict__ is shadowed by another member (for example a property) then this function will be unable to find instance members. New in version 3.2.

ggetattr_static() does not resolve descriptors, for example slot descriptors or getset descriptors on objects implemented in C. The descriptor object is returned instead of the underlying attribute.

You can handle these with code like the following. Note that for arbitrary getset descriptors invoking these may trigger code execution:
# example code for resolving the builtin descriptor types

class _foo:
    __slots__ = ['foo']

slot_descriptor = type(_foo.foo)
getset_descriptor = type(type(open(__file__).name))
wrapper_descriptor = type(str.__dict__['__add__'])
descriptor_types = (slot_descriptor, getset_descriptor, wrapper_descriptor)

result = getattr_static(some_object, 'foo')
if type(result) in descriptor_types:
    try:
        result = result.__get__()
    except AttributeError:
        # descriptors can raise AttributeError to
        # indicate there is no underlying value
        # in which case the descriptor itself will
        # have to do
        pass

28.12.7 Current State of a Generator

When implementing coroutine schedulers and for other advanced uses of generators, it is useful to determine
whether a generator is currently executing, is waiting to start or resume or execution, or has already terminated.
getgeneratorstate() allows the current state of a generator to be determined easily.

inspect.getgeneratorstate(generator)
Get current state of a generator-iterator.

Possible states are:

- GEN_CREATED: Waiting to start execution.
- GEN_RUNNING: Currently being executed by the interpreter.
- GEN_SUSPENDED: Currently suspended at a yield expression.
- GEN_CLOSED: Execution has completed.

New in version 3.2.

The current internal state of the generator can also be queried. This is mostly useful for testing purposes, to ensure
that internal state is being updated as expected:

inspect.getgeneratorlocals(generator)
Get the mapping of live local variables in generator to their current values. A dictionary is returned that
maps from variable names to values. This is the equivalent of calling locals() in the body of the gener-
ator, and all the same caveats apply.

If generator is a generator with no currently associated frame, then an empty dictionary is returned. 
TypeError is raised if generator is not a Python generator object.

CPython implementation detail: This function relies on the generator exposing a Python stack frame for
introspection, which isn’t guaranteed to be the case in all implementations of Python. In such cases, this
function will always return an empty dictionary. New in version 3.3.

28.13 site — Site-specific configuration hook

Source code: Lib/site.py
This module is automatically imported during initialization. The automatic import can be suppressed using the interpreter’s `-S` option.

Importing this module will append site-specific paths to the module search path and add a few builtins, unless `-S` was used. In that case, this module can be safely imported with no automatic modifications to the module search path or additions to the builtins. To explicitly trigger the usual site-specific additions, call the `site.main()` function. Changed in version 3.3: Importing the module used to trigger paths manipulation even when using `-S`. It starts by constructing up to four directories from a head and a tail part. For the head part, it uses `sys.prefix` and `sys.exec_prefix`; empty heads are skipped. For the tail part, it uses the empty string and then `lib/site-packages` (on Windows) or `lib/pythonX.Y/site-packages` and then `lib/site-python` (on Unix and Macintosh). For each of the distinct head-tail combinations, it sees if it refers to an existing directory, and if so, adds it to `sys.path` and also inspects the newly added path for configuration files.

If a file named “pyvenv.cfg” exists one directory above `sys.executable`, `sys.prefix` and `sys.exec_prefix` are set to that directory and it is also checked for site-packages and site-python (`sys.base_prefix` and `sys.base_exec_prefix` will always be the “real” prefixes of the Python installation). If “pyvenv.cfg” (a bootstrap configuration file) contains the key “include-system-site-packages” set to anything other than “false” (case-insensitive), the system-level prefixes will still also be searched for site-packages; otherwise they won’t.

A path configuration file is a file whose name has the form `name.pth` and exists in one of the four directories mentioned above; its contents are additional items (one per line) to be added to `sys.path`. Non-existing items are never added to `sys.path`, and no check is made that the item refers to a directory rather than a file. No item is added to `sys.path` more than once. Blank lines and lines beginning with `#` are skipped. Lines starting with `import` (followed by space or tab) are executed.

For example, suppose `sys.prefix` and `sys.exec_prefix` are set to `/usr/local`. The Python X.Y library is then installed in `/usr/local/lib/pythonX.Y`. Suppose this has a subdirectory `/usr/local/lib/pythonX.Y/site-packages` with three subdirectories, foo, bar and spam, and two path configuration files, foo.pth and bar.pth. Assume foo.pth contains the following:

```
# foo package configuration
foo
bar
bletch
```

and bar.pth contains:

```
# bar package configuration
bar
```

Then the following version-specific directories are added to `sys.path`, in this order:

```
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

Note that bletch is omitted because it doesn’t exist; the bar directory precedes the foo directory because bar.pth comes alphabetically before foo.pth; and spam is omitted because it is not mentioned in either path configuration file.

After these path manipulations, an attempt is made to import a module named `sitecustomize`, which can perform arbitrary site-specific customizations. It is typically created by a system administrator in the site-packages directory. If this import fails with an `ImportError` exception, it is silently ignored.

After this, an attempt is made to import a module named `usercustomize`, which can perform arbitrary user-specific customizations, if `ENABLE_USER_SITE` is true. This file is intended to be created in the user site-packages directory (see below), which is part of `sys.path` unless disabled by `-s`. An `ImportError` will be silently ignored.

Note that for some non-Unix systems, `sys.prefix` and `sys.exec_prefix` are empty, and the path manipulations are skipped; however the import of `sitecustomize` and `usercustomize` is still attempted.
site.PREFIXES
A list of prefixes for site-packages directories.

site.ENABLE_USER_SITE
Flag showing the status of the user site-packages directory. True means that it is enabled and was added to sys.path. False means that it was disabled by user request (with -s or PYTHONNOUSERSITE). None means it was disabled for security reasons (mismatch between user or group id and effective id) or by an administrator.

site.USER_SITE
Path to the user site-packages for the running Python. Can be None if getusersitepackages() hasn’t been called yet. Default value is ~/.local/lib/pythonX.Y/site-packages for UNIX and non-framework Mac OS X builds, ~/Library/Python/X.Y/lib/python/site-packages for Mac framework builds, and %APPDATA%\Python\PythonXY\site-packages on Windows. This directory is a site directory, which means that .pth files in it will be processed.

site.USER_BASE
Path to the base directory for the user site-packages. Can be None if getuserbase() hasn’t been called yet. Default value is ~/.local for UNIX and Mac OS X non-framework builds, ~/Library/Python/X.Y for Mac framework builds, and %APPDATA%\Python for Windows. This value is used by Distutils to compute the installation directories for scripts, data files, Python modules, etc. for the user installation scheme. See also PYTHONUSERBASE.

site.main()
Adds all the standard site-specific directories to the module search path. This function is called automatically when this module is imported, unless the python interpreter was started with the -S flag. Changed in version 3.3: This function used to be called unconditionally.

site.addsitedir(sitedir, known_paths=None)
Add a directory to sys.path and process its .pth files. Typically used in sitecustomize or usercustomize (see above).

site.getsitepackages()
Return a list containing all global site-packages directories (and possibly site-python). New in version 3.2.

site.getuserbase()
Return the path of the user base directory, USER_BASE. If it is not initialized yet, this function will also set it, respecting PYTHONUSERBASE. New in version 3.2.

site.getusersitepackages()
Return the path of the user-specific site-packages directory, USER_SITE. If it is not initialized yet, this function will also set it, respecting PYTHONNOUSERSITE and USER_BASE. New in version 3.2.

The site module also provides a way to get the user directories from the command line:

$ python3 -m site --user-site
/home/user/.local/lib/python3.3/site-packages

If it is called without arguments, it will print the contents of sys.path on the standard output, followed by the value of USER_BASE and whether the directory exists, then the same thing for USER_SITE, and finally the value of ENABLE_USER_SITE.

--user-base
Print the path to the user base directory.

--user-site
Print the path to the user site-packages directory.

If both options are given, user base and user site will be printed (always in this order), separated by os.pathsep.

If any option is given, the script will exit with one of these values: 0 if the user site-packages directory is enabled, 1 if it was disabled by the user, 2 if it is disabled for security reasons or by an administrator, and a value greater than 2 if there is an error.
28.14 fpectl — Floating point exception control

Platforms: Unix

Note: The fpectl module is not built by default, and its usage is discouraged and may be dangerous except in the hands of experts. See also the section Limitations and other considerations on limitations for more details.

Most computers carry out floating point operations in conformance with the so-called IEEE-754 standard. On any real computer, some floating point operations produce results that cannot be expressed as a normal floating point value. For example, try

```python
>>> import math
>>> math.exp(1000)
inf
>>> math.exp(1000) / math.exp(1000)
nan
```

(The example above will work on many platforms. DEC Alpha may be one exception.) “Inf” is a special, non-numeric value in IEEE-754 that stands for “infinity”, and “nan” means “not a number.” Note that, other than the non-numeric results, nothing special happened when you asked Python to carry out those calculations. That is in fact the default behaviour prescribed in the IEEE-754 standard, and if it works for you, stop reading now.

In some circumstances, it would be better to raise an exception and stop processing at the point where the faulty operation was attempted. The fpectl module is for use in that situation. It provides control over floating point units from several hardware manufacturers, allowing the user to turn on the generation of SIGFPE whenever any of the IEEE-754 exceptions Division by Zero, Overflow, or Invalid Operation occurs. In tandem with a pair of wrapper macros that are inserted into the C code comprising your Python system, SIGFPE is trapped and converted into the Python FloatingPointError exception.

The fpectl module defines the following functions and may raise the given exception:

```python
fpectl.turnon_sigfpe()
    Turn on the generation of SIGFPE, and set up an appropriate signal handler.

fpectl.turnoff_sigfpe()
    Reset default handling of floating point exceptions.
```

exception fpectl.FloatingPointError
    After turnon_sigfpe() has been executed, a floating point operation that raises one of the IEEE-754 exceptions Division by Zero, Overflow, or Invalid Operation will in turn raise this standard Python exception.

28.14.1 Example

The following example demonstrates how to start up and test operation of the fpectl module.

```python
>>> import fpectl
>>> import fpetest
>>> fpectl.turnon_sigfpe()
>>> fpetest.test()
overflow   PASS
FloatingPointError: Overflow

div by 0   PASS
FloatingPointError: Division by zero
    [ more output from test elided ]
```
>>> import math
>>> math.exp(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
FloatingPointError: in math_1

28.14.2 Limitations and other considerations

Setting up a given processor to trap IEEE-754 floating point errors currently requires custom code on a per-architecture basis. You may have to modify fpectl to control your particular hardware.

Conversion of an IEEE-754 exception to a Python exception requires that the wrapper macros PyFPE_START_PROTECT and PyFPE_END_PROTECT be inserted into your code in an appropriate fashion. Python itself has been modified to support the fpectl module, but many other codes of interest to numerical analysts have not.

The fpectl module is not thread-safe.

See Also:

Some files in the source distribution may be interesting in learning more about how this module operates. The include file Include/pyfpe.h discusses the implementation of this module at some length. Modules/fpetestmodule.c gives several examples of use. Many additional examples can be found in Objects/floatobject.c.

28.15 distutils — Building and installing Python modules

The distutils package provides support for building and installing additional modules into a Python installation. The new modules may be either 100%-pure Python, or may be extension modules written in C, or may be collections of Python packages which include modules coded in both Python and C.

User documentation and API reference are provided in another document:

See Also:

distutils-index The manual for developers and packagers of Python modules. This describes how to prepare distutils-based packages so that they may be easily installed into an existing Python installation. It also contains instructions for end-users wanting to install a distutils-based package, install-index.
The modules described in this chapter allow writing interfaces similar to Python’s interactive interpreter. If you want a Python interpreter that supports some special feature in addition to the Python language, you should look at the `code` module. (The `codeop` module is lower-level, used to support compiling a possibly-incomplete chunk of Python code.)

The full list of modules described in this chapter is:

### 29.1 code — Interpreter base classes

The `code` module provides facilities to implement read-eval-print loops in Python. Two classes and convenience functions are included which can be used to build applications which provide an interactive interpreter prompt.

**class code.InteractiveInterpreter**( `locals=None` )

This class deals with parsing and interpreter state (the user’s namespace); it does not deal with input buffering or prompting or input file naming (the filename is always passed in explicitly). The optional `locals` argument specifies the dictionary in which code will be executed; it defaults to a newly created dictionary with key ‘__name__’ set to ‘__console__’ and key ‘__doc__’ set to None.

**class code.InteractiveConsole**( `locals=None, filename=<console>`)  

Closely emulate the behavior of the interactive Python interpreter. This class builds on `InteractiveInterpreter` and adds prompting using the familiar `sys.ps1` and `sys.ps2`, and input buffering.

**code.interact**( `banner=None, readfunc=None, local=None`)  

Convenience function to run a read-eval-print loop. This creates a new instance of `InteractiveConsole` and sets `readfunc` to be used as the `InteractiveConsole.raw_input()` method, if provided. If `local` is provided, it is passed to the `InteractiveConsole` constructor for use as the default namespace for the interpreter loop. The `interact()` method of the instance is then run with `banner` passed as the banner to use, if provided. The console object is discarded after use.

**code.compile_command**( `source, filename=<input>”, symbol=”single”`)  

This function is useful for programs that want to emulate Python’s interpreter main loop (a.k.a. the read-eval-print loop). The tricky part is to determine when the user has entered an incomplete command that can be completed by entering more text (as opposed to a complete command or a syntax error). This function almost always makes the same decision as the real interpreter main loop.

`source` is the source string; `filename` is the optional filename from which source was read, defaulting to ‘<input>’; and `symbol` is the optional grammar start symbol, which should be either ‘single’ (the default) or ‘eval’.

Returns a code object (the same as `compile(source, filename, symbol)`) if the command is complete and valid; None if the command is incomplete; raises `SyntaxError` if the command is complete and contains a syntax error, or raises `OverflowError` or `ValueError` if the command contains an invalid literal.
29.1.1 Interactive Interpreter Objects

`InteractiveInterpreter.runsource(source, filename="<input>", symbol="single")`

Compile and run some source in the interpreter. Arguments are the same as for `compile_command()`: the default for `filename` is `'<input>'`, and for `symbol` is `single`. One several things can happen:

- The input is incorrect; `compile_command()` raised an exception (`SyntaxError` or `OverflowError`). A syntax traceback will be printed by calling the `showsyntaxerror()` method. `runsource()` returns `False`.

- The input is incomplete, and more input is required; `compile_command()` returned `None`. `runsource()` returns `True`.

- The input is complete; `compile_command()` returned a code object. The code is executed by calling the `runcode()` (which also handles run-time exceptions, except for `SystemExit`). `runsource()` returns `False`.

The return value can be used to decide whether to use `sys.ps1` or `sys.ps2` to prompt the next line.

`InteractiveInterpreter.runcode(code)`

Execute a code object. When an exception occurs, `showtraceback()` is called to display a traceback. All exceptions are caught except `SystemExit`, which is allowed to propagate.

A note about `KeyboardInterrupt`: this exception may occur elsewhere in this code, and may not always be caught. The caller should be prepared to deal with it.

`InteractiveInterpreter.showsyntaxerror(filename=None)`

Display the syntax error that just occurred. This does not display a stack trace because there isn’t one for syntax errors. If `filename` is given, it is stuffed into the exception instead of the default filename provided by Python’s parser, because it always uses `<string>` when reading from a string. The output is written by the `write()` method.

`InteractiveInterpreter.showtraceback()`

Display the exception that just occurred. We remove the first stack item because it is within the interpreter object implementation. The output is written by the `write()` method.

`InteractiveInterpreter.write(data)`

Write a string to the standard error stream (`sys.stderr`). Derived classes should override this to provide the appropriate output handling as needed.

29.1.2 Interactive Console Objects

The `InteractiveConsole` class is a subclass of `InteractiveInterpreter`, and so offers all the methods of the interpreter objects as well as the following additions.

`InteractiveConsole.interact(banner=None)`

Closely emulate the interactive Python console. The optional banner argument specify the banner to print before the first interaction; by default it prints a banner similar to the one printed by the standard Python interpreter, followed by the class name of the console object in parentheses (so as not to confuse this with the real interpreter – since it’s so close!).

`InteractiveConsole.push(line)`

Push a line of source text to the interpreter. The line should not have a trailing newline; it may have internal newlines. The line is appended to a buffer and the interpreter's `runsource()` method is called with the concatenated contents of the buffer as source. If this indicates that the command was executed or invalid, the buffer is reset; otherwise, the command is incomplete, and the buffer is left as it was after the line was appended. The return value is `True` if more input is required, `False` if the line was dealt with in some way (this is the same as `runsource()`).

`InteractiveConsole.resetbuffer()`

Remove any unhandled source text from the input buffer.

`InteractiveConsole.raw_input(prompt="")`

Write a prompt and read a line. The returned line does not include the trailing newline. When the user
enters the EOF key sequence, \texttt{EOFError} is raised. The base implementation reads from \texttt{sys.stdin}; a subclass may replace this with a different implementation.

\section*{29.2 \texttt{codeop} — Compile Python code}

The \texttt{codeop} module provides utilities upon which the Python read-eval-print loop can be emulated, as is done in the \texttt{code} module. As a result, you probably don’t want to use the module directly; if you want to include such a loop in your program you probably want to use the \texttt{code} module instead.

There are two parts to this job:

1. Being able to tell if a line of input completes a Python statement: in short, telling whether to print ‘\texttt{>>>}\’ or ‘\texttt{...}\’ next.
2. Remembering which future statements the user has entered, so subsequent input can be compiled with these in effect.

The \texttt{codeop} module provides a way of doing each of these things, and a way of doing them both.

To do just the former:

\begin{verbatim}
\texttt{codeop.compile_command(source, filename="<input>", symbol="single")}
\end{verbatim}

Tries to compile \texttt{source}, which should be a string of Python code and return a code object if \texttt{source} is valid Python code. In that case, the filename attribute of the code object will be \texttt{filename}, which defaults to ‘\texttt{<input>}’. Returns \texttt{None} if \texttt{source} is not valid Python code, but is a prefix of valid Python code.

If there is a problem with \texttt{source}, an exception will be raised. \texttt{SyntaxError} is raised if there is invalid Python syntax, and \texttt{OverflowError} or \texttt{ValueError} if there is an invalid literal.

The \texttt{symbol} argument determines whether \texttt{source} is compiled as a statement (‘\texttt{single}’, the default) or as an expression (‘\texttt{eval}’). Any other value will cause \texttt{ValueError} to be raised.

\begin{note}
It is possible (but not likely) that the parser stops parsing with a successful outcome before reaching the end of the source; in this case, trailing symbols may be ignored instead of causing an error. For example, a backslash followed by two newlines may be followed by arbitrary garbage. This will be fixed once the API for the parser is better.
\end{note}

\begin{class}
\texttt{codeop.Compile}

Instances of this class have \texttt{__call__()} methods identical in signature to the built-in function \texttt{compile()}, but with the difference that if the instance compiles program text containing a \texttt{__future__} statement, the instance ‘remembers’ and compiles all subsequent program texts with the statement in force.
\end{class}

\begin{class}
\texttt{codeop.CommandCompiler}

Instances of this class have \texttt{__call__()} methods identical in signature to \texttt{compile_command()}; the difference is that if the instance compiles program text containing a \texttt{__future__} statement, the instance ‘remembers’ and compiles all subsequent program texts with the statement in force.
\end{class}
The modules described in this chapter provide new ways to import other Python modules and hooks for customizing the import process.

The full list of modules described in this chapter is:

30.1 imp — Access the import internals

This module provides an interface to the mechanisms used to implement the import statement. It defines the following constants and functions:

Note: New programs should use importlib rather than this module.

- imp.get_magic()  
  Return the magic string value used to recognize byte-compiled code files (.pyc files). (This value may be different for each Python version.)

- imp.get_suffixes()  
  Return a list of 3-element tuples, each describing a particular type of module. Each triple has the form (suffix, mode, type), where suffix is a string to be appended to the module name to form the filename to search for, mode is the mode string to pass to the built-in open() function to open the file (this can be 'r' for text files or 'rb' for binary files), and type is the file type, which has one of the values PY_SOURCE, PY_COMPILED, or C_EXTENSION, described below. Deprecated since version 3.3: Use the constants defined on importlib.machinery instead.

- imp.find_module(name[, path])  
  Try to find the module name. If path is omitted or None, the list of directory names given by sys.path is searched, but first a few special places are searched: the function tries to find a built-in module with the given name (C_BUILTIN), then a frozen module (PY_FROZEN), and on some systems some other places are looked in as well (on Windows, it looks in the registry which may point to a specific file).

  Otherwise, path must be a list of directory names; each directory is searched for files with any of the suffixes returned by get_suffixes() above. Invalid names in the list are silently ignored (but all list items must be strings).

  If search is successful, the return value is a 3-element tuple (file, pathname, description): file is an open file object positioned at the beginning, pathname is the pathname of the file found, and description is a 3-element tuple as contained in the list returned by get_suffixes() describing the kind of module found.

  If the module does not live in a file, the returned file is None, pathname is the empty string, and the description tuple contains empty strings for its suffix and mode; the module type is indicated as given in parentheses above. If the search is unsuccessful, ImportError is raised. Other exceptions indicate problems with the arguments or environment.
If the module is a package, file is None, pathname is the package path and the last item in the description tuple is PKG_DIRECTORY.

This function does not handle hierarchical module names (names containing dots). In order to find \( P.M \), that is, submodule \( M \) of package \( P \), use \texttt{find\_module()} and \texttt{load\_module()} to find and load package \( P \), and then use \texttt{find\_module()} with the \texttt{path} argument set to \( P._\text{\texttt{__path__}} \). When \( P \) itself has a dotted name, apply this recipe recursively. Deprecated since version 3.3: Use \texttt{importlib.find\_loader()} instead.

\texttt{imp.load\_module(name, file, pathname, description)}

Load a module that was previously found by \texttt{find\_module()} (or by an otherwise conducted search yielding compatible results). This function does more than importing the module: if the module was already imported, it will reload the module! The name argument indicates the full module name (including the package name, if this is a submodule of a package). The file argument is an open file, and pathname is the corresponding file name; these can be None and “”, respectively, when the module is a package or not being loaded from a file. The description argument is a tuple, as would be returned by \texttt{get\_suffixes()}, describing what kind of module must be loaded.

If the load is successful, the return value is the module object; otherwise, an exception (usually \texttt{ImportError}) is raised.

\textbf{Important:} the caller is responsible for closing the file argument, if it was not None, even when an exception is raised. This is best done using a try ... finally statement. Deprecated since version 3.3: Unneeded as loaders should be used to load modules and \texttt{find\_module()} is deprecated.

\texttt{imp.new\_module(name)}

Return a new empty module object called name. This object is not inserted in \texttt{sys.modules}.

\texttt{imp.reload(module)}

Reload a previously imported module. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (the same as the module argument).

When \texttt{reload(module)} is executed:

- Python modules’ code is recompiled and the module-level code reexecuted, defining a new set of objects which are bound to names in the module’s dictionary. The \texttt{init} function of extension modules is not called a second time.
- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.
- The names in the module namespace are updated to point to any new or changed objects.
- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats:

If a module is syntactically correct but its initialization fails, the first import statement for it does not bind its name locally, but does store a (partially initialized) module object in \texttt{sys.modules}. To reload the module you must first import it again (this will bind the name to the partially initialized module object) before you can \texttt{reload()} it.

When a module is reloaded, its dictionary (containing the module’s global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module’s advantage if it maintains a global table or cache of objects — with a try statement it can test for the table’s presence and skip its initialization if desired:

\texttt{try:}
\hspace{1em}cache
\texttt{except NameError:}
\hspace{1em}cache = {}
It is legal though generally not very useful to reload built-in or dynamically loaded modules, except for `sys`, `__main__`, and `builtins`. In many cases, however, extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from ... import ...`, calling `reload()` for the other module does not redefine the objects imported from it — one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (`module.*name*`) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes. Changed in version 3.3: Relies on both `__name__` and `__loader__` being defined on the module being reloaded instead of just `__name__`.

The following functions are conveniences for handling PEP 3147 byte-compiled file paths. New in version 3.2.

```python
imp.cache_from_source(path, debug_override=None)
```

Return the PEP 3147 path to the byte-compiled file associated with the source `path`. For example, if `path` is `/foo/bar/baz.py` the return value would be `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. The `cpython-32` string comes from the current magic tag (see `get_tag()`); if `sys.implementation.cache_tag` is not defined then `NotImplementedError` will be raised. The returned path will end in `.pyc` when `__debug__` is True or `.pyo` for an optimized Python (i.e. `__debug__` is False). By passing in True or False for `debug_override` you can override the system’s value for `__debug__` for extension selection.

`path` need not exist. Changed in version 3.3: If `sys.implementation.cache_tag` is None, then `NotImplementedError` is raised.

```python
imp.source_from_cache(path)
```

Given the `path` to a PEP 3147 file name, return the associated source code file path. For example, if `path` is `/foo/bar/__pycache__/baz.cpython-32.pyc` the returned path would be `/foo/bar/baz.py`. `path` need not exist, however if it does not conform to PEP 3147 format, a `ValueError` is raised. If `sys.implementation.cache_tag` is not defined, `NotImplementedError` is raised. Changed in version 3.3: Raise `NotImplementedError` when `sys.implementation.cache_tag` is not defined.

```python
imp.get_tag()
```

Return the PEP 3147 magic tag string matching this version of Python’s magic number, as returned by `get_magic()`.

**Note:** You may use `sys.implementation.cache_tag` directly starting in Python 3.3.

The following functions help interact with the import system’s internal locking mechanism. Locking semantics of imports are an implementation detail which may vary from release to release. However, Python ensures that circular imports work without any deadlocks.

```python
imp.lock_held()
```

Return `True` if the global import lock is currently held, else `False`. On platforms without threads, always return `False`.

On platforms with threads, a thread executing an import first holds a global import lock, then sets up a per-module lock for the rest of the import. This blocks other threads from importing the same module until the original import completes, preventing other threads from seeing incomplete module objects constructed by the original thread. An exception is made for circular imports, which by construction have to expose an incomplete module object at some point.

Changed in version 3.3: The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

```python
imp.acquire_lock()
```

Acquire the interpreter’s global import lock for the current thread. This lock should be used by import hooks to ensure thread-safety when importing modules.
Once a thread has acquired the import lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

On platforms without threads, this function does nothing.

Changed in version 3.3: The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

```python
imp.release_lock()
```

Release the interpreter’s global import lock. On platforms without threads, this function does nothing.

Changed in version 3.3: The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks. The following constants with integer values, defined in this module, are used to indicate the search result of `find_module()`.

- `imp.PY_SOURCE`: The module was found as a source file. Deprecated since version 3.3.
- `imp.PY_COMPILED`: The module was found as a compiled code object file. Deprecated since version 3.3.
- `imp.C_EXTENSION`: The module was found as dynamically loadable shared library. Deprecated since version 3.3.
- `imp.PKG_DIRECTORY`: The module was found as a package directory. Deprecated since version 3.3.
- `imp.C_BUILTIN`: The module was found as a built-in module. Deprecated since version 3.3.
- `imp.PY_FROZEN`: The module was found as a frozen module. Deprecated since version 3.3.

```python
class imp.NullImporter(path_string):
```

The `NullImporter` type is a PEP 302 import hook that handles non-directory path strings by failing to find any modules. Calling this type with an existing directory or empty string raises `ImportError`. Otherwise, a `NullImporter` instance is returned.

Instances have only one method:

```python
find_module(fullname[, path])
```

This method always returns `None`, indicating that the requested module could not be found.

Changed in version 3.3: `None` is inserted into `sys.path_importer_cache` instead of an instance of `NullImporter`.

### 30.1.1 Examples

The following function emulates what was the standard import statement up to Python 1.4 (no hierarchical module names). (This implementation wouldn’t work in that version, since `find_module()` has been extended and `load_module()` has been added in 1.4.)

```python
import imp
import sys

def __import__(name, globals=None, locals=None, fromlist=None):
    # Fast path: see if the module has already been imported.
    try:
        return sys.modules[name]
    except KeyError:
        pass

    # If any of the following calls raises an exception,
    # there’s a problem we can’t handle -- let the caller handle it.
```
fp, pathname, description = imp.find_module(name)

try:
    return imp.load_module(name, fp, pathname, description)
finally:
    # Since we may exit via an exception, close fp explicitly.
    if fp:
        fp.close()

30.2 zipimport — Import modules from Zip archives

This module adds the ability to import Python modules (*.py, *.py[co]) and packages from ZIP-format archives. It is usually not needed to use the zipimport module explicitly; it is automatically used by the built-in import mechanism for sys.path items that are paths to ZIP archives.

Typically, sys.path is a list of directory names as strings. This module also allows an item of sys.path to be a string naming a ZIP file archive. The ZIP archive can contain a subdirectory structure to support package imports, and a path within the archive can be specified to only import from a subdirectory. For example, the path example.zip/lib/ would only import from the lib/ subdirectory within the archive.

Any files may be present in the ZIP archive, but only files .py and .py[co] are available for import. ZIP import of dynamic modules (.pyd, .so) is disallowed. Note that if an archive only contains .py files, Python will not attempt to modify the archive by adding the corresponding .pyc or .pyo file, meaning that if a ZIP archive doesn’t contain .pyc files, importing may be rather slow.

ZIP archives with an archive comment are currently not supported.

See Also:
PKZIP Application Note Documentation on the ZIP file format by Phil Katz, the creator of the format and algorithms used.

PEP 273 - Import Modules from Zip Archives Written by James C. Ahlstrom, who also provided an implementation. Python 2.3 follows the specification in PEP 273, but uses an implementation written by Just van Rossum that uses the import hooks described in PEP 302.

PEP 302 - New Import Hooks The PEP to add the import hooks that help this module work.

This module defines an exception:

definition zipimport.ZipImportError
    Exception raised by zipimporter objects. It’s a subclass of ImportError, so it can be caught as ImportError, too.

30.2.1 zipimporter Objects

zipimporter is the class for importing ZIP files.

class zipimporter.zipimporter (archivepath)
    Create a new zipimporter instance. archivepath must be a path to a ZIP file, or to a specific path within a ZIP file. For example, an archivepath of foo/bar.zip/lib will look for modules in the lib directory inside the ZIP file foo/bar.zip (provided that it exists).

    ZipImportError is raised if archivepath doesn’t point to a valid ZIP archive.

    find_module (fullname [, path ])
    Search for a module specified by fullname. fullname must be the fully qualified (dotted) module name. It returns the zipimporter instance itself if the module was found, or None if it wasn’t. The optional path argument is ignored—it’s there for compatibility with the importer protocol.
**get_code**(fullname)
Return the code object for the specified module. Raise **ZipImportError** if the module couldn’t be found.

**get_data**(pathname)
Return the data associated with pathname. Raise **OSError** if the file wasn’t found. Changed in version 3.3: **IOError** used to be raised instead of **OSError**.

**get_filename**(fullname)
Return the value **_file_** would be set to if the specified module was imported. Raise **ZipImportError** if the module couldn’t be found. New in version 3.1.

**get_source**(fullname)
Return the source code for the specified module. Raise **ZipImportError** if the module couldn’t be found, return **None** if the archive does contain the module, but has no source for it.

**is_package**(fullname)
Return True if the module specified by fullname is a package. Raise **ZipImportError** if the module couldn’t be found.

**load_module**(fullname)
Load the module specified by fullname. **fullname** must be the fully qualified (dotted) module name. It returns the imported module, or raises **ZipImportError** if it wasn’t found.

**archive**
The file name of the importer’s associated ZIP file, without a possible subpath.

**prefix**
The subpath within the ZIP file where modules are searched. This is the empty string for zipimporter objects which point to the root of the ZIP file.

The **archive** and **prefix** attributes, when combined with a slash, equal the original **archivepath** argument given to the **zipimporter** constructor.

### 30.2.2 Examples

Here is an example that imports a module from a ZIP archive - note that the **zipimport** module is not explicitly used.

```
$ unzip -l example.zip
Archive: example.zip
Length  Date    Time   Name
-------- ----      ----      ----
8467     11-26-02 22:30  jwzthreading.py
-------- -------
8467 1 file
$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, ‘example.zip’)  # Add .zip file to front of path
>>> import jwzthreading
>>> jwzthreading.__file__
‘example.zip/jwzthreading.py’
```

### 30.3 pkgutil — Package extension utility

**Source code:** Lib/pkgutil.py

This module provides utilities for the import system, in particular package support.
pkgutil.extend_path(path, name)

Extend the search path for the modules which comprise a package. Intended use is to place the following code in a package’s __init__.py:

```python
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

This will add to the package’s __path__ all subdirectories of directories on sys.path named after the package. This is useful if one wants to distribute different parts of a single logical package as multiple directories.

It also looks for *.pkg files beginning where * matches the name argument. This feature is similar to *.pth files (see the site module for more information), except that it doesn’t special-case lines starting with import. A *.pkg file is trusted at face value: apart from checking for duplicates, all entries found in a *.pkg file are added to the path, regardless of whether they exist on the filesystem. (This is a feature.)

If the input path is not a list (as is the case for frozen packages) it is returned unchanged. The input path is not modified; an extended copy is returned. Items are only appended to the copy at the end.

It is assumed that sys.path is a sequence. Items of sys.path that are not strings referring to existing directories are ignored. Unicode items on sys.path that cause errors when used as filenames may cause this function to raise an exception (in line with os.path.isdir() behavior).

class pkgutil.ImpImporter(dirname=None)

PEP 302 Importer that wraps Python’s “classic” import algorithm.

If dirname is a string, a PEP 302 importer is created that searches that directory. If dirname is None, a PEP 302 importer is created that searches the current sys.path, plus any modules that are frozen or built-in.

Note that ImpImporter does not currently support being used by placement on sys.meta_path. Deprecated since version 3.3: This emulation is no longer needed, as the standard import mechanism is now fully PEP 302 compliant and available in importlib

class pkgutil.ImpLoader(fullname, file, filename, etc)

PEP 302 Loader that wraps Python’s “classic” import algorithm. Deprecated since version 3.3: This emulation is no longer needed, as the standard import mechanism is now fully PEP 302 compliant and available in importlib

class pkgutil.find_loader(fullname)

Retrieve a PEP 302 module loader for the given fullname.

This is a convenience wrapper around importlib.find_loader() that sets the path argument correctly when searching for submodules, and also ensures parent packages (if any) are imported before searching for submodules. Changed in version 3.3: Updated to be based directly on importlib rather than relying on the package internal PEP 302 import emulation.

class pkgutil.get_importer(path_item)

Retrieve a PEP 302 importer for the given path_item.

The returned importer is cached in sys.path_importer_cache if it was newly created by a path hook.

The cache (or part of it) can be cleared manually if a rescan of sys.path_hooks is necessary. Changed in version 3.3: Updated to be based directly on importlib rather than relying on the package internal PEP 302 import emulation.

class pkgutil.get_loader(module_or_name)

Get a PEP 302 “loader” object for module_or_name.

If the module or package is accessible via the normal import mechanism, a wrapper around the relevant part of that machinery is returned. Returns None if the module cannot be found or imported. If the named module is not already imported, its containing package (if any) is imported, in order to establish the package __path__.

This function uses iter_importers(), and is thus subject to the same limitations regarding platform-specific special import locations such as the Windows registry. Changed in version 3.3: Updated to be based directly on importlib rather than relying on the package internal PEP 302 import emulation.
pkgutil.iter_importers(fullname='')

Yield PEP 302 importers for the given module name.

If fullname contains a '.', the importers will be for the package containing fullname, otherwise they will be all registered top level importers (i.e. those on both sys.meta_path and sys.path_hooks).

If the named module is in a package, that package is imported as a side effect of invoking this function.

If no module name is specified, all top level importers are produced. Changed in version 3.3: Updated to be based directly on importlib rather than relying on the package internal PEP 302 import emulation.

pkgutil.iter_modules(path=None, prefix='')

Yields (module_finder, name, ispkg) for all submodules on path, or, if path is None, all top-level modules on sys.path.

path should be either None or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

Note: Only works for a finder which defines an iter_modules() method. This interface is non-standard, so the module also provides implementations for importlib.machinery.FileFinder and zipimport.zipimporter.

Changed in version 3.3: Updated to be based directly on importlib rather than relying on the package internal PEP 302 import emulation.

pkgutil.walk_packages(path=None, prefix='.', onerror=None)

Yields (module_finder, name, ispkg) for all modules recursively on path, or, if path is None, all accessible modules.

path should be either None or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

Note that this function must import all packages (not all modules!) on the given path, in order to access the __path__ attribute to find submodules.

onerror is a function which gets called with one argument (the name of the package which was being imported) if any exception occurs while trying to import a package. If no onerror function is supplied, ImportError errors are caught and ignored, while all other exceptions are propagated, terminating the search.

Examples:

```python
# list all modules python can access
walk_packages()

# list all submodules of ctypes
walk_packages(ctypes.__path__, ctypes.__name__ + '.')
```

Note: Only works for a finder which defines an iter_modules() method. This interface is non-standard, so the module also provides implementations for importlib.machinery.FileFinder and zipimport.zipimporter.

Changed in version 3.3: Updated to be based directly on importlib rather than relying on the package internal PEP 302 import emulation.

pkgutil.get_data(package, resource)

Get a resource from a package.

This is a wrapper for the PEP 302 loader get_data() API. The package argument should be the name of a package, in standard module format (foo.bar). The resource argument should be in the form of a relative filename, using / as the path separator. The parent directory name .. is not allowed, and nor is a rooted name (starting with a /).
The function returns a binary string that is the contents of the specified resource.

For packages located in the filesystem, which have already been imported, this is the rough equivalent of:

```python
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()
```

If the package cannot be located or loaded, or it uses a PEP 302 loader which does not support `get_data()`, then `None` is returned.

### 30.4 modulefinder — Find modules used by a script

**Source code:** Lib/modulefinder.py

This module provides a `ModuleFinder` class that can be used to determine the set of modules imported by a script. `modulefinder.py` can also be run as a script, giving the filename of a Python script as its argument, after which a report of the imported modules will be printed.

#### `modulefinder.AddPackagePath(pkg_name, path)`
- Record that the package named `pkg_name` can be found in the specified `path`.

#### `modulefinder.ReplacePackage(oldname, newname)`
- Allows specifying that the module named `oldname` is in fact the package named `newname`.

```python
class modulefinder.ModuleFinder(path=None, debug=0, excludes=[], replace_paths=[]):
    This class provides `run_script()` and `report()` methods to determine the set of modules imported by a script. `path` can be a list of directories to search for modules; if not specified, `sys.path` is used. `debug` sets the debugging level; higher values make the class print debugging messages about what it’s doing. `excludes` is a list of module names to exclude from the analysis. `replace_paths` is a list of `(oldpath, newpath)` tuples that will be replaced in module paths.

    `report()`
    - Print a report to standard output that lists the modules imported by the script and their paths, as well as modules that are missing or seem to be missing.

    `run_script(pathname)`
    - Analyze the contents of the `pathname` file, which must contain Python code.

    `modules`
    - A dictionary mapping module names to modules. See Example usage of ModuleFinder

### 30.4.1 Example usage of ModuleFinder

The script that is going to get analyzed later on (bacon.py):

```python
import re, itertools

try:  
    import baconhameggs
except ImportError:  
    pass

try:  
    import guido.python.ham
except ImportError:  
    pass
```

The script that will output the report of bacon.py:
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print('Loaded modules:')
for name, mod in finder.modules.items():
    print(' %s: ' % name, end='')
    print(','.join(list(mod.globalnames.keys())[:3]))

print('-'*50)
print('Modules not imported: ')
print('\n'.join(finder.badmodules.keys()))

Sample output (may vary depending on the architecture):

Loaded modules:
_types: _types
copyreg: __inverted_registry, _slotnames, __all__
sre_compile: isstring, _sre, _optimize_unicode
_sre: _sre_constants: REPEAT_ONE, makedict, AT_END_LINE
sys: re: __module__, finditer, _expand
 itertools: __main__: re, itertools, baconhameggs
sre_parse: __getslice__, _PATTERN_ENDERS, SRE_FLAG_UNICODE
 array: types: __module__, IntType, TypeError

---------------------------------------------------
Modules not imported: 
guido.python.ham
baconhameggs

30.5 runpy — Locating and executing Python modules

Source code: Lib/runpy.py

The runpy module is used to locate and run Python modules without importing them first. Its main use is to
implement the -m command line switch that allows scripts to be located using the Python module namespace
rather than the filesystem.

Note that this is not a sandbox module - all code is executed in the current process, and any side effects (such as
cached imports of other modules) will remain in place after the functions have returned.

Furthermore, any functions and classes defined by the executed code are not guaranteed to work correctly after a
runpy function has returned. If that limitation is not acceptable for a given use case, importlib is likely to be
a more suitable choice than this module.

The runpy module provides two functions:

runpy.run_module (mod_name, init_globals=None, run_name=None, alter_sys=False)

Execute the code of the specified module and return the resulting module globals dictionary. The module’s
code is first located using the standard import mechanism (refer to PEP 302 for details) and then executed
in a fresh module namespace.

If the supplied module name refers to a package rather than a normal module, then that package is imported
and the __main__ submodule within that package is then executed and the resulting module globals dic-
tionary returned.
The optional dictionary argument `initGlobals` may be used to pre-populate the module’s globals dictionary before the code is executed. The supplied dictionary will not be modified. If any of the special global variables below are defined in the supplied dictionary, those definitions are overridden by `run_module()`.

The special global variables `__name__`, `__file__`, `__cached__`, `__loader__` and `__package__` are set in the globals dictionary before the module code is executed (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail).

`__name__` is set to `run_name` if this optional argument is not `None`, to `mod_name + '.' + __main__` if the named module is a package and to the `mod_name` argument otherwise.

`__file__` is set to the name provided by the module loader. If the loader does not make filename information available, this variable is set to `None`.

`__cached__` will be set to `None`.

`__loader__` is set to the PEP 302 module loader used to retrieve the code for the module (This loader may be a wrapper around the standard import mechanism).

`__package__` is set to `mod_name` if the named module is a package and to `mod_name.rpartition('.')[0]` otherwise.

If the argument `alterSys` is supplied and evaluates to `True`, then `sys.argv[0]` is updated with the value of `__file__` and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. Both `sys.argv[0]` and `sys.modules[__name__]` are restored to their original values before the function returns.

Note that this manipulation of `sys` is not thread-safe. Other threads may see the partially initialised module, as well as the altered list of arguments. It is recommended that the `sys` module be left alone when invoking this function from threaded code. Changed in version 3.1: Added ability to execute packages by looking for a `__main__` submodule. Changed in version 3.2: Added `__cached__` global variable (see PEP 3147).

```
runpy.run_path(file_path, init_globals=None, run_name=None)
```

Execute the code at the named filesystem location and return the resulting module globals dictionary. As with a script name supplied to the CPython command line, the supplied path may refer to a Python source file, a compiled bytecode file or a valid `sys.path` entry containing a `__main__` module (e.g. a zipfile containing a top-level `__main__.py` file).

For a simple script, the specified code is simply executed in a fresh module namespace. For a valid `sys.path` entry (typically a zipfile or directory), the entry is first added to the beginning of `sys.path`. The function then looks for and executes a `__main__` module using the updated path. Note that there is no special protection against invoking an existing `__main__` entry located elsewhere on `sys.path` if there is no such module at the specified location.

The optional dictionary argument `initGlobals` may be used to pre-populate the module’s globals dictionary before the code is executed. The supplied dictionary will not be modified. If any of the special global variables below are defined in the supplied dictionary, those definitions are overridden by `run_path()`.

The special global variables `__name__`, `__file__`, `__loader__` and `__package__` are set in the globals dictionary before the module code is executed (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail).

`__name__` is set to `run_name` if this optional argument is not `None` and to `<run_path>`’ otherwise.

`__file__` is set to the name provided by the module loader. If the loader does not make filename information available, this variable is set to `None`. For a simple script, this will be set to `file_path`.

`__loader__` is set to the PEP 302 module loader used to retrieve the code for the module (This loader may be a wrapper around the standard import mechanism). For a simple script, this will be set to `None`.

`__package__` is set to `__name__.rpartition('.')[0]`.

A number of alterations are also made to the `sys` module. Firstly, `sys.path` may be altered as described above. `sys.argv[0]` is updated with the value of `file_path` and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. All modifications to items in `sys` are reverted before the function returns.
Note that, unlike `run_module()`, the alterations made to `sys` are not optional in this function as these adjustments are essential to allowing the execution of `sys.path` entries. As the thread-safety limitations still apply, use of this function in threaded code should be either serialised with the import lock or delegated to a separate process. New in version 3.2.

See Also:

- **PEP 338** - Executing modules as scripts  PEP written and implemented by Nick Coghlan.
- **PEP 366** - Main module explicit relative imports  PEP written and implemented by Nick Coghlan.

30.6 *importlib* – An implementation of *import*

New in version 3.1.

30.6.1 Introduction

The purpose of the *importlib* package is two-fold. One is to provide an implementation of the *import* statement (and thus, by extension, the `__import__()` function) in Python source code. This provides an implementation of *import* which is portable to any Python interpreter. This also provides a reference implementation which is easier to comprehend than one implemented in a programming language other than Python.

Two, the components to implement *import* are exposed in this package, making it easier for users to create their own custom objects (known generically as an *importer*) to participate in the import process. Details on custom importers can be found in PEP 302.

See Also:

- **import**  The language reference for the *import* statement.
- **Packages specification**  Original specification of packages. Some semantics have changed since the writing of this document (e.g. redirecting based on `None` in `sys.modules`).
- **The __import__() function**  The *import* statement is syntactic sugar for this function.
- **PEP 235**  Import on Case-Insensitive Platforms
- **PEP 263**  Defining Python Source Code Encodings
- **PEP 302**  New Import Hooks
- **PEP 328**  Imports: Multi-Line and Absolute/Relative
- **PEP 366**  Main module explicit relative imports
- **PEP 3120**  Using UTF-8 as the Default Source Encoding
- **PEP 3147**  PYC Repository Directories

30.6.2 Functions

```python
importlib.__import__(name, globals=None, locals=None, fromlist=(), level=0)
```

An implementation of the built-in `__import__()` function.

```python
importlib.import_module(name, package=None)
```

Import a module. The `name` argument specifies what module to import in absolute or relative terms (e.g. either `pkg.mod` or `..mod`). If the name is specified in relative terms, then the `package` argument must be set to the name of the package which is to act as the anchor for resolving the package name (e.g. `import_module('..mod', 'pkg.subpkg')` will import `pkg.mod`).
The `import_module()` function acts as a simplifying wrapper around `importlib.__import__()`. This means all semantics of the function are derived from `importlib.__import__()`, including requiring the package from which an import is occurring to have been previously imported (i.e., `package` must already be imported). The most important difference is that `import_module()` returns the most nested package or module that was imported (e.g. `pkg.mod`), while `__import__()` returns the top-level package or module (e.g. `pkg`).

```python
importlib.find_loader(name, path=None)
```

Find the loader for a module, optionally within the specified `path`. If the module is in `sys.modules`, then `sys.modules[name]` is returned (unless the loader would be `None`, in which case `ValueError` is raised). Otherwise a search using `sys.meta_path` is done. `None` is returned if no loader is found.

A dotted name does not have its parent’s implicitly imported as that requires loading them and that may not be desired. To properly import a submodule you will need to import all parent packages of the submodule and use the correct argument to `path`.

```python
importlib.invalidate_caches()
```

Invalidate the internal caches of finders stored at `sys.meta_path`. If a finder implements `invalidate_caches()` then it will be called to perform the invalidation. This function should be called if any modules are created/installed while your program is running to guarantee all finders will notice the new module’s existence. New in version 3.3.

### 30.6.3 `importlib.abc` – Abstract base classes related to import

The `importlib.abc` module contains all of the core abstract base classes used by `import`. Some subclasses of the core abstract base classes are also provided to help in implementing the core ABCs.

**ABC hierarchy:**

```plaintext
object
|-- Finder (deprecated)
   |   |-- MetaPathFinder
   |   |-- PathEntryFinder
   |-- Loader
      |-- ResourceLoader
      |-- InspectLoader
      |   |-- ExecutionLoader
      |       `-- FileLoader
      |           `-- SourceLoader
      |               |-- PyLoader (deprecated)
      |               `-- PyPycLoader (deprecated)
```

```python
class importlib.abc.Finder
    An abstract base class representing a `finder`. Deprecated since version 3.3: Use `MetaPathFinder` or `PathEntryFinder` instead.

    `find_module` *(fullname, path=None)*
    An abstract method for finding a `loader` for the specified module. Originally specified in PEP 302, this method was meant for use in `sys.meta_path` and in the path-based import subsystem.

class importlib.abc.MetaPathFinder
    An abstract base class representing a `meta path finder`. For compatibility, this is a subclass of `Finder`. New in version 3.3.

    `find_module` *(fullname, path)*
    An abstract method for finding a `loader` for the specified module. If this is a top-level import, `path` will be `None`. Otherwise, this is a search for a subpackage or module and `path` will be the value of `__path__` from the parent package. If a loader cannot be found, `None` is returned.

    `invalidate_caches()`
    An optional method which, when called, should invalidate any internal cache used by the finder.
```
Used by `importlib.invalidate_caches()` when invalidating the caches of all finders on `sys.meta_path`.

**class importlib.abc.PathEntryFinder**

An abstract base class representing a *path entry finder*. Though it bears some similarities to `MetaPathFinder`, `PathEntryFinder` is meant for use only within the path-based import subsystem provided by `PathFinder`. This ABC is a subclass of `Finder` for compatibility. New in version 3.3.

**find_loader (fullname)**

An abstract method for finding a *loader* for the specified module. Returns a 2-tuple of `(loader, portion)` where `portion` is a sequence of file system locations contributing to part of a namespace package. The loader may be `None` while specifying `portion` to signify the contribution of the file system locations to a namespace package. An empty list can be used for `portion` to signify the loader is not part of a package. If `loader` is `None` and `portion` is the empty list then no loader or location for a namespace package were found (i.e. failure to find anything for the module).

**find_module (fullname)**

A concrete implementation of `Finder.find_module()` which is equivalent to `self.find_loader(fullname)[0]`.

**invalidate_caches()**

An optional method which, when called, should invalidate any internal cache used by the finder. Used by `PathFinder.invalidate_caches()` when invalidating the caches of all cached finders.

**class importlib.abc.Loader**

An abstract base class for a *loader*. See PEP 302 for the exact definition for a loader.

**load_module (fullname)**

An abstract method for loading a module. If the module cannot be loaded, `ImportError` is raised, otherwise the loaded module is returned.

If the requested module already exists in `sys.modules`, that module should be used and reloaded. Otherwise the loader should create a new module and insert it into `sys.modules` before any loading begins, to prevent recursion from the import. If the loader inserted a module and the load fails, it must be removed by the loader from `sys.modules`; modules already in `sys.modules` before the loader began execution should be left alone. The `importlib.util.module_for_loader()` decorator handles all of these details.

The loader should set several attributes on the module. (Note that some of these attributes can change when a module is reloaded.)

- `__name__` The name of the module.
- `__file__` The path to where the module data is stored (not set for built-in modules).
- `__cached__` The path to where a compiled version of the module is/should be stored (not set when the attribute would be inappropriate).
- `__path__` A list of strings specifying the search path within a package. This attribute is not set on modules.
- `__package__` The parent package for the module/package. If the module is top-level then it has a value of the empty string. The `importlib.util.set_package()` decorator can handle the details for `__package__`.
- `__loader__` The loader used to load the module. (This is not set by the built-in import machinery, but it should be set whenever a *loader* is used.)

**module_repr (module)**

An abstract method which when implemented calculates and returns the given module’s repr, as a string.

**class importlib.abc.ResourceLoader**

An abstract base class for a *loader* which implements the optional PEP 302 protocol for loading arbitrary resources from the storage back-end.
The Python Library Reference, Release 3.3.3

get_data(path)
An abstract method to return the bytes for the data located at path. Loaders that have a file-like storage back-end that allows storing arbitrary data can implement this abstract method to give direct access to the data stored. IOError is to be raised if the path cannot be found. The path is expected to be constructed using a module’s __file__ attribute or an item from a package’s __path__.

class importlib.abc.InspectLoader
An abstract base class for a loader which implements the optional PEP 302 protocol for loaders that inspect modules.

get_code(fullname)
An abstract method to return the code object for a module. None is returned if the module does not have a code object (e.g. built-in module). ImportError is raised if loader cannot find the requested module.

get_source(fullname)
An abstract method to return the source of a module. It is returned as a text string using universal newlines, translating all recognized line separators into ‘\n’ characters. Returns None if no source is available (e.g. a built-in module). Raises ImportError if the loader cannot find the module specified.

is_package(fullname)
An abstract method to return a true value if the module is a package, a false value otherwise. ImportError is raised if the loader cannot find the module.

class importlib.abc.ExecutionLoader
An abstract base class which inherits from InspectLoader that, when implemented, helps a module to be executed as a script. The ABC represents an optional PEP 302 protocol.

get_filename(fullname)
An abstract method that is to return the value of __file__ for the specified module. If no path is available, ImportError is raised.

If source code is available, then the method should return the path to the source file, regardless of whether a bytecode was used to load the module.

class importlib.abc.FileLoader(fullname, path)
An abstract base class which inherits from ResourceLoader and ExecutionLoader, providing concrete implementations of ResourceLoader.get_data() and ExecutionLoader.get_filename().

The fullname argument is a fully resolved name of the module the loader is to handle. The path argument is the path to the file for the module. New in version 3.3.

name
The name of the module the loader can handle.

path
Path to the file of the module.

load_module(fullname)
Calls super’s load_module().

get_filename(fullname)
Returns path.

get_data(path)
Returns the open, binary file for path.

class importlib.abc.SourceLoader
An abstract base class for implementing source (and optionally bytecode) file loading. The class inherits from both ResourceLoader and ExecutionLoader, requiring the implementation of:

• ResourceLoader.get_data()
• `ExecutionLoader.get_filename()` Should only return the path to the source file; sourceless loading is not supported.

The abstract methods defined by this class are to add optional bytecode file support. Not implementing these optional methods causes the loader to only work with source code. Implementing the methods allows the loader to work with source and bytecode files; it does not allow for sourceless loading where only bytecode is provided. Bytecode files are an optimization to speed up loading by removing the parsing step of Python’s compiler, and so no bytecode-specific API is exposed.

`path_stats(path)`
Optional abstract method which returns a `dict` containing metadata about the specified path. Supported dictionary keys are:

• `mtime` (mandatory): an integer or floating-point number representing the modification time of the source code;
• `size` (optional): the size in bytes of the source code.

Any other keys in the dictionary are ignored, to allow for future extensions. New in version 3.3.

`path_mtime(path)`
Optional abstract method which returns the modification time for the specified path. Deprecated since version 3.3: This method is deprecated in favour of `path_stats()`. You don’t have to implement it, but it is still available for compatibility purposes.

`set_data(path, data)`
Optional abstract method which writes the specified bytes to a file path. Any intermediate directories which do not exist are to be created automatically.

When writing to the path fails because the path is read-only (`errno.EACCES/PermissionError`), do not propagate the exception.

`get_code(fullname)`
Concrete implementation of `InspectLoader.get_code()`.

`load_module(fullname)`
Concrete implementation of `Loader.load_module()`.

`get_source(fullname)`
Concrete implementation of `InspectLoader.get_source()`.

`is_package(fullname)`
Concrete implementation of `InspectLoader.is_package()`. A module is determined to be a package if its file path (as provided by `ExecutionLoader.get_filename()`) is a file named `__init__` when the file extension is removed and the module name itself does not end in `__init__`.

`class importlib.abc.PyLoader`
An abstract base class inheriting from `ExecutionLoader` and `ResourceLoader` designed to ease the loading of Python source modules (bytecode is not handled; see `SourceLoader` for a source/bytecode ABC). A subclass implementing this ABC will only need to worry about exposing how the source code is stored; all other details for loading Python source code will be handled by the concrete implementations of key methods. Deprecated since version 3.2: This class has been deprecated in favor of `SourceLoader` and is slated for removal in Python 3.4. See below for how to create a subclass that is compatible with Python 3.1 onwards. If compatibility with Python 3.1 is required, then use the following idiom to implement a subclass that will work with Python 3.1 onwards (make sure to implement `ExecutionLoader.get_filename()`):

```python
try:
    from importlib.abc import SourceLoader
except ImportError:
    from importlib.abc import PyLoader as SourceLoader

class CustomLoader(SourceLoader):
```


def get_filename(self, fullname):
    """Return the path to the source file."""
    # Implement ...

def source_path(self, fullname):
    """Implement source_path in terms of get_filename."""
    try:
        return self.get_filename(fullname)
    except ImportError:
        return None

def is_package(self, fullname):
    """Implement is_package by looking for an __init__ file
name as returned by get_filename."""
    filename = os.path.basename(self.get_filename(fullname))
    return os.path.splitext(filename)[0] == '__init__'

source_path(fullname)
An abstract method that returns the path to the source code for a module. Should return None if there
is no source code. Raises ImportError if the loader knows it cannot handle the module.

get_filename(fullname)
A concrete implementation of importlib.abc.ExecutionLoader.get_filename() that
relies on source_path(). If source_path() returns None, then ImportError is raised.

load_module(fullname)
A concrete implementation of importlib.abc.Loader.load_module() that loads Python
source code. All needed information comes from the abstract methods required by this ABC. The
only pertinent assumption made by this method is that when loading a package __path__ is set to
[os.path.dirname(__file__)].

get_code(fullname)
A concrete implementation of importlib.abc.InspectLoader.get_code() that creates
code objects from Python source code, by requesting the source code (using source_path() and
get_data()) and compiling it with the built-in compile() function.

get_source(fullname)
A concrete implementation of importlib.abc.InspectLoader.get_source(). Uses
importlib.abc.ResourceLoader.get_data() and source_path() to get the source
code. It tries to guess the source encoding using tokenize.detect_encoding().

class importlib.abc.PyPycLoader
An abstract base class inheriting from PyLoader. This ABC is meant to help in creating loaders that
support both Python source and bytecode. Deprecated since version 3.2: This class has been deprecated in
favor of SourceLoader and to properly support PEP 3147. If compatibility is required with Python 3.1,
implement both SourceLoader and PyLoader; instructions on how to do so are included in the doc-
umentation for PyLoader. Do note that this solution will not support sourceless/bytecode-only loading;
only source and bytecode loading.Changed in version 3.3: Updated to parse (but not use) the new source
size field in bytecode files when reading and to write out the field properly when writing.

source_mtime(fullname)
An abstract method which returns the modification time for the source code of the specified module.
The modification time should be an integer. If there is no source code, return None. If the module
cannot be found then ImportError is raised.

bytecode_path(fullname)
An abstract method which returns the path to the bytecode for the specified module, if it exists. It
returns None if no bytecode exists (yet). Raises ImportError if the loader knows it cannot handle
the module.

get_filename(fullname)
A concrete implementation of ExecutionLoader.get_filename() that relies on
write_bytecode (fullname, bytecode)
An abstract method which has the loader write bytecode for future use. If the bytecode is written, return True. Return False if the bytecode could not be written. This method should not be called if sys.dont_write_bytecode is true. The bytecode argument should be a bytes string or bytes array.

30.6.4 importlib.machinery – Importers and path hooks

This module contains the various objects that help import find and load modules.

importlib.machinery.SOURCE_SUFFIXES
A list of strings representing the recognized file suffixes for source modules. New in version 3.3.

importlib.machinery.DEBUG_BYTECODE_SUFFIXES
A list of strings representing the file suffixes for non-optimized bytecode modules. New in version 3.3.

importlib.machinery.OPTIMIZED_BYTECODE_SUFFIXES
A list of strings representing the file suffixes for optimized bytecode modules. New in version 3.3.

importlib.machinery.BYTECODE_SUFFIXES
A list of strings representing the recognized file suffixes for bytecode modules. Set to either DEBUG_BYTECODE_SUFFIXES or OPTIMIZED_BYTECODE_SUFFIXES based on whether __debug__ is true. New in version 3.3.

importlib.machinery.EXTENSION_SUFFIXES
A list of strings representing the recognized file suffixes for extension modules. New in version 3.3.

importlib.machinery.all_suffixes ()
Returns a combined list of strings representing all file suffixes for modules recognized by the standard import machinery. This is a helper for code which simply needs to know if a filesystem path potentially refers to a module without needing any details on the kind of module (for example, inspect.getmodulename ()). New in version 3.3.

class importlib.machinery.BuiltinImporter
An importer for built-in modules. All known built-in modules are listed in sys.builtin_module_names. This class implements the importlib.abc.MetaPathFinder and importlib.abc.InspectLoader ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

class importlib.machinery.FrozenImporter
An importer for frozen modules. This class implements the importlib.abc.MetaPathFinder and importlib.abc.InspectLoader ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

class importlib.machinery.WindowsRegistryFinder
Finder for modules declared in the Windows registry. This class implements the importlib.abc.Finder ABC.

Only class methods are defined by this class to alleviate the need for instantiation. New in version 3.3.

class importlib.machinery.PathFinder
A Finder for sys.path and package __path__ attributes. This class implements the importlib.abc.MetaPathFinder ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

classmethod find_module (fullname, path=None)
Class method that attempts to find a loader for the module specified by fullname on sys.path or, if defined, on path. For each path entry that is searched, sys.path_importer_cache is checked.
If a non-false object is found then it is used as the path entry finder to look for the module being searched for. If no entry is found in sys.path_importer_cache, then sys.path_hooks is searched for a finder for the path entry and, if found, is stored in sys.path_importer_cache along with being queried about the module. If no finder is ever found then None is both stored in the cache and returned.

classmethod invalidate_caches()
    Calls importlib.abc.PathEntryFinder.invalidate_caches() on all finders stored in sys.path_importer_cache.

class importlib.machinery.FileFinder (path, *loader_details)
    A concrete implementation of importlib.abc.PathEntryFinder which caches results from the file system.

    The path argument is the directory for which the finder is in charge of searching.

    The loader_details argument is a variable number of 2-item tuples each containing a loader and a sequence of file suffixes the loader recognizes. The loaders are expected to be callables which accept two arguments of the module’s name and the path to the file found.

    The finder will cache the directory contents as necessary, making stat calls for each module search to verify the cache is not outdated. Because cache staleness relies upon the granularity of the operating system’s state information of the file system, there is a potential race condition of searching for a module, creating a new file, and then searching for the module the new file represents. If the operations happen fast enough to fit within the granularity of stat calls, then the module search will fail. To prevent this from happening, when you create a module dynamically, make sure to call importlib.invalidate_caches(). New in version 3.3.

    path
        The path the finder will search in.

    find_loader (fullname)
        Attempt to find the loader to handle fullname within path.

    invalidate_caches ()
        Clear out the internal cache.

classmethod path_hook (*loader_details)
    A class method which returns a closure for use on sys.path_hooks. An instance of FileFinder is returned by the closure using the path argument given to the closure directly and loader_details indirectly.

    If the argument to the closure is not an existing directory, ImportError is raised.

class importlib.machinery.SourceFileLoader (fullname, path)
    A concrete implementation of importlib.abc.SourceLoader by subclassing importlib.abc.FileLoader and providing some concrete implementations of other methods. New in version 3.3.

    name
        The name of the module that this loader will handle.

    path
        The path to the source file.

    is_package (fullname)
        Return true if path appears to be for a package.

    path_stats (path)
        Concrete implementation of importlib.abc.SourceLoader.path_stats().

    set_data (path, data)
        Concrete implementation of importlib.abc.SourceLoader.set_data().

class importlib.machinery.SourcelessFileLoader (fullname, path)
    A concrete implementation of importlib.abc.FileLoader which can import bytecode files (i.e. no source code files exist).
Please note that direct use of bytecode files (and thus not source code files) inhibits your modules from being usable by all Python implementations or new versions of Python which change the bytecode format. New in version 3.3.

**name**
The name of the module the loader will handle.

**path**
The path to the bytecode file.

**is_package** *(fullname)*
Determines if the module is a package based on `path`.

**get_code** *(fullname)*
Returns the code object for `name` created from `path`.

**get_source** *(fullname)*
Returns None as bytecode files have no source when this loader is used.

```python
class importlib.machinery.ExtensionFileLoader (fullname, path):
    A concrete implementation of importlib.abc.InspectLoader for extension modules.
```

The `fullname` argument specifies the name of the module the loader is to support. The `path` argument is the path to the extension module’s file. New in version 3.3.

**name**
Name of the module the loader supports.

**path**
Path to the extension module.

**load_module** *(fullname)*
Loads the extension module if and only if `fullname` is the same as `name` or is None.

**is_package** *(fullname)*
Returns True if the file path points to a package’s __init__ module based on EXTENSION_SUFFIXES.

**get_code** *(fullname)*
Returns None as extension modules lack a code object.

**get_source** *(fullname)*
Returns None as extension modules do not have source code.

### 30.6.5 importlib.util – Utility code for importers

This module contains the various objects that help in the construction of an importer.

```python
importlib.util.resolve_name (name, package)
```

Resolve a relative module name to an absolute one.

If `name` has no leading dots, then `name` is simply returned. This allows for usage such as `importlib.util.resolve_name(‘sys’, __package__)` without doing a check to see if the `package` argument is needed.

`ValueError` is raised if `name` is a relative module name but package is a false value (e.g. None or the empty string). `ValueError` is also raised a relative name would escape its containing package (e.g. requesting ..bacon from within the spam package). New in version 3.3.

```python
@importlib.util.module_for_loader
```

A decorator for a loader method, to handle selecting the proper module object to load with. The decorated method is expected to have a call signature taking two positional arguments (e.g. `load_module(self, module)`) for which the second argument will be the module `object` to be used by the loader. Note that the decorator will not work on static methods because of the assumption of two arguments.
The decorated method will take in the name of the module to be loaded as expected for a loader. If the module is not found in sys.modules then a new one is constructed with its __name__ attribute set to name, __loader__ set to self, and __package__ set if importlib.abc.InspectLoader.is_package() is defined for self and does not raise ImportError for name. If a new module is not needed then the module found in sys.modules will be passed into the method.

If an exception is raised by the decorated method and a module was added to sys.modules it will be removed to prevent a partially initialized module from being left in sys.modules. If the module was already in sys.modules then it is left alone.

Use of this decorator handles all the details of which module object a loader should initialize as specified by PEP 302 as best as possible. Changed in version 3.3: __loader__ and __package__ are automatically set (when possible).

@importlib.util.set_loader

A decorator for a loader method, to set the __loader__ attribute on loaded modules. If the attribute is already set the decorator does nothing. It is assumed that the first positional argument to the wrapped method (i.e. self) is what __loader__ should be set to.

Note: It is recommended that module_for_loader() be used over this decorator as it subsumes this functionality.

@importlib.util.set_package

A decorator for a loader to set the __package__ attribute on the module returned by the loader. If __package__ is set and has a value other than None it will not be changed. Note that the module returned by the loader is what has the attribute set on and not the module found in sys.modules.

Reliance on this decorator is discouraged when it is possible to set __package__ before importing. By setting it beforehand the code for the module is executed with the attribute set and thus can be used by global level code during initialization.

Note: It is recommended that module_for_loader() be used over this decorator as it subsumes this functionality.
CHAPTER
THIRTYONE

PYTHON LANGUAGE SERVICES

Python provides a number of modules to assist in working with the Python language. These modules support

tokenizing, parsing, syntax analysis, bytecode disassembly, and various other facilities.

These modules include:

31.1 parser — Access Python parse trees

The parser module provides an interface to Python’s internal parser and byte-code compiler. The primary

purpose for this interface is to allow Python code to edit the parse tree of a Python expression and create executable
code from this. This is better than trying to parse and modify an arbitrary Python code fragment as a string because

parsing is performed in a manner identical to the code forming the application. It is also faster.

Note: From Python 2.5 onward, it’s much more convenient to cut in at the Abstract Syntax Tree (AST) generation

and compilation stage, using the ast module.

There are a few things to note about this module which are important to making use of the data structures created.
This is not a tutorial on editing the parse trees for Python code, but some examples of using the parser module
are presented.

Most importantly, a good understanding of the Python grammar processed by the internal parser is required. For
full information on the language syntax, refer to reference-index. The parser itself is created from a grammar
specification defined in the file Grammar/Grammar in the standard Python distribution. The parse trees stored
in the ST objects created by this module are the actual output from the internal parser when created by the expr()
or suite() functions, described below. The ST objects created by sequence2st() faithfully simulate those
structures. Be aware that the values of the sequences which are considered “correct” will vary from one version of
Python to another as the formal grammar for the language is revised. However, transporting code from one Python
version to another as source text will always allow correct parse trees to be created in the target version, with the
only restriction being that migrating to an older version of the interpreter will not support more recent language
constructs. The parse trees are not typically compatible from one version to another, whereas source code has
always been forward-compatible.

Each element of the sequences returned by st2list() or st2tuple() has a simple form. Sequences repre-
senting non-terminal elements in the grammar always have a length greater than one. The first element is an
integer which identifies a production in the grammar. These integers are given symbolic names in the C header file
Include/graminit.h and the Python module symbol. Each additional element of the sequence represents

a component of the production as recognized in the input string: these are always sequences which have the same
form as the parent. An important aspect of this structure which should be noted is that keywords used to identify
the parent node type, such as the keyword if in an if_stmt, are included in the node tree without any special
treatment. For example, the if keyword is represented by the tuple (1, ‘if’), where 1 is the numeric value
associated with all NAME tokens, including variable and function names defined by the user. In an alternate form
returned when line number information is requested, the same token might be represented as (1, ‘if’, 12),
where the 12 represents the line number at which the terminal symbol was found.
Terminal elements are represented in much the same way, but without any child elements and the addition of the source text which was identified. The example of the `if` keyword above is representative. The various types of terminal symbols are defined in the C header file `Include/token.h` and the Python module `token`.

The ST objects are not required to support the functionality of this module, but are provided for three purposes: to allow an application to amortize the cost of processing complex parse trees, to provide a parse tree representation which conserves memory space when compared to the Python list or tuple representation, and to ease the creation of additional modules in C which manipulate parse trees. A simple “wrapper” class may be created in Python to hide the use of ST objects.

The `parser` module defines functions for a few distinct purposes. The most important purposes are to create ST objects and to convert ST objects to other representations such as parse trees and compiled code objects, but there are also functions which serve to query the type of parse tree represented by an ST object.

**See Also:**

- **Module symbol** Useful constants representing internal nodes of the parse tree.
- **Module token** Useful constants representing leaf nodes of the parse tree and functions for testing node values.

### 31.1.1 Creating ST Objects

ST objects may be created from source code or from a parse tree. When creating an ST object from source, different functions are used to create the `eval` and `exec` forms.

**parser.expr**

The `expr()` function parses the parameter `source` as if it were an input to `compile(source, 'file.py', 'eval')`. If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

**parser.suite**

The `suite()` function parses the parameter `source` as if it were an input to `compile(source, 'file.py', 'exec')`. If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

**parser.sequence2st**

This function accepts a parse tree represented as a sequence and builds an internal representation if possible. If it can validate that the tree conforms to the Python grammar and all nodes are valid node types in the host version of Python, an ST object is created from the internal representation and returned to the called. If there is a problem creating the internal representation, or if the tree cannot be validated, a `ParserError` exception is raised. An ST object created this way should not be assumed to compile correctly; normal exceptions raised by compilation may still be initiated when the ST object is passed to `compilest()`. This may indicate problems not related to syntax (such as a `MemoryError` exception), but may also be due to constructs such as the result of parsing `del f(0)`, which escapes the Python parser but is checked by the bytecode compiler.

Sequences representing terminal tokens may be represented as either two-element lists of the form `(1, 'name')` or as three-element lists of the form `(1, 'name', 56)`. If the third element is present, it is assumed to be a valid line number. The line number may be specified for any subset of the terminal symbols in the input tree.

**parser.tuple2st**

This is the same function as `sequence2st()`. This entry point is maintained for backward compatibility.

### 31.1.2 Converting ST Objects

ST objects, regardless of the input used to create them, may be converted to parse trees represented as list- or tuple- trees, or may be compiled into executable code objects. Parse trees may be extracted with or without line numbering information.

**parser.st2list**

This function accepts an ST object from the caller in `st` and returns a Python list representing the equivalent
parse tree. The resulting list representation can be used for inspection or the creation of a new parse tree in list form. This function does not fail so long as memory is available to build the list representation. If the parse tree will only be used for inspection, \texttt{st2tuple()} should be used instead to reduce memory consumption and fragmentation. When the list representation is required, this function is significantly faster than retrieving a tuple representation and converting that to nested lists.

If \texttt{line\_info} is true, line number information will be included for all terminal tokens as a third element of the list representing the token. Note that the line number provided specifies the line on which the token \emph{ends}. This information is omitted if the flag is false or omitted.

\begin{itemize}
  \item \texttt{parser.st2tuple(st, line\_info=False, col\_info=False)}
  \begin{itemize}
    \item This function accepts an ST object from the caller in \texttt{st} and returns a Python tuple representing the equivalent parse tree. Other than returning a tuple instead of a list, this function is identical to \texttt{st2list()}.
    \item If \texttt{line\_info} is true, line number information will be included for all terminal tokens as a third element of the list representing the token. This information is omitted if the flag is false or omitted.
  \end{itemize}
\end{itemize}

\begin{itemize}
  \item \texttt{parser.compilest(st, filename='<syntax-tree>')}
  \begin{itemize}
    \item The Python byte compiler can be invoked on an ST object to produce code objects which can be used as part of a call to the built-in \texttt{exec()} or \texttt{eval()} functions. This function provides the interface to the compiler, passing the internal parse tree from \texttt{st} to the parser, using the source file name specified by the \texttt{filename} parameter. The default value supplied for \texttt{filename} indicates that the source was an ST object.
    \item Compiling an ST object may result in exceptions related to compilation; an example would be a \texttt{SyntaxError} caused by the parse tree for \texttt{del f(0)}: this statement is considered legal within the formal grammar for Python but is not a legal language construct. The \texttt{SyntaxError} raised for this condition is actually generated by the Python byte-compiler normally, which is why it can be raised at this point by the \texttt{parser} module. Most causes of compilation failure can be diagnosed programmatically by inspection of the parse tree.
  \end{itemize}
\end{itemize}

\subsection{Queries on ST Objects}

Two functions are provided which allow an application to determine if an ST was created as an expression or a suite. Neither of these functions can be used to determine if an ST was created from source code via \texttt{expr()} or \texttt{suite()} or from a parse tree via \texttt{sequence2st()}.

\begin{itemize}
  \item \texttt{parser.isexpr(st)}
    \begin{itemize}
      \item When \texttt{st} represents an ‘\texttt{eval}’ form, this function returns true, otherwise it returns false. This is useful, since code objects normally cannot be queried for this information using existing built-in functions. Note that the code objects created by \texttt{compilest()} cannot be queried like this either, and are identical to those created by the built-in \texttt{compile()} function.
    \end{itemize}
  \item \texttt{parser.issuite(st)}
    \begin{itemize}
      \item This function mirrors \texttt{isexpr()} in that it reports whether an ST object represents an ‘\texttt{exec}’ form, commonly known as a “suite.” It is not safe to assume that this function is equivalent to \texttt{not isexpr(st)}, as additional syntactic fragments may be supported in the future.
    \end{itemize}
\end{itemize}

\subsection{Exceptions and Error Handling}

The parser module defines a single exception, but may also pass other built-in exceptions from other portions of the Python runtime environment. See each function for information about the exceptions it can raise.

\begin{itemize}
  \item \texttt{exception parser.ParserError}
    \begin{itemize}
      \item Exception raised when a failure occurs within the parser module. This is generally produced for validation failures rather than the built-in \texttt{SyntaxError} raised during normal parsing. The exception argument is either a string describing the reason of the failure or a tuple containing a sequence causing the failure from a parse tree passed to \texttt{sequence2st()} and an explanatory string. Calls to \texttt{sequence2st()} need to be able to handle either type of exception, while calls to other functions in the module will only need to be aware of the simple string values.
    \end{itemize}
\end{itemize}
Note that the functions `compilest()`, `expr()`, and `suite()` may raise exceptions which are normally raised by the parsing and compilation process. These include the built-in exceptions `MemoryError`, `OverflowError`, `SyntaxError`, and `SystemError`. In these cases, these exceptions carry all the meaning normally associated with them. Refer to the descriptions of each function for detailed information.

### 31.1.5 ST Objects

Ordered and equality comparisons are supported between ST objects. Pickling of ST objects (using the `pickle` module) is also supported.

**parser.SType**

The type of the objects returned by `expr()`, `suite()` and `sequence2st()`.

ST objects have the following methods:

- **ST.compile(filename='<syntax-tree>')**
  
  Same as `compilest(st, filename)`.

- **ST.isexpr()**
  
  Same as `isexpr(st)`.

- **ST.issuite()**
  
  Same as `issuite(st)`.

- **ST.tolist(line_info=False, col_info=False)**
  
  Same as `st2list(st, line_info, col_info)`.

- **ST.totuple(line_info=False, col_info=False)**
  
  Same as `st2tuple(st, line_info, col_info)`.

### 31.1.6 Example: Emulation of `compile()`

While many useful operations may take place between parsing and bytecode generation, the simplest operation is to do nothing. For this purpose, using the `parser` module to produce an intermediate data structure is equivalent to the code

```
>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
>>> eval(code)
10
```

The equivalent operation using the `parser` module is somewhat longer, and allows the intermediate internal parse tree to be retained as an ST object:

```
>>> import parser
>>> st = parser.expr('a + 5')
>>> code = st.compile('file.py')
>>> a = 5
>>> eval(code)
10
```

An application which needs both ST and code objects can package this code into readily available functions:

```python
import parser
def load_suite(source_string):
    st = parser.suite(source_string)
    return st, st.compile()

def load_expression(source_string):
    st = parser.expr(source_string)
    return st, st.compile()
```
31.2 ast — Abstract Syntax Trees

Source code: Lib/ast.py

The ast module helps Python applications to process trees of the Python abstract syntax grammar. The abstract syntax itself might change with each Python release; this module helps to find out programatically what the current grammar looks like.

An abstract syntax tree can be generated by passing ast.PyCF_ONLY_AST as a flag to the compile() built-in function, or using the parse() helper provided in this module. The result will be a tree of objects whose classes all inherit from ast.AST. An abstract syntax tree can be compiled into a Python code object using the built-in compile() function.

31.2.1 Node classes

class ast.AST

This is the base of all AST node classes. The actual node classes are derived from the Parser/Python.asdl file, which is reproduced below. They are defined in the _ast C module and re-exported in ast.

There is one class defined for each left-hand side symbol in the abstract grammar (for example, ast.stmt or ast.expr). In addition, there is one class defined for each constructor on the right-hand side; these classes inherit from the classes for the left-hand side trees. For example, ast.BinOp inherits from ast.expr. For production rules with alternatives (aka “sums”), the left-hand side class is abstract: only instances of specific constructor nodes are ever created.

_fields

Each concrete class has an attribute _fields which gives the names of all child nodes.

Each instance of a concrete class has one attribute for each child node, of the type as defined in the grammar. For example, ast BinOp instances have an attribute left of type ast.expr.

If these attributes are marked as optional in the grammar (using a question mark), the value might be None. If the attributes can have zero-or-more values (marked with an asterisk), the values are represented as Python lists. All possible attributes must be present and have valid values when compiling an AST with compile().

lineno
col_offset

Instances of ast.expr and ast.stmt subclasses have lineno and col_offset attributes. The lineno is the line number of source text (1-indexed so the first line is line 1) and the col_offset is the UTF-8 byte offset of the first token that generated the node. The UTF-8 offset is recorded because the parser uses UTF-8 internally.

The constructor of a class ast.T parses its arguments as follows:

• If there are positional arguments, there must be as many as there are items in T._fields; they will be assigned as attributes of these names.

• If there are keyword arguments, they will set the attributes of the same names to the given values.

For example, to create and populate an ast.UnaryOp node, you could use

```python
node = ast.UnaryOp()
node.op = ast.USub()
node.operand = ast.Num()
node.operand.n = 5
node.operand.lineno = 0
node.operand.col_offset = 0
node.lineno = 0
node.col_offset = 0
```
or the more compact

```python
node = ast.UnaryOp(ast.USub(), ast.Num(5, lineno=0, col_offset=0),
                   lineno=0, col_offset=0)
```

### 31.2.2 Abstract Grammar

The abstract grammar is currently defined as follows:

```python
-- ASDL’s five builtin types are identifier, int, string, bytes, object
module Python
{
    mod = Module(stmt* body)
    | Interactive(stmt* body)
    | Expression(expr body)

    -- not really an actual node but useful in Jython’s typesystem.
    | Suite(stmt* body)

    stmt = FunctionDef(identifier name, arguments args,
                        stmt* body, expr* decorator_list, expr? returns)
    | ClassDef(identifier name, expr* bases, keyword* keywords, expr? starargs, expr? kwargs, stmt* body, expr* decorator_list)
    | Return(expr? value)

    | Delete(expr* targets)
    | Assign(expr* targets, expr value)
    | AugAssign(expr target, operator op, expr value)

    -- use ‘orelse’ because else is a keyword in target languages
    | For(expr target, expr iter, stmt* body, stmt* orelse)
    | While(expr test, stmt* body, stmt* orelse)
    | If(expr test, stmt* body, stmt* orelse)
    | With(withitem* items, stmt* body)

    | Raise(expr? exc, expr? cause)
    | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
    | Assert(expr test, expr? msg)

    | Import(alias* names)
    | ImportFrom(identifier? module, alias* names, int? level)

    | Global(identifier* names)
    | Nonlocal(identifier* names)
    | Expr(expr value)
    | Pass | Break | Continue

    -- XXX Jython will be different
    -- col_offset is the byte offset in the utf8 string the parser uses
    attributes (int lineno, int col_offset)

    -- BoolOp() can use left & right?
```
expr = BoolOp(boolop op, expr* values)
| BinOp(expr left, operator op, expr right)
| UnaryOp(unaryop op, expr operand)
| Lambda(arguments args, expr body)
| IfExp(expr test, expr body, expr orelse)
| Dict(expr* keys, expr* values)
| Set(expr* elts)
| ListComp(expr elt, comprehension* generators)
| SetComp(expr elt, comprehension* generators)
| DictComp(expr key, expr value, comprehension* generators)
| GeneratorExp(expr elt, comprehension* generators)
-- the grammar constrains where yield expressions can occur
| Yield(expr? value)
| YieldFrom(expr value)
-- need sequences for compare to distinguish between
-- x < 4 < 3 and (x < 4) < 3
| Compare(expr left, cmpop* ops, expr* comparators)
| Call(expr func, expr* args, keyword* keywords,
| expr? starargs, expr? kwargs)
| Num(object n) -- a number as a PyObject.
| Str(string s) -- need to specify raw, unicode, etc?
| Bytes(bytes s)
| Ellipsis
-- other literals? bools?
-- the following expression can appear in assignment context
| Attribute(expr value, identifier attr, expr_context ctx)
| Subscript(expr value, slice slice, expr_context ctx)
| Starred(expr value, expr_context ctx)
| Name(identifier id, expr_context ctx)
| List(expr* elts, expr_context ctx)
| Tuple(expr* elts, expr_context ctx)
-- col_offset is the byte offset in the utf8 string the parser uses
-- attributes (int lineno, int col_offset)

expr_context = Load | Store | Del | AugLoad | AugStore | Param

slice = Slice(expr? lower, expr? upper, expr? step)
| ExtSlice(slice* dims)
| Index(expr value)

boolop = And | Or

operator = Add | Sub | Mult | Div | Mod | Pow | LShift
| RShift | BitOr | BitXor | BitAnd | FloorDiv

unaryop = Invert | Not | UAdd | USub

cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

comprehension = (expr target, expr iter, expr* ifs)

excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
-- attributes (int lineno, int col_offset)

arguments = (arg* args, identifier? vararg, expr? varargannotation,
| arg* kwonlyargs, identifier? kwarg,
The Python Library Reference, Release 3.3.3

expr? kwargannotation, expr* defaults,
expr* kw_defaults)
arg = (identifier arg, expr? annotation)

-- keyword arguments supplied to call
keyword = (identifier arg, expr value)

-- import name with optional ‘as’ alias.
alias = (identifier name, identifier? asname)

withitem = (expr context_expr, expr? optional_vars)
}

31.2.3 ast Helpers

Apart from the node classes, ast module defines these utility functions and classes for traversing abstract syntax
trees:

ast.parse(source, filename='<unknown>', mode='exec')
Parse the source into an AST node. Equivalent to compile(source, filename, mode,
ast.PyCF_ONLY_AST).

ast.literal_eval(node_or_string)
Safely evaluate an expression node or a string containing a Python expression. The string or node provided
may only consist of the following Python literal structures: strings, bytes, numbers, tuples, lists, dicts, sets,
booleans, and None.

This can be used for safely evaluating strings containing Python expressions from untrusted sources without
the need to parse the values oneself. Changed in version 3.2: Now allows bytes and set literals.

ast.get_docstring(node, clean=True)
Return the docstring of the given node (which must be a FunctionDef, ClassDef or Module
node), or None if it has no docstring. If clean is true, clean up the docstring’s indentation with
inspect.cleandoc().

ast.fix_missing_locations(node)
When you compile a node tree with compile(), the compiler expects lineno and col_offset at-
tributes for every node that supports them. This is rather tedious to fill in for generated nodes, so this helper
adds these attributes recursively where not already set, by setting them to the values of the parent node. It
works recursively starting at node.

ast.increment_lineno(node, n=1)
Increment the line number of each node in the tree starting at node by n. This is useful to “move code” to a
different location in a file.

ast.copy_location(new_node, old_node)
Copy source location (lineno and col_offset) from old_node to new_node if possible, and return
new_node.

ast.iter_fields(node)
Yield a tuple of (fieldname, value) for each field in node._fields that is present on node.

ast.iter_child_nodes(node)
Yield all direct child nodes of node, that is, all fields that are nodes and all items of fields that are lists of
nodes.

ast.walk(node)
Recursively yield all descendant nodes in the tree starting at node (including node itself), in no specified
order. This is useful if you only want to modify nodes in place and don’t care about the context.

class ast.NodeVisitor
A node visitor base class that walks the abstract syntax tree and calls a visitor function for every node found.
This function may return a value which is forwarded by the visit() method.
This class is meant to be subclassed, with the subclass adding visitor methods.

**visit(node)**
Visit a node. The default implementation calls the method called ‘self.visit_classname’ where classname is the name of the node class, or generic_visit() if that method doesn’t exist.

**generic_visit(node)**
This visitor calls visit() on all children of the node.

Note that child nodes of nodes that have a custom visitor method won’t be visited unless the visitor calls generic_visit() or visits them itself.

Don’t use the NodeVisitor if you want to apply changes to nodes during traversal. For this a special visitor exists (NodeTransformer) that allows modifications.

```python
class ast.NodeTransformer
A NodeVisitor subclass that walks the abstract syntax tree and allows modification of nodes.

The NodeTransformer will walk the AST and use the return value of the visitor methods to replace or remove the old node. If the return value of the visitor method is None, the node will be removed from its location, otherwise it is replaced with the return value. The return value may be the original node in which case no replacement takes place.

Here is an example transformer that rewrites all occurrences of name lookups (foo) to data[‘foo’]:

class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return copy_location(Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Index(value=Str(s=node.id)),
            ctx=node.ctx
        ), node)

Keep in mind that if the node you’re operating on has child nodes you must either transform the child nodes yourself or call the generic_visit() method for the node first.

For nodes that were part of a collection of statements (that applies to all statement nodes), the visitor may also return a list of nodes rather than just a single node.

Usually you use the transformer like this:

node = YourTransformer().visit(node)

ast.dump(node, annotate_fields=True, include_attributes=False)
```

**symtable — Access to the compiler’s symbol tables**

Symbol tables are generated by the compiler from AST just before bytecode is generated. The symbol table is responsible for calculating the scope of every identifier in the code. symtable provides an interface to examine these tables.

**31.3.1 Generating Symbol Tables**

```python
symtable.symtable(code, filename, compile_type)
```

Return the toplevel SymbolTable for the Python source code. filename is the name of the file containing
the code. compile_type is like the mode argument to compile().

31.3.2 Examining Symbol Tables

class symtable.SymbolTable
   A namespace table for a block. The constructor is not public.
   
   get_type()
      Return the type of the symbol table. Possible values are 'class', 'module', and 'function'.
   
   get_id()
      Return the table’s identifier.
   
   get_name()
      Return the table’s name. This is the name of the class if the table is for a class, the name of the function if the table is for a function, or 'top' if the table is global (get_type() returns 'module').
   
   get_lineno()
      Return the number of the first line in the block this table represents.
   
   is_optimized()
      Return True if the locals in this table can be optimized.
   
   is_nested()
      Return True if the block is a nested class or function.
   
   has_children()
      Return True if the block has nested namespaces within it. These can be obtained with get_children().
   
   has_exec()
      Return True if the block uses exec.
   
   has_import_star()
      Return True if the block uses a starred from-import.
   
   get_identifiers()
      Return a list of names of symbols in this table.
   
   lookup(name)
      Lookup name in the table and return a Symbol instance.
   
   get_symbols()
      Return a list of Symbol instances for names in the table.
   
   get_children()
      Return a list of the nested symbol tables.

class symtable.Function
   A namespace for a function or method. This class inherits SymbolTable.
   
   get_parameters()
      Return a tuple containing names of parameters to this function.
   
   get_locals()
      Return a tuple containing names of locals in this function.
   
   getGlobals()
      Return a tuple containing names of globals in this function.
   
   get_frees()
      Return a tuple containing names of free variables in this function.

class symtable.Class
   A namespace of a class. This class inherits SymbolTable.
get_methods()
    Return a tuple containing the names of methods declared in the class.

class symtable.Symbol
    An entry in a SymbolTable corresponding to an identifier in the source. The constructor is not public.
    get_name()
        Return the symbol's name.
    is_referenced()
        Return True if the symbol is used in its block.
    is_imported()
        Return True if the symbol is created from an import statement.
    is_parameter()
        Return True if the symbol is a parameter.
    is_global()
        Return True if the symbol is global.
    is_declared_global()
        Return True if the symbol is declared global with a global statement.
    is_local()
        Return True if the symbol is local to its block.
    is_free()
        Return True if the symbol is referenced in its block, but not assigned to.
    is_assigned()
        Return True if the symbol is assigned to in its block.
    is_namespace()
        Return True if name binding introduces new namespace.
        If the name is used as the target of a function or class statement, this will be true.
        For example:

        >>> table = symtable.symtable("def some_func(): pass", "string", "exec")
        >>> table.lookup("some_func").is_namespace()
        True

        Note that a single name can be bound to multiple objects. If the result is True, the name may also be bound to other objects, like an int or list, that does not introduce a new namespace.

    get_namespaces()
        Return a list of namespaces bound to this name.
    get_namespace()
        Return the namespace bound to this name. If more than one namespace is bound, a ValueError is raised.

31.4 symbol — Constants used with Python parse trees

Source code: Lib/symbol.py

This module provides constants which represent the numeric values of internal nodes of the parse tree. Unlike most Python constants, these use lower-case names. Refer to the file Grammar/Grammar in the Python distribution for the definitions of the names in the context of the language grammar. The specific numeric values which the names map to may change between Python versions.

This module also provides one additional data object:
symbol.sym_name
Dictionary mapping the numeric values of the constants defined in this module back to name strings, allowing more human-readable representation of parse trees to be generated.

31.5 token — Constants used with Python parse trees

Source code: Lib/token.py

This module provides constants which represent the numeric values of leaf nodes of the parse tree (terminal tokens). Refer to the file Grammar/Grammar in the Python distribution for the definitions of the names in the context of the language grammar. The specific numeric values which the names map to may change between Python versions.

The module also provides a mapping from numeric codes to names and some functions. The functions mirror definitions in the Python C header files.

token.tok_name
Dictionary mapping the numeric values of the constants defined in this module back to name strings, allowing more human-readable representation of parse trees to be generated.

token.ISTERMINAL(x)
Return true for terminal token values.

token.ISNONTERMINAL(x)
Return true for non-terminal token values.

token.ISEOF(x)
Return true if x is the marker indicating the end of input.

The token constants are:

token.ENDMARKER
token.NAME
token.NUMBER
token.STRING
token.NEWLINE
token.INDENT
token.DEDENT
token.LPAR
token.RPAR
token.LSQB
token.RSQB
token.COLON
token.COMMA
token.SEMI
token.PLUS
token.MINUS
token.STAR
token.SLASH
token.VBAR
token.AMPER
token.LESS
token.GREATER
token.EQUAL
token.DOT
token.PERCENT
token.LBRACE
token.RBRACE
token.EQEQUAL
token. NOTEQUAL
token. LESSEQUAL
token. GREATEREQUAL
token. TILDE
token. CIRCUMFLEX
token. LEFTSHIFT
token. RIGHTSHIFT
token. DOUBLESTAR
token. PLUSEQUAL
token. MINEQUAL
token. STAREQUAL
token. SLASHEQUAL
token. PERCENTEQUAL
token. AMPEREQUAL
token. VBAREQUAL
token. CIRCUMFLEXEQUAL
token. LEFTSHIFTEQUAL
token. RIGHTSHIFTEQUAL
token. DOUBLESTAREQUAL
token. DOUBLESLASH
token. DOUBLESLASHEQUAL
token. AT
token. RARROW
token. ELLIPSIS
token. OP
token. ERRORTOKEN
token. N_TOKENS
token. NT_OFFSET

See Also:
Module parser  The second example for the parser module shows how to use the symbol module.

31.6 keyword — Testing for Python keywords

Source code: Lib/keyword.py

This module allows a Python program to determine if a string is a keyword.

```python
def iskeyword(s):
    return s in kwlist
```

The `iskeyword` function takes a string `s` and returns `True` if `s` is a Python keyword.

```python
def kwlist:
    return ['keyword', 'import', 'return', ...]
```

The `kwlist` function is a sequence containing all the keywords defined for the interpreter. If any keywords are defined to only be active when particular `__future__` statements are in effect, these will be included as well.

31.7 tokenize — Tokenizer for Python source

Source code: Lib/tokenize.py

The `tokenize` module provides a lexical scanner for Python source code, implemented in Python. The scanner in this module returns comments as tokens as well, making it useful for implementing “pretty-printers,” including colorizers for on-screen displays.
To simplify token stream handling, all operators and delimiters tokens are returned using the generic `token.OP` token type. The exact type can be determined by checking the `exact_type` property on the named tuple returned from `tokenize.tokenize()`.

### 31.7.1 Tokenizing Input

The primary entry point is a generator:

```python
tokenize.tokenize(readline)
```

The `tokenize()` generator requires one argument, `readline`, which must be a callable object which provides the same interface as the `io.IOBase.readline()` method of file objects. Each call to the function should return one line of input as bytes.

The generator produces 5-tuples with these members: the token type; the token string; a 2-tuple `(srow, scol)` of ints specifying the row and column where the token begins in the source; a 2-tuple `(erow, ecol)` of ints specifying the row and column where the token ends in the source; and the line on which the token was found. The line passed (the last tuple item) is the logical line; continuation lines are included. The 5 tuple is returned as a named tuple with the field names: `type string start end line`.

The returned `named tuple` has a additional property named `exact_type` that contains the exact operator type for `token.OP` tokens. For all other token types `exact_type` equals the named tuple `type` field. Changed in version 3.1: Added support for named tuples.Changed in version 3.3: Added support for `exact_type`. `tokenize()` determines the source encoding of the file by looking for a UTF-8 BOM or encoding cookie, according to PEP 263.

All constants from the `token` module are also exported from `tokenize`, as are three additional token type values:

```python
tokenize.COMMENT
    Token value used to indicate a comment.

tokenize.NL
    Token value used to indicate a non-terminating newline. The NEWLINE token indicates the end of a logical line of Python code; NL tokens are generated when a logical line of code is continued over multiple physical lines.

tokenize.ENCODING
    Token value that indicates the encoding used to decode the source bytes into text. The first token returned by `tokenize()` will always be an ENCODING token.
```

Another function is provided to reverse the tokenization process. This is useful for creating tools that tokenize a script, modify the token stream, and write back the modified script.

```python
tokenize.untokenize(iterable)
```

Converts tokens back into Python source code. The `iterable` must return sequences with at least two elements, the token type and the token string. Any additional sequence elements are ignored.

The reconstructed script is returned as a single string. The result is guaranteed to tokenize back to match the input so that the conversion is lossless and round-trips are assured. The guarantee applies only to the token type and token string as the spacing between tokens (column positions) may change.

It returns bytes, encoded using the ENCODING token, which is the first token sequence output by `tokenize()`.

`tokenize()` needs to detect the encoding of source files it tokenizes. The function it uses to do this is available:

```python
tokenize.detect_encoding(readline)
```

The `detect_encoding()` function is used to detect the encoding that should be used to decode a Python source file. It requires one argument, `readline`, in the same way as the `tokenize()` generator.

It will call `readline` a maximum of twice, and return the encoding used (as a string) and a list of any lines (not decoded from bytes) it has read in.
It detects the encoding from the presence of a UTF-8 BOM or an encoding cookie as specified in PEP 263. If both a BOM and a cookie are present, but disagree, a SyntaxError will be raised. Note that if the BOM is found, ‘utf-8-sig’ will be returned as an encoding.

If no encoding is specified, then the default of ‘utf-8’ will be returned.

Use open() to open Python source files: it uses detect_encoding() to detect the file encoding.

tokenize.open(filename)

Open a file in read only mode using the encoding detected by detect_encoding(). New in version 3.2.

31.7.2 Command-Line Usage

New in version 3.3. The tokenize module can be executed as a script from the command line. It is as simple as:

```
python -m tokenize [-e] [filename.py]
```

The following options are accepted:

- `-h`, `-help`
  show this help message and exit

- `-e`, `-exact`
  display token names using the exact type

If `filename.py` is specified its contents are tokenized to stdout. Otherwise, tokenization is performed on stdin.

31.7.3 Examples

Example of a script rewriter that transforms float literals into Decimal objects:

```python
from tokenize import tokenize, untokenize, NUMBER, STRING, NAME, OP
from io import BytesIO

def decistmt(s):
    """Substitute Decimals for floats in a string of statements."

    >>> from decimal import Decimal
    >>> s = 'print(+21.3e-5**.1234/81.7)'
    >>> decistmt(s)
    "print (+Decimal ('21.3e-5')*Decimal ('.1234')/Decimal ('81.7'))"

    The format of the exponent is inherited from the platform C library.
    Known cases are "e-007" (Windows) and "e-07" (not Windows). Since
    we're only showing 12 digits, and the 13th isn't close to 5, the
    rest of the output should be platform-independent.

    >>> exec(s) #doctest: +ELLIPSIS
    -3.21716034272e-0...7

    Output from calculations with Decimal should be identical across all
    platforms.

    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7
    """

    result = []
g = tokenize(BytesIO(s.encode('utf-8')).readline) # tokenize the string
for toknum, tokval, _, _, _ in g:
The Python Library Reference, Release 3.3.3

```python
if toknum == NUMBER and '.' in tokval:  # replace NUMBER tokens
    result.extend([
        (NAME, 'Decimal'),
        (OP, '('),
        (STRING, repr(tokval)),
        (OP, ')')
    ])
else:
    result.append((toknum, tokval))
return untokenize(result).decode('utf-8')
```

Example of tokenizing from the command line. The script:

```python
def say_hello():
    print("Hello, World!")
```

will be tokenized to the following output where the first column is the range of the line/column coordinates where the token is found, the second column is the name of the token, and the final column is the value of the token (if any):

$ python -m tokenize hello.py
0,0-0,0: ENCODING 'utf-8'
1,0-1,3: NAME 'def'
1,4-1,13: NAME 'say_hello'
1,13-1,14: OP '('
1,14-1,15: OP ')'
1,15-1,16: OP ':'
1,16-1,17: NEWLINE '\n'
2,0-2,4: INDENT ' '
2,4-2,9: NAME 'print'
2,9-2,10: OP '('
2,10-2,25: STRING '"Hello, World!"'
2,25-2,26: OP ')''
2,26-2,27: NEWLINE '\n'
3,0-3,1: NL '\n'
4,0-4,0: DEDENT ''
4,0-4,11: OP ')'
4,11-4,12: NEWLINE '\n'
5,0-5,0: ENDMARKER ''

The exact token type names can be displayed using the `-e` option:

$ python -m tokenize -e hello.py
0,0-0,0: ENCODING 'utf-8'
1,0-1,3: NAME 'def'
1,4-1,13: NAME 'say_hello'
1,13-1,14: LPAR '('
1,14-1,15: RPAR ')'
1,15-1,16: COLON ':'
1,16-1,17: NEWLINE '\n'
2,0-2,4: INDENT ' '
2,4-2,9: NAME 'print'
2,9-2,10: LPAR '('
2,10-2,25: STRING '"Hello, World!"'
2,25-2,26: RPAR ')'
2,26-2,27: NEWLINE '\n'
3,0-3,1: NL '\n'
```

Chapter 31. Python Language Services
Source code: Lib/tabnanny.py

For the time being this module is intended to be called as a script. However it is possible to import it into an IDE and use the function `check()` described below.

Note: The API provided by this module is likely to change in future releases; such changes may not be backward compatible.

tabnanny.check (file_or_dir)
If `file_or_dir` is a directory and not a symbolic link, then recursively descend the directory tree named by `file_or_dir`, checking all `.py` files along the way. If `file_or_dir` is an ordinary Python source file, it is checked for whitespace related problems. The diagnostic messages are written to standard output using the `print()` function.

tabnanny.verbose
Flag indicating whether to print verbose messages. This is incremented by the `-v` option if called as a script.

tabnanny.filename_only
Flag indicating whether to print only the filenames of files containing whitespace related problems. This is set to true by the `-q` option if called as a script.

exception tabnanny.NannyNag
Raised by `tokeneater()` if detecting an ambiguous indent. Captured and handled in `check()`.

tabnanny.tokeneater (type, token, start, end, line)
This function is used by `check()` as a callback parameter to the function `tokenize.tokenize()`.

See Also:

Module tokenize Lexical scanner for Python source code.

31.9 pycbr — Python class browser support

Source code: Lib/pyclbr.py

The `pyclbr` module can be used to determine some limited information about the classes, methods and top-level functions defined in a module. The information provided is sufficient to implement a traditional three-pane class browser. The information is extracted from the source code rather than by importing the module, so this module is safe to use with untrusted code. This restriction makes it impossible to use this module with modules not implemented in Python, including all standard and optional extension modules.

pyclbr.readmodule (module, path=None)
Read a module and return a dictionary mapping class names to class descriptor objects. The parameter `module` should be the name of a module as a string; it may be the name of a module within a package. The `path` parameter should be a sequence, and is used to augment the value of `sys.path`, which is used to locate module source code.
pyclbr.readmodule_ex(module, path=None)

Like readmodule(), but the returned dictionary, in addition to mapping class names to class descriptor objects, also maps top-level function names to function descriptor objects. Moreover, if the module being read is a package, the key '__path__' in the returned dictionary has as its value a list which contains the package search path.

### 31.9.1 Class Objects

The Class objects used as values in the dictionary returned by readmodule() and readmodule_ex() provide the following data attributes:

- **Class.module**
  The name of the module defining the class described by the class descriptor.

- **Class.name**
  The name of the class.

- **Class.super**
  A list of Class objects which describe the immediate base classes of the class being described. Classes which are named as superclasses but which are not discoverable by readmodule() are listed as a string with the class name instead of as Class objects.

- **Class.methods**
  A dictionary mapping method names to line numbers.

- **Class.file**
  Name of the file containing the class statement defining the class.

- **Class.lineno**
  The line number of the class statement within the file named by file.

### 31.9.2 Function Objects

The Function objects used as values in the dictionary returned by readmodule_ex() provide the following attributes:

- **Function.module**
  The name of the module defining the function described by the function descriptor.

- **Function.name**
  The name of the function.

- **Function.file**
  Name of the file containing the def statement defining the function.

- **Function.lineno**
  The line number of the def statement within the file named by file.

### 31.10 py_compile — Compile Python source files

Source code: Lib/py_compile.py

The py_compile module provides a function to generate a byte-code file from a source file, and another function used when the module source file is invoked as a script.

Though not often needed, this function can be useful when installing modules for shared use, especially if some of the users may not have permission to write the byte-code cache files in the directory containing the source code.

**exception py_compile.PyCompileError**

Exception raised when an error occurs while attempting to compile the file.
py_compile.compile(file, cfile=None, dfile=None, doraise=False, optimize=-1)

Compile a source file to byte-code and write out the byte-code cache file. The source code is loaded from the file name file. The byte-code is written to cfile, which defaults to the PEP 3147 path, ending in .pyc (.pyo if optimization is enabled in the current interpreter). For example, if file is /foo/bar/baz.py, cfile will default to /foo/bar/__pycache__/baz.cpython-32.pyc for Python 3.2. If dfile is specified, it is used as the name of the source file in error messages when instead of file, if doraise is true, a PyCompileError is raised when an error is encountered while compiling file. If doraise is false (the default), an error string is written to sys.stderr, but no exception is raised. This function returns the path to byte-compiled file, i.e. whatever cfile value was used.

optimize controls the optimization level and is passed to the built-in compile() function. The default of -1 selects the optimization level of the current interpreter. Changed in version 3.2: Changed default value of cfile to be PEP 3147-compliant. Previous default was file + ‘c’ (‘o’ if optimization was enabled). Also added the optimize parameter.

py_compile.main(args=None)

Compile several source files. The files named in args (or on the command line, if args is None) are compiled and the resulting bytecode is cached in the normal manner. This function does not search a directory structure to locate source files; it only compiles files named explicitly. If ‘-’ is the only parameter in args, the list of files is taken from standard input. Changed in version 3.2: Added support for ‘-’.

When this module is run as a script, the main() is used to compile all the files named on the command line. The exit status is nonzero if one of the files could not be compiled.

See Also:

Module compileall Utilities to compile all Python source files in a directory tree.

31.11 compileall — Byte-compile Python libraries

This module provides some utility functions to support installing Python libraries. These functions compile Python source files in a directory tree. This module can be used to create the cached byte-code files at library installation time, which makes them available for use even by users who don’t have write permission to the library directories.

31.11.1 Command-line use

This module can work as a script (using python -m compileall) to compile Python sources.

(directory|file)...

Positional arguments are files to compile or directories that contain source files, traversed recursively. If no argument is given, behave as if the command line was -l <directories from sys.path>.

-l

Do not recurse into subdirectories, only compile source code files directly contained in the named or implied directories.

-f

Force rebuild even if timestamps are up-to-date.

-q

Do not print the list of files compiled, print only error messages.

-d destdir

Directory prepended to the path to each file being compiled. This will appear in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

-x regex

regex is used to search the full path to each file considered for compilation, and if the regex produces a match, the file is skipped.
-i list
   Read the file list and add each line that it contains to the list of files and directories to compile. If list
   is -, read lines from stdin.

-b
   Write the byte-code files to their legacy locations and names, which may overwrite byte-code files created
   by another version of Python. The default is to write files to their PEP 3147 locations and names, which
   allows byte-code files from multiple versions of Python to coexist.

Changed in version 3.2: Added the -i, -b and -h options. There is no command-line option to control the
optimization level used by the compile() function, because the Python interpreter itself already provides the
option: python -O -m compileall.

31.11.2 Public functions

compileall.compile_dir(dir, maxlevels=10, ddir=None, force=False, rx=None, quiet=False,
   legacy=False, optimize=-1)
   Recursively descend the directory tree named by dir, compiling all .py files along the way.

   The maxlevels parameter is used to limit the depth of the recursion; it defaults to 10.

   If ddir is given, it is prepended to the path to each file being compiled for use in compilation time tracebacks,
   and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases
   where the source file does not exist at the time the byte-code file is executed.

   If force is true, modules are re-compiled even if the timestamps are up to date.

   If rx is given, its search method is called on the complete path to each file considered for compilation, and
   if it returns a true value, the file is skipped.

   If quiet is true, nothing is printed to the standard output unless errors occur.

   If legacy is true, byte-code files are written to their legacy locations and names, which may overwrite byte-
   code files created by another version of Python. The default is to write files to their PEP 3147 locations and
   names, which allows byte-code files from multiple versions of Python to coexist.

   optimize specifies the optimization level for the compiler. It is passed to the built-in compile() function.
   Changed in version 3.2: Added the legacy and optimize parameter.

compileall.compile_file(fullname, ddir=None, force=False, rx=None, quiet=False,
   legacy=False, optimize=-1)
   Compile the file with path fullname.

   If ddir is given, it is prepended to the path to the file being compiled for use in compilation time tracebacks,
   and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases
   where the source file does not exist at the time the byte-code file is executed.

   If rx is given, its search method is passed the full path name to the file being compiled, and if it returns a
   true value, the file is not compiled and True is returned.

   If quiet is true, nothing is printed to the standard output unless errors occur.

   If legacy is true, byte-code files are written to their legacy locations and names, which may overwrite byte-
   code files created by another version of Python. The default is to write files to their PEP 3147 locations and
   names, which allows byte-code files from multiple versions of Python to coexist.

   optimize specifies the optimization level for the compiler. It is passed to the built-in compile() function.
   New in version 3.2.

compileall.compile_path(skip_curdir=True, maxlevels=0, force=False, legacy=False,
   optimize=-1)
   Byte-compile all the .py files found along sys.path. If skip_curdir is true (the default), the current
directory is not included in the search. All other parameters are passed to the compile_dir() function.
   Note that unlike the other compile functions, maxlevels defaults to 0. Changed in version 3.2: Added
   the legacy and optimize parameter.
To force a recompile of all the .py files in the Lib/ subdirectory and all its subdirectories:

```python
import compileall

compileall.compile_dir('Lib/', force=True)
```

# Perform same compilation, excluding files in .svn directories.

```python
import re

compileall.compile_dir('Lib/', rx=re.compile(r'[\/]\[.\]svn'), force=True)
```

See Also:

Module `py_compile` Byte-compile a single source file.

### 31.12 dis — Disassembler for Python bytecode

**Source code:** Lib/dis.py

The `dis` module supports the analysis of CPython bytecode by disassembling it. The CPython bytecode which this module takes as an input is defined in the file `Include/opcode.h` and used by the compiler and the interpreter.

**CPython implementation detail:** Bytecode is an implementation detail of the CPython interpreter. No guarantees are made that bytecode will not be added, removed, or changed between versions of Python. Use of this module should not be considered to work across Python VMs or Python releases.

Example: Given the function `myfunc()`:

```python
def myfunc(alist):
    return len(alist)
```

the following command can be used to get the disassembly of `myfunc()`:

```python
>>> dis.dis(myfunc)
   2 0 LOAD_GLOBAL       0 (len)
   3  LOAD_FAST          0 (alist)
   6  CALL_FUNCTION      1
   9  RETURN_VALUE
```

(The “2” is a line number).

The `dis` module defines the following functions and constants:

```python
dis.code_info(x)
```
Return a formatted multi-line string with detailed code object information for the supplied function, method, source code string or code object.

Note that the exact contents of code info strings are highly implementation dependent and they may change arbitrarily across Python VMs or Python releases. New in version 3.2.

```python
dis.show_code(x)
```
Print detailed code object information for the supplied function, method, source code string or code object to stdout.

This is a convenient shorthand for `print(code_info(x))`, intended for interactive exploration at the interpreter prompt. New in version 3.2.

```python
dis.dis(x=None)
```
Disassemble the x object. x can denote either a module, a class, a method, a function, a code object, a string of source code or a byte sequence of raw bytecode. For a module, it disassembles all functions. For a class, it disassembles all methods. For a code object or sequence of raw bytecode, it prints one line per bytecode instruction. Strings are first compiled to code objects with the `compile()` built-in function before being disassembled. If no object is provided, this function disassembles the last traceback.
Disassemble the top-of-stack function of a traceback, using the last traceback if none was passed. The instruction causing the exception is indicated.

Disassemble a code object, indicating the last instruction if lasti was provided. The output is divided in the following columns:

1. the line number, for the first instruction of each line
2. the current instruction, indicated as -->
3. a labelled instruction, indicated with >>
4. the address of the instruction,
5. the operation code name,
6. operation parameters, and
7. interpretation of the parameters in parentheses.

The parameter interpretation recognizes local and global variable names, constant values, branch targets, and compare operators.

This generator function uses the co_firstlineno and co_lnotab attributes of the code object code to find the offsets which are starts of lines in the source code. They are generated as (offset, lineno) pairs.

Detect all offsets in the code object code which are jump targets, and return a list of these offsets.

Sequence of operation names, indexable using the bytecode.

Dictionary mapping operation names to bytecodes.

Sequence of all compare operation names.

Sequence of bytecodes that have a constant parameter.

Sequence of bytecodes that access a free variable.

Sequence of bytecodes that access an attribute by name.

Sequence of bytecodes that have a relative jump target.

Sequence of bytecodes that have an absolute jump target.

Sequence of bytecodes that access a local variable.

Sequence of bytecodes of Boolean operations.

### 31.12.1 Python Bytecode Instructions

The Python compiler currently generates the following bytecode instructions.
General instructions

NOP
Do nothing code. Used as a placeholder by the bytecode optimizer.

POP_TOP
Removes the top-of-stack (TOS) item.

ROT_TWO
Swaps the two top-most stack items.

ROT_THREE
Lifts second and third stack item one position up, moves top down to position three.

DUP_TOP
Duplicates the reference on top of the stack.

DUP_TOP_TWO
Duplicates the two references on top of the stack, leaving them in the same order.

Unary operations

Unary operations take the top of the stack, apply the operation, and push the result back on the stack.

UNARY_POSITIVE
Implements \( TOS = +TOS \).

UNARY_NEGATIVE
Implements \( TOS = -TOS \).

UNARY_NOT
Implements \( TOS = \text{not } TOS \).

UNARY_INVERT
Implements \( TOS = \sim TOS \).

GET_ITER
Implements \( TOS = \text{iter}(TOS) \).

Binary operations

Binary operations remove the top of the stack (TOS) and the second top-most stack item (TOS1) from the stack. They perform the operation, and put the result back on the stack.

BINARY_POWER
Implements \( TOS = TOS1 ** TOS \).

BINARY_MULTIPLY
Implements \( TOS = TOS1 * TOS \).

BINARY_FLOOR_DIVIDE
Implements \( TOS = TOS1 // TOS \).

BINARY_TRUE_DIVIDE
Implements \( TOS = TOS1 / TOS \).

BINARY_MODULO
Implements \( TOS = TOS1 \% TOS \).

BINARY_ADD
Implements \( TOS = TOS1 + TOS \).

BINARY_SUBTRACT
Implements \( TOS = TOS1 - TOS \).

BINARY_SUBSCR
Implements \( TOS = TOS1[TOS] \).

BINARY_LSHIFT
Implements \( TOS = TOS1 \ll TOS \).
BINARY_RSHIFT
    Implements TOS = TOS1 >> TOS.

BINARY_AND
    Implements TOS = TOS1 & TOS.

BINARY_XOR
    Implements TOS = TOS1 ^ TOS.

BINARY_OR
    Implements TOS = TOS1 | TOS.

In-place operations
In-place operations are like binary operations, in that they remove TOS and TOS1, and push the result back on the stack, but the operation is done in-place when TOS1 supports it, and the resulting TOS may be (but does not have to be) the original TOS1.

INPLACE_POWER
    Implements in-place TOS = TOS1 ** TOS.

INPLACE_MULTIPLY
    Implements in-place TOS = TOS1 * TOS.

INPLACE_FLOOR_DIVIDE
    Implements in-place TOS = TOS1 // TOS.

INPLACE_TRUE_DIVIDE
    Implements in-place TOS = TOS1 / TOS.

INPLACE_MODULO
    Implements in-place TOS = TOS1 % TOS.

INPLACE_ADD
    Implements in-place TOS = TOS1 + TOS.

INPLACE_SUBTRACT
    Implements in-place TOS = TOS1 - TOS.

INPLACE_LSHIFT
    Implements in-place TOS = TOS1 << TOS.

INPLACE_RSHIFT
    Implements in-place TOS = TOS1 >> TOS.

INPLACE_AND
    Implements in-place TOS = TOS1 & TOS.

INPLACE_XOR
    Implements in-place TOS = TOS1 ^ TOS.

INPLACE_OR
    Implements in-place TOS = TOS1 | TOS.

STORE_SUBSCR
    Implements TOS1[TOS] = TOS2.

DELETE_SUBSCR
    Implements del TOS1[TOS].

Miscellaneous opcodes

PRINT_EXPR
    Implements the expression statement for the interactive mode. TOS is removed from the stack and printed.
    In non-interactive mode, an expression statement is terminated with POP_STACK.

BREAK_LOOP
    Terminates a loop due to a break statement.
CONTINUE_LOOP (target)
Continues a loop due to a continue statement. target is the address to jump to (which should be a FOR_ITER instruction).

SET_ADD (i)
Calls set.add(TOS1[-i], TOS). Used to implement set comprehensions.

LIST_APPEND (i)
Calls list.append(TOS[-i], TOS). Used to implement list comprehensions.

MAP_ADD (i)
Calls dict.setitem(TOS1[-i], TOS, TOS1). Used to implement dict comprehensions.

For all of the SET_ADD, LIST_APPEND and MAP_ADD instructions, while the added value or key/value pair is popped off, the container object remains on the stack so that it is available for further iterations of the loop.

RETURN_VALUE
Returns with TOS to the caller of the function.

YIELD_VALUE
Pops TOS and yields it from a generator.

YIELD_FROM
Pops TOS and delegates to it as a subiterator from a generator. New in version 3.3.

IMPORT_STAR
Loads all symbols not starting with ‘_’ directly from the module TOS to the local namespace. The module is popped after loading all names. This opcode implements from module import *.

POP_BLOCK
Removes one block from the block stack. Per frame, there is a stack of blocks, denoting nested loops, try statements, and such.

POP_EXCEPT
Removes one block from the block stack. The popped block must be an exception handler block, as implicitly created when entering an except handler. In addition to popping extraneous values from the frame stack, the last three popped values are used to restore the exception state.

END_FINALLY
Terminates a finally clause. The interpreter recalls whether the exception has to be re-raised, or whether the function returns, and continues with the outer-next block.

LOAD_BUILD_CLASS
Pushes builtins.__build_class__() onto the stack. It is later called by CALL_FUNCTION to construct a class.

SETUP_WITH (delta)
This opcode performs several operations before a with block starts. First, it loads __exit__() from the context manager and pushes it onto the stack for later use by WITH_CLEANUP. Then, __enter__() is called, and a finally block pointing to delta is pushed. Finally, the result of calling the enter method is pushed onto the stack. The next opcode will either ignore it (POP_TOP), or store it in (a) variable(s) (STORE_FAST, STORE_NAME, or UNPACK_SEQUENCE).

WITH_CLEANUP
 Cleans up the stack when a with statement block exits. TOS is the context manager’s __exit__() bound method. Below TOS are 1–3 values indicating how/why the finally clause was entered:

•SECOND = None
•(SECOND, THIRD) = (WHY_{RETURN, CONTINUE}), retval
•SECOND = WHY_*; no retval below it
•(SECOND, THIRD, FOURTH) = exc_info()

In the last case, TOS(SECOND, THIRD, FOURTH) is called, otherwise TOS(None, None, None). In addition, TOS is removed from the stack.
If the stack represents an exception, and the function call returns a ‘true’ value, this information is "zapped" and replaced with a single WHY_SILENCED to prevent END_FINALLY from re-raising the exception. (But non-local gotos will still be resumed.)

**STORE_LOCALS**

Pops TOS from the stack and stores it as the current frame’s _f_locals. This is used in class construction.

All of the following opcodes expect arguments. An argument is two bytes, with the more significant byte last.

**STORE_NAME**(namei)

Implements name = TOS, where namei is the index of name in the attribute _co_names of the code object. The compiler tries to use STORE_FAST or STORE_GLOBAL if possible.

**DELETE_NAME**(namei)

Implements del name, where namei is the index into _co_names attribute of the code object.

**UNPACK_SEQUENCE**(count)

Unpacks TOS into count individual values, which are put onto the stack right-to-left.

**UNPACK_EX**(counts)

Implements assignment with a starred target: Unpacks an iterable in TOS into individual values, where the total number of values can be smaller than the number of items in the iterable: one the new values will be a list of all leftover items.

The low byte of counts is the number of values before the list value, the high byte of counts the number of values after it. The resulting values are put onto the stack right-to-left.

**STORE_ATTR**(namei)

Implements TOS.name = TOS1, where namei is the index of name in _co_names.

**DELETE_ATTR**(namei)

Implements del TOS.name, using namei as index into _co_names.

**STORE_GLOBAL**(namei)

Works as STORE_NAME, but stores the name as a global.

**DELETE_GLOBAL**(namei)

Works as DELETE_NAME, but deletes a global name.

**BUILD_TUPLE**(count)

Creates a tuple consuming count items from the stack, and pushes the resulting tuple onto the stack.

**BUILD_LIST**(count)

Works as BUILD_TUPLE, but creates a list.

**BUILD_SET**(count)

Works as BUILD_TUPLE, but creates a set.

**BUILD_MAP**(count)

Pushes a new dictionary object onto the stack. The dictionary is pre-sized to hold count entries.

**LOAD_ATTR**(namei)

Replaces TOS with getattr(TOS, _co_names[namei]).

**COMPARE_OP**(opname)

Performs a Boolean operation. The operation name can be found in _cmp_op[opname].

**IMPORT_NAME**(namei)

Imports the module _co_names[namei]. TOS and TOS1 are popped and provide the fromlist and level arguments of __import__(). The module object is pushed onto the stack. The current namespace is not affected: for a proper import statement, a subsequent STORE_FAST instruction modifies the namespace.
IMPORT_FROM (namei)
Loads the attribute co_names[namei] from the module found in TOS. The resulting object is pushed onto the stack, to be subsequently stored by a STORE_FAST instruction.

JUMP_FORWARD (delta)
Increments bytecode counter by delta.

POP_JUMP_IF_TRUE (target)
If TOS is true, sets the bytecode counter to target. TOS is popped.

POP_JUMP_IF_FALSE (target)
If TOS is false, sets the bytecode counter to target. TOS is popped.

JUMP_IF_TRUE_OR_POP (target)
If TOS is true, sets the bytecode counter to target and leaves TOS on the stack. Otherwise (TOS is false), TOS is popped.

JUMP_IF_FALSE_OR_POP (target)
If TOS is false, sets the bytecode counter to target and leaves TOS on the stack. Otherwise (TOS is true), TOS is popped.

JUMP_ABSOLUTE (target)
Set bytecode counter to target.

FOR_ITER (delta)
TOS is an iterator. Call its __next__() method. If this yields a new value, push it on the stack (leaving the iterator below it). If the iterator indicates it is exhausted TOS is popped, and the byte code counter is incremented by delta.

LOAD_GLOBAL (namei)
Loads the global named co_names[namei] onto the stack.

SETUP_LOOP (delta)
Pushes a block for a loop onto the block stack. The block spans from the current instruction with a size of delta bytes.

SETUP_EXCEPT (delta)
Pushes a try block from a try-except clause onto the block stack. delta points to the first except block.

SETUP_FINALLY (delta)
Pushes a try block from a try-except clause onto the block stack. delta points to the finally block.

STORE_MAP
Store a key and value pair in a dictionary. Pops the key and value while leaving the dictionary on the stack.

LOAD_FAST (var_num)
Pushes a reference to the local co_varnames[var_num] onto the stack.

STORE_FAST (var_num)
Stores TOS into the local co_varnames[var_num].

DELETE_FAST (var_num)
Deletes local co_varnames[var_num].

LOAD_CLOSURE (i)
Pushes a reference to the cell contained in slot i of the cell and free variable storage. The name of the variable is co_cellvars[i] if i is less than the length of co_cellvars. Otherwise it is co_freevars[i - len(co_cellvars)].

LOAD_DEREF (i)
Loads the cell contained in slot i of the cell and free variable storage. Pushes a reference to the object the cell contains on the stack.

STORE_DEREF (i)
Stores TOS into the cell contained in slot i of the cell and free variable storage.

DELETE_DEREF (i)
Empties the cell contained in slot i of the cell and free variable storage. Used by the del statement.
RAISE_VARARGS (argc)

Raises an exception. \texttt{argc} indicates the number of parameters to the raise statement, ranging from 0 to 3. The handler will find the traceback as TOS2, the parameter as TOS1, and the exception as TOS.

CALL_FUNCTION (argc)

Calls a function. The low byte of \texttt{argc} indicates the number of positional parameters, the high byte the number of keyword parameters. On the stack, the opcode finds the keyword parameters first. For each keyword argument, the value is on top of the key. Below the keyword parameters, the positional parameters are on the stack, with the right-most parameter on top. Below the parameters, the function object to call is on the stack. Pops all function arguments, and the function itself off the stack, and pushes the return value.

MAKE_FUNCTION (argc)

Pushes a new function object on the stack. From bottom to top, the consumed stack must consist of

- \texttt{argc} & 0xFF default argument objects in positional order
- \texttt{(argc >> 8)} & 0xFF pairs of name and default argument, with the name just below the object on the stack, for keyword-only parameters
- \texttt{(argc >> 16)} & 0x7FFF parameter annotation objects
- a tuple listing the parameter names for the annotations (only if there are any annotation objects)
- the code associated with the function (at TOS1)
- the \texttt{qualified name} of the function (at TOS)

MAKE_CLOSURE (argc)

Creates a new function object, sets its \_\_closure\_\_ slot, and pushes it on the stack. TOS is the \texttt{qualified name} of the function, TOS1 is the code associated with the function, and TOS2 is the tuple containing cells for the closure’s free variables. The function also has \texttt{argc} default parameters, which are found below the cells.

BUILD_SLICE (argc)

Pushes a slice object on the stack. \texttt{argc} must be 2 or 3. If it is 2, \texttt{slice(TOS1, TOS)} is pushed; if it is 3, \texttt{slice(TOS2, TOS1, TOS)} is pushed. See the \texttt{slice()} built-in function for more information.

EXTENDED_ARG (ext)

Prefixes any opcode which has an argument too big to fit into the default two bytes. \texttt{ext} holds two additional bytes which, taken together with the subsequent opcode’s argument, comprise a four-byte argument, \texttt{ext} being the two most-significant bytes.

CALL_FUNCTION_VAR (argc)

Calls a function. \texttt{argc} is interpreted as in CALL_FUNCTION. The top element on the stack contains the variable argument list, followed by keyword and positional arguments.

CALL_FUNCTION_KW (argc)

Calls a function. \texttt{argc} is interpreted as in CALL_FUNCTION. The top element on the stack contains the keyword arguments dictionary, followed by explicit keyword and positional arguments.

CALL_FUNCTION_VAR_KW (argc)

Calls a function. \texttt{argc} is interpreted as in CALL_FUNCTION. The top element on the stack contains the keyword arguments dictionary, followed by the variable-arguments tuple, followed by explicit keyword and positional arguments.

HAVE_ARGUMENT

This is not really an opcode. It identifies the dividing line between opcodes which don’t take arguments < \texttt{HAVE_ARGUMENT} and those which do >= \texttt{HAVE_ARGUMENT}.

31.13 pickletools — Tools for pickle developers

Source code: Lib/pickletools.py
This module contains various constants relating to the intimate details of the pickle module, some lengthy comments about the implementation, and a few useful functions for analyzing pickled data. The contents of this module are useful for Python core developers who are working on the pickle; ordinary users of the pickle module probably won’t find the pickletools module relevant.

### 31.13.1 Command line usage

New in version 3.2. When invoked from the command line, `python -m pickletools` will disassemble the contents of one or more pickle files. Note that if you want to see the Python object stored in the pickle rather than the details of pickle format, you may want to use `-m pickle` instead. However, when the pickle file that you want to examine comes from an untrusted source, `-m pickletools` is a safer option because it does not execute pickle bytecode.

For example, with a tuple `(1, 2)` pickled in file `x.pickle`:

```
$ python -m pickle x.pickle
(1, 2)
$ python -m pickletools x.pickle
0: \x80 PROTO 3
2: K BININT1 1
4: K BININT1 2
6: \x86 TUPLE2
7: q BININPUT 0
9: . STOP
highest protocol among opcodes = 2
```

#### Command line options

- `-a`, `-annotate`
  Annotate each line with a short opcode description.

- `-o`, `-output=<file>`
  Name of a file where the output should be written.

- `-l`, `-indentlevel=<num>`
  The number of blanks by which to indent a new MARK level.

- `-m`, `-memo`
  When multiple objects are disassembled, preserve memo between disassemblies.

- `-p`, `-preamble=<preamble>`
  When more than one pickle file are specified, print given preamble before each disassembly.

### 31.13.2 Programmatic Interface

`pickletools.dis` *(pickle, out=None, memo=None, indentlevel=4, annotate=0)*

Outputs a symbolic disassembly of the pickle to the file-like object `out`, defaulting to `sys.stdout`. `pickle` can be a string or a file-like object. `memo` can be a Python dictionary that will be used as the pickle’s memo; it can be used to perform disassemblies across multiple pickles created by the same pickler. Successive levels, indicated by MARK opcodes in the stream, are indented by `indentlevel` spaces. If a nonzero value is given to `annotate`, each opcode in the output is annotated with a short description. The value of `annotate` is used as a hint for the column where annotation should start.

New in version 3.2: The `annotate` argument.

`pickletools.genops` *(pickle)*

Provides an iterator over all of the opcodes in a pickle, returning a sequence of `(opcode, arg, pos)`
triples. *opcode* is an instance of an *OpcodeInfo* class; *arg* is the decoded value, as a Python object, of the opcode’s argument; *pos* is the position at which this opcode is located. *pickle* can be a string or a file-like object.

**pickletools.optimize**(*picklestring*)

Returns a new equivalent pickle string after eliminating unused *PUT* opcodes. The optimized pickle is shorter, takes less transmission time, requires less storage space, and unpickles more efficiently.
CHAPTER

THIRTYTWO

MISCELLANEOUS SERVICES

The modules described in this chapter provide miscellaneous services that are available in all Python versions. Here’s an overview:

32.1 formatter — Generic output formatting

This module supports two interface definitions, each with multiple implementations: The formatter interface, and the writer interface which is required by the formatter interface.

Formatters objects transform an abstract flow of formatting events into specific output events on writer objects. Formatters manage several stack structures to allow various properties of a writer object to be changed and restored; writers need not be able to handle relative changes nor any sort of “change back” operation. Specific writer properties which may be controlled via formatter objects are horizontal alignment, font, and left margin indentations. A mechanism is provided which supports providing arbitrary, non-exclusive style settings to a writer as well. Additional interfaces facilitate formatting events which are not reversible, such as paragraph separation.

Writer objects encapsulate device interfaces. Abstract devices, such as file formats, are supported as well as physical devices. The provided implementations all work with abstract devices. The interface makes available mechanisms for setting the properties which formatter objects manage and inserting data into the output.

32.1.1 The Formatter Interface

Interfaces to create formatters are dependent on the specific formatter class being instantiated. The interfaces described below are the required interfaces which all formatters must support once initialized.

One data element is defined at the module level:

formatter.AS_IS
Value which can be used in the font specification passed to the push_font() method described below, or as the new value to any other push_property() method. Pushing the AS_IS value allows the corresponding pop_property() method to be called without having to track whether the property was changed.

The following attributes are defined for formatter instance objects:

formatter.writer
The writer instance with which the formatter interacts.

formatter.end_paragraph(blanklines)
Close any open paragraphs and insert at least blanklines before the next paragraph.

formatter.add_line_break()
Add a hard line break if one does not already exist. This does not break the logical paragraph.

formatter.add_hor_rule(*args, **kw)
Insert a horizontal rule in the output. A hard break is inserted if there is data in the current paragraph,
but the logical paragraph is not broken. The arguments and keywords are passed on to the writer’s send_line_break() method.

formatter.add_flow_data(data)
Provide data which should be formatted with collapsed whitespace. Whitespace from preceding and successive calls to add_flow_data() is considered as well when the whitespace collapse is performed. The data which is passed to this method is expected to be word-wrapped by the output device. Note that any word-wrapping still must be performed by the writer object due to the need to rely on device and font information.

formatter.add_literal_data(data)
Provide data which should be passed to the writer unchanged. Whitespace, including newline and tab characters, are considered legal in the value of data.

formatter.add_label_data(format, counter)
Insert a label which should be placed to the left of the current left margin. This should be used for constructing bulleted or numbered lists. If the format value is a string, it is interpreted as a format specification for counter, which should be an integer. The result of this formatting becomes the value of the label; if format is not a string it is used as the label value directly. The label value is passed as the only argument to the writer’s send_label_data() method. Interpretation of non-string label values is dependent on the associated writer.

Format specifications are strings which, in combination with a counter value, are used to compute label values. Each character in the format string is copied to the label value, with some characters recognized to indicate a transform on the counter value. Specifically, the character ‘1’ represents the counter value formatter as an Arabic number, the characters ‘A’ and ‘a’ represent alphabetic representations of the counter value in upper and lower case, respectively, and ‘I’ and ‘i’ represent the counter value in Roman numerals, in upper and lower case. Note that the alphabetic and roman transforms require that the counter value be greater than zero.

formatter.flush_softspace()
Send any pending whitespace buffered from a previous call to add_flow_data() to the associated writer object. This should be called before any direct manipulation of the writer object.

formatter.push_alignment(align)
Push a new alignment setting onto the alignment stack. This may be AS_IS if no change is desired. If the alignment value is changed from the previous setting, the writer’s new_alignment() method is called with the align value.

formatter.pop_alignment()
Restore the previous alignment.

formatter.push_font((size, italic, bold, teletype))
Change some or all font properties of the writer object. Properties which are not set to AS_IS are set to the values passed in while others are maintained at their current settings. The writer’s new_font() method is called with the fully resolved font specification.

formatter.pop_font()
Restore the previous font.

formatter.push_margin(margin)
Increase the number of left margin indentations by one, associating the logical tag margin with the new indentation. The initial margin level is 0. Changed values of the logical tag must be true values; false values other than AS_IS are not sufficient to change the margin.

formatter.pop_margin()
Restore the previous margin.

formatter.push_style(*styles)
Push any number of arbitrary style specifications. All styles are pushed onto the styles stack in order. A tuple representing the entire stack, including AS_IS values, is passed to the writer’s new_styles() method.

formatter.pop_style(n=1)
Pop the last n style specifications passed to push_style(). A tuple representing the revised stack,
including AS_IS values, is passed to the writer’s new_styles() method.

formatter.set_spacing(spacing)
Set the spacing style for the writer.

formatter.assert_line_data(flag=1)
Inform the formatter that data has been added to the current paragraph out-of-band. This should be used when the writer has been manipulated directly. The optional flag argument can be set to false if the writer manipulations produced a hard line break at the end of the output.

32.1.2 Formatter Implementations

Two implementations of formatter objects are provided by this module. Most applications may use one of these classes without modification or subclassing.

class formatter.NullFormatter (writer=None)
A formatter which does nothing. If writer is omitted, a NullWriter instance is created. No methods of the writer are called by NullFormatter instances. Implementations should inherit from this class if implementing a writer interface but don’t need to inherit any implementation.

class formatter.AbstractFormatter (writer)
The standard formatter. This implementation has demonstrated wide applicability to many writers, and may be used directly in most circumstances. It has been used to implement a full-featured World Wide Web browser.

32.1.3 The Writer Interface

Interfaces to create writers are dependent on the specific writer class being instantiated. The interfaces described below are the required interfaces which all writers must support once initialized. Note that while most applications can use the AbstractFormatter class as a formatter, the writer must typically be provided by the application.

writer.flush()
Flush any buffered output or device control events.

writer.new_alignment (align)
Set the alignment style. The align value can be any object, but by convention is a string or None, where None indicates that the writer’s “preferred” alignment should be used. Conventional align values are ‘left’, ‘center’, ‘right’, and ‘justify’.

writer.new_font (font)
Set the font style. The value of font will be None, indicating that the device’s default font should be used, or a tuple of the form (size, italic, bold, teletype). Size will be a string indicating the size of font that should be used; specific strings and their interpretation must be defined by the application. The italic, bold, and teletype values are Boolean values specifying which of those font attributes should be used.

writer.new_margin (margin, level)
Set the margin level to the integer level and the logical tag to margin. Interpretation of the logical tag is at the writer’s discretion; the only restriction on the value of the logical tag is that it not be a false value for non-zero values of level.

writer.new_spacing (spacing)
Set the spacing style to spacing.

writer.new_styles (styles)
Set additional styles. The styles value is a tuple of arbitrary values; the value AS_IS should be ignored. The styles tuple may be interpreted either as a set or as a stack depending on the requirements of the application and writer implementation.

writer.send_line_break()
Break the current line.
**32.1.4 Writer Implementations**

Three implementations of the writer object interface are provided as examples by this module. Most applications will need to derive new writer classes from the `NullWriter` class.

```python
class formatter.NullWriter
    A writer which only provides the interface definition; no actions are taken on any methods. This should be
    the base class for all writers which do not need to inherit any implementation methods.

class formatter.AbstractWriter
    A writer which can be used in debugging formatters, but not much else. Each method simply announces
    itself by printing its name and arguments on standard output.

class formatter.DumbWriter (file=None, maxcol=72)
    Simple writer class which writes output on the `file object` passed in as `file` or, if `file` is omitted, on standard
    output. The output is simply word-wrapped to the number of columns specified by `maxcol`. This class is
    suitable for reflowing a sequence of paragraphs.
```
This chapter describes modules that are only available on MS Windows platforms.

### 33.1 msilib — Read and write Microsoft Installer files

**Platforms:** Windows

The msilib supports the creation of Microsoft Installer (.msi) files. Because these files often contain an embedded “cabinet” file (.cab), it also exposes an API to create CAB files. Support for reading .cab files is currently not implemented; read support for the .msi database is possible.

This package aims to provide complete access to all tables in an .msi file, therefore, it is a fairly low-level API. Two primary applications of this package are the distutils command bdist_msi, and the creation of Python installer package itself (although that currently uses a different version of msilib).

The package contents can be roughly split into four parts: low-level CAB routines, low-level MSI routines, higher-level MSI routines, and standard table structures.

```python
msilib.FCICreate(cabname, files)
```

Create a new CAB file named `cabname`. `files` must be a list of tuples, each containing the name of the file on disk, and the name of the file inside the CAB file.

The files are added to the CAB file in the order they appear in the list. All files are added into a single CAB file, using the MSZIP compression algorithm.

Callbacks to Python for the various steps of MSI creation are currently not exposed.

```python
msilib.UuidCreate()
```

Return the string representation of a new unique identifier. This wraps the Windows API functions UuidCreate() and UuidToString().

```python
msilib.OpenDatabase(path, persist)
```

Return a new database object by calling MsiOpenDatabase. `path` is the file name of the MSI file; `persist` can be one of the constants MSIDBOPEN_CREATEDIRECT, MSIDBOPEN_CREATE, MSIDBOPEN_DIRECT, MSIDBOPEN_READONLY, or MSIDBOPEN_TRANSACT, and may include the flag MSIDBOPEN_PATCHFILE. See the Microsoft documentation for the meaning of these flags; depending on the flags, an existing database is opened, or a new one created.

```python
msilib.CreateRecord(count)
```

Return a new record object by calling MSICreateRecord(). `count` is the number of fields of the record.

```python
msilib.init_database(name, schema, ProductName, ProductCode, ProductVersion, Manufacturer)
```

Create and return a new database `name`, initialize it with `schema`, and set the properties `ProductName`, `ProductCode`, `ProductVersion`, and `Manufacturer`.

`schema` must be a module object containing `tables` and `_Validation_records` attributes; typically, `msilib.schema` should be used.

The database will contain just the schema and the validation records when this function returns.
msilib.add_data(database, table, records)
Add all records to the table named table in database.

The table argument must be one of the predefined tables in the MSI schema, e.g. 'Feature', 'File', 'Component', 'Dialog', 'Control', etc.

records should be a list of tuples, each one containing all fields of a record according to the schema of the table. For optional fields, None can be passed.

Field values can be ints, strings, or instances of the Binary class.

class msilib.Binary(filename)
Represents entries in the Binary table; inserting such an object using add_data() reads the file named filename into the table.

msilib.add_tables(database, module)
Add all table content from module to database. module must contain an attribute tables listing all tables for which content should be added, and one attribute per table that has the actual content.

This is typically used to install the sequence tables.

msilib.add_stream(database, name, path)
Add the file path into the _Stream table of database, with the stream name name.

msilib.gen_uuid()
Return a new UUID, in the format that MSI typically requires (i.e. in curly braces, and with all hexdigits in upper-case).

See Also:
FCICreateFile UuidCreate UuidToString

33.1.1 Database Objects

Database.OpenView(sql)
Return a view object, by calling MSIDatabaseOpenView(). sql is the SQL statement to execute.

Database.Commit()
Commit the changes pending in the current transaction, by calling MSIDatabaseCommit().

Database.GetSummaryInformation(count)
Return a new summary information object, by calling MsiGetSummaryInformation(). count is the maximum number of updated values.

See Also:
MSIDatabaseOpenView MSIDatabaseCommit MSIGetSummaryInformation

33.1.2 View Objects

View.Execute(params)
Execute the SQL query of the view, through MSIViewExecute(). If params is not None, it is a record describing actual values of the parameter tokens in the query.

View.GetColumnInfo(kind)
Return a record describing the columns of the view, through calling MsiViewGetColumnInfo(). kind can be either MSICOLINFO_NAMES or MSICOLINFO_TYPES.

View.Fetch()
Return a result record of the query, through calling MsiViewFetch().

View.Modify(kind, data)
Modify the view, by calling MsiViewModify(). kind can be one of MSIMODIFY_SEEK, MSIMODIFY_REFRESH, MSIMODIFY_INSERT, MSIMODIFY_UPDATE, MSIMODIFY_ASSIGN, MSIMODIFY_REPLACE, MSIMODIFY_MERGE, MSIMODIFY_DELETE,
data must be a record describing the new data.

View.Close()
Close the view, through MsiViewClose()

See Also:
MsiViewExecute MSIViewGetColumnInfo MsiViewFetch MsiViewModify MsiViewClose

33.1.3 Summary Information Objects

SummaryInformation.GetProperty(field)
Return a property of the summary, through MsiSummaryInfoGetProperty(). field is the name of the property, and can be one of the constants PID_CODEPAGE, PID_TITLE, PID_SUBJECT, PID_AUTHOR, PID_KEYWORDS, PID_COMMENTS, PID_TEMPLATE, PID_LASTAUTHOR, PID_REVNUMBER, PID_LASTPRINTED, PID_CREATE_DTM, PID_LASTSAVE_DTM, PID_PAGECOUNT, PID_WORDCOUNT, PID_CHARCOUNT, PID_APPNAME, or PID_SECURITY.

SummaryInformation.GetPropertyCount()
Return the number of summary properties, through MsiSummaryInfoGetPropertyCount().

SummaryInformation.SetProperty(field, value)
Set a property through MsiSummaryInfoSetProperty(). field can have the same values as in GetProperty(), value is the new value of the property. Possible value types are integer and string.

SummaryInformation.Persist()
Write the modified properties to the summary information stream, using MsiSummaryInfoPersist().

See Also:
MsiSummaryInfoGetProperty MsiSummaryInfoGetPropertyCount MsiSummaryInfoSetProperty MsiSummaryInfoPersist

33.1.4 Record Objects

Record.GetFieldCount()
Return the number of fields of the record, through MsiRecordGetFieldCount().

Record.GetInteger(field)
Return the value of field as an integer where possible. field must be an integer.

Record.GetString(field)
Return the value of field as a string where possible. field must be an integer.

Record.SetString(field, value)
Set field to value through MsiRecordSetString(). field must be an integer; value a string.

Record.SetStream(field, value)
Set field to the contents of the file named value, through MsiRecordSetStream(). field must be an integer, value a string.

Record.SetInteger(field, value)
Set field to value through MsiRecordSetInteger(). Both field and value must be an integer.

Record.ClearData()
Set all fields of the record to 0, through MsiRecordClearData().

See Also:
MsiRecordGetFieldCount MsiRecordSetString MsiRecordSetStream MsiRecordSetInteger MsiRecordClear
33.1.5 Errors

All wrappers around MSI functions raise `MsiError`; the string inside the exception will contain more detail.

33.1.6 CAB Objects

class `msilib.CAB` *(name)*

The class `CAB` represents a CAB file. During MSI construction, files will be added simultaneously to the `Files` table, and to a CAB file. Then, when all files have been added, the CAB file can be written, then added to the MSI file.

`name` is the name of the CAB file in the MSI file.

`append` *(full, file, logical)*

Add the file with the pathname `full` to the CAB file, under the name `logical`. If there is already a file named `logical`, a new file name is created.

Return the index of the file in the CAB file, and the new name of the file inside the CAB file.

`commit` *(database)*

Generate a CAB file, add it as a stream to the MSI file, put it into the `Media` table, and remove the generated file from the disk.

33.1.7 Directory Objects

class `msilib.Directory` *(database, cab, basedir, physical, logical, default[, componentflags ])*

Create a new directory in the Directory table. There is a current component at each point in time for the directory, which is either explicitly created through `start_component()`, or implicitly when files are added for the first time. Files are added into the current component, and into the cab file. To create a directory, a base directory object needs to be specified (can be `None`), the path to the physical directory, and a logical directory name. `default` specifies the `DefaultDir` slot in the directory table. `componentflags` specifies the default flags that new components get.

`start_component` *(component=None, feature=None, flags=None, keyfile=None, uuid=None)*

Add an entry to the Component table, and make this component the current component for this directory. If no component name is given, the directory name is used. If no `feature` is given, the current feature is used. If no `flags` are given, the directory’s default flags are used. If no `keyfile` is given, the KeyPath is left null in the Component table.

`add_file` *(file, src=None, version=None, language=None)*

Add a file to the current component of the directory, starting a new one if there is no current component. By default, the file name in the source and the file table will be identical. If the `src` file is specified, it is interpreted relative to the current directory. Optionally, a `version` and a `language` can be specified for the entry in the File table.

`glob` *(pattern, exclude=None)*

Add a list of files to the current component as specified in the `glob` pattern. Individual files can be excluded in the `exclude` list.

`remove_pyc()`

Remove `.pyc`/`.pyo` files on uninstall.

See Also:

Directory Table File Table Component Table Feature Components Table

33.1.8 Features

class `msilib.Feature` *(db, id, title, desc, display, level=1, parent=None, directory=None, attributes=0)*

Add a new record to the Feature table, using the values `id`, `parent.id`, `title`, `desc`, `display`, `level`, `direc-
tory, and attributes. The resulting feature object can be passed to the `start_component()` method of `Directory`.

```python
set_current()
```

Make this feature the current feature of `msilib`. New components are automatically added to the default feature, unless a feature is explicitly specified.

See Also:

Feature Table

### 33.1.9 GUI classes

`msilib` provides several classes that wrap the GUI tables in an MSI database. However, no standard user interface is provided; use `bdist_msi` to create MSI files with a user-interface for installing Python packages.

```python
class msilib.Control(dlg, name)
```

Base class of the dialog controls. `dlg` is the dialog object the control belongs to, and `name` is the control’s name.

```python
event(event, argument, condition=1, ordering=None)
```

Make an entry into the `ControlEvent` table for this control.

```python
mapping(event, attribute)
```

Make an entry into the `EventMapping` table for this control.

```python
condition(action, condition)
```

Make an entry into the `ControlCondition` table for this control.

```python
class msilib.RadioButtonGroup(dlg, name, property)
```

Create a radio button control named `name`. `property` is the installer property that gets set when a radio button is selected.

```python
add(name, x, y, width, height, text, value=None)
```

Add a radio button named `name` to the group, at the coordinates `x`, `y`, `width`, `height`, and with the label `text`. If `value` is `None`, it defaults to `name`.

```python
class msilib.Dialog(db, name, x, y, w, h, attr, title, first, default, cancel)
```

Return a new `Dialog` object. An entry in the `Dialog` table is made, with the specified coordinates, dialog attributes, title, name of the first, default, and cancel controls.

```python
control(name, type, x, y, width, height, attributes, property, text, control_next, help)
```

Return a new `Control` object. An entry in the `Control` table is made with the specified parameters. This is a generic method; for specific types, specialized methods are provided.

```python
text(name, x, y, width, height, attributes, text)
```

Add and return a `Text` control.

```python
bitmap(name, x, y, width, height, text)
```

Add and return a `Bitmap` control.

```python
line(name, x, y, width, height)
```

Add and return a `Line` control.

```python
pushbutton(name, x, y, width, height, attributes, text, next_control)
```

Add and return a `PushButton` control.

```python
radiogroup(name, x, y, width, height, attributes, property, text, next_control)
```

Add and return a `RadioButtonGroup` control.

```python
checkbox(name, x, y, width, height, attributes, property, text, next_control)
```

Add and return a `CheckBox` control.

See Also:
33.1.10 Precomputed tables

`msilib` provides a few subpackages that contain only schema and table definitions. Currently, these definitions are based on MSI version 2.0.

- `msilib.schema` This is the standard MSI schema for MSI 2.0, with the `tables` variable providing a list of table definitions, and `_Validation_records` providing the data for MSI validation.
- `msilib.sequence` This module contains table contents for the standard sequence tables: `AdminExecuteSequence`, `AdminUISequence`, `AdvExExecuteSequence`, `InstallExecuteSequence`, and `InstallUISequence`.
- `msilib.text` This module contains definitions for the UIText and ActionText tables, for the standard installer actions.

33.2 `msvcrt` – Useful routines from the MS VC++ runtime

*Platforms: Windows*

These functions provide access to some useful capabilities on Windows platforms. Some higher-level modules use these functions to build the Windows implementations of their services. For example, the `getpass` module uses this in the implementation of the `getpass()` function.

Further documentation on these functions can be found in the Platform API documentation.

The module implements both the normal and wide char variants of the console I/O api. The normal API deals only with ASCII characters and is of limited use for internationalized applications. The wide char API should be used wherever possible. Changed in version 3.3: Operations in this module now raise `OSError` where `IOError` was raised.

33.2.1 File Operations

- `msvcrt.locking(fd, mode, nbytes)`
  Lock part of a file based on file descriptor `fd` from the C runtime. Raises `OSError` on failure. The locked region of the file extends from the current file position for `nbytes` bytes, and may continue beyond the end of the file. `mode` must be one of the `LK_*` constants listed below. Multiple regions in a file may be locked at the same time, but may not overlap. Adjacent regions are not merged; they must be unlocked individually.

- `msvcrt.LK_LOCK`
- `msvcrt.LK_RLCK`
  Locks the specified bytes. If the bytes cannot be locked, the program immediately tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, `OSError` is raised.

- `msvcrt.LK_NBLCK`
- `msvcrt.LK_NBRLOCK`
  Locks the specified bytes. If the bytes cannot be locked, `OSError` is raised.

- `msvcrt.LK_UNLCK`
  Unlocks the specified bytes, which must have been previously locked.

- `msvcrt.setmode(fd, flags)`
  Set the line-end translation mode for the file descriptor `fd`. To set it to text mode, `flags` should be `os.O_TEXT`; for binary, it should be `os.O_BINARY`.
msvcrt.\open\_osfhandle (\textit{handle, flags})

Create a C runtime file descriptor from the file handle \textit{handle}. The \textit{flags} parameter should be a bitwise OR of \texttt{os.O\_APPEND}, \texttt{os.O\_RDONLY}, and \texttt{os.O\_TEXT}. The returned file descriptor may be used as a parameter to \texttt{os.fdopen()} to create a file object.

msvcrt.\get\_osfhandle (\textit{fd})

Return the file handle for the file descriptor \textit{fd}. Raises \texttt{OSError} if \textit{fd} is not recognized.

### 33.2.2 Console I/O

msvcrt.\kbhit()

Return true if a keypress is waiting to be read.

msvcrt.\getch()

Read a keypress and return the resulting character as a byte string. Nothing is echoed to the console. This call will block if a keypress is not already available, but will not wait for Enter to be pressed. If the pressed key was a special function key, this will return ‘\000’ or ‘\xe0’; the next call will return the keycode. The Control-C keypress cannot be read with this function.

msvcrt.\getwch()

Wide char variant of \texttt{getch()}, returning a Unicode value.

msvcrt.\getche()

Similar to \texttt{getch()}, but the keypress will be echoed if it represents a printable character.

msvcrt.\getwchar()

Wide char variant of \texttt{getche()}, returning a Unicode value.

msvcrt.\putch (\texttt{char})

Print the byte string \texttt{char} to the console without buffering.

msvcrt.\putwchar (\texttt{unicode\_char})

Wide char variant of \texttt{putch()}, accepting a Unicode value.

msvcrt.\ungetch (\texttt{char})

Cause the byte string \texttt{char} to be “pushed back” into the console buffer; it will be the next character read by \texttt{getch()} or \texttt{getche()}.

msvcrt.\ungetwchar (\texttt{unicode\_char})

Wide char variant of \texttt{ungetch()}, accepting a Unicode value.

### 33.2.3 Other Functions

msvcrt.\heapmin()

Force the \texttt{malloc()} heap to clean itself up and return unused blocks to the operating system. On failure, this raises \texttt{OSError}.

### 33.3 winreg – Windows registry access

\textit{Platforms:} Windows

These functions expose the Windows registry API to Python. Instead of using an integer as the registry handle, a \textit{handle object} is used to ensure that the handles are closed correctly, even if the programmer neglects to explicitly close them. Changed in version 3.3: Several functions in this module used to raise a \texttt{WindowsError}, which is now an alias of \texttt{OSError}.
33.3.1 Functions

This module offers the following functions:

`winreg.CloseKey(hkey)`
Closes a previously opened registry key. The `hkey` argument specifies a previously opened key.

**Note:** If `hkey` is not closed using this method (or via `hkey.Close()`), it is closed when the `hkey` object is destroyed by Python.

`winreg.ConnectRegistry(computer_name, key)`
Establishes a connection to a predefined registry handle on another computer, and returns a handle object. `computer_name` is the name of the remote computer, of the form `r'\computername'`. If `None`, the local computer is used.

`key` is the predefined handle to connect to.

The return value is the handle of the opened key. If the function fails, an `OSError` exception is raised. Changed in version 3.3: See above.

`winreg.CreateKey(key, sub_key)`
Creates or opens the specified key, returning a handle object.

`key` is an already open key, or one of the predefined `HKEY_*` constants.

`sub_key` is a string that names the key this method opens or creates.

If `key` is one of the predefined keys, `sub_key` may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an `OSError` exception is raised. Changed in version 3.3: See above.

`winreg.CreateKeyEx(key, sub_key, reserved=0, access=KEY_WRITE)`
Creates or opens the specified key, returning a handle object.

`key` is an already open key, or one of the predefined `HKEY_*` constants.

`sub_key` is a string that names the key this method opens or creates.

`reserved` is a reserved integer, and must be zero. The default is zero.

`access` is an integer that specifies an access mask that describes the desired security access for the key. Default is `KEY_WRITE`. See Access Rights for other allowed values.

If `key` is one of the predefined keys, `sub_key` may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an `OSError` exception is raised. New in version 3.2. Changed in version 3.3: See above.

`winreg.DeleteKey(key, sub_key)`
Deletes the specified key.

`key` is an already open key, or one of the predefined `HKEY_*` constants.

`sub_key` is a string that must be a subkey of the key identified by the `key` parameter. This value must not be `None`, and the key may not have subkeys.

*This method can not delete keys with subkeys.*

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an `OSError` exception is raised. Changed in version 3.3: See above.
The Python Library Reference, Release 3.3.3

```
winreg.DeleteKeyEx(key, sub_key, access=KEY_WOW64_64KEY, reserved=0)
```

Deletes the specified key.

**Note:** The `DeleteKeyEx()` function is implemented with the RegDeleteKeyEx Windows API function, which is specific to 64-bit versions of Windows. See the RegDeleteKeyEx documentation.

- `key` is an already open key, or one of the predefined `HKEY_* constants`.
- `sub_key` is a string that must be a subkey of the key identified by the `key` parameter. This value must not be `None`, and the key may not have subkeys.
- `reserved` is a reserved integer, and must be zero. The default is zero.
- `access` is an integer that specifies an access mask that describes the desired security access for the key. Default is `KEY_ALL_ACCESS`. See Access Rights for other allowed values.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an `OSError` exception is raised.

On unsupported Windows versions, `NotImplementedError` is raised. New in version 3.2. Changed in version 3.3: See above.

```
winreg.DeleteValue(key, value)
```

Removes a named value from a registry key.

- `key` is an already open key, or one of the predefined `HKEY_* constants`.
- `value` is a string that identifies the value to remove.

```
winreg.EnumKey(key, index)
```

Enumerates subkeys of an open registry key, returning a string.

- `key` is an already open key, or one of the predefined `HKEY_* constants`.
- `index` is an integer that identifies the index of the key to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly until an `OSError` exception is raised, indicating, no more values are available. Changed in version 3.3: See above.

```
winreg.EnumValue(key, index)
```

Enumerates values of an open registry key, returning a tuple.

- `key` is an already open key, or one of the predefined `HKEY_* constants`.
- `index` is an integer that identifies the index of the value to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly, until an `OSError` exception is raised, indicating no more values.

The result is a tuple of 3 items:

<table>
<thead>
<tr>
<th>Index</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A string that identifies the value name</td>
</tr>
<tr>
<td>1</td>
<td>An object that holds the value data, and whose type depends on the underlying registry type</td>
</tr>
<tr>
<td>2</td>
<td>An integer that identifies the type of the value data (see table in docs for <code>SetValueEx()</code>)</td>
</tr>
</tbody>
</table>

Changed in version 3.3: See above.

```
winreg.ExpandEnvironmentStrings(str)
```

Expands environment variable placeholders `%NAME%` in strings like `REG_EXPAND_SZ`:

```
>>> ExpandEnvironmentStrings('%windir%')
'C:\\Windows'
```
winreg.FlushKey(key)

Writes all the attributes of a key to the registry.

$key$ is an already open key, or one of the predefined $HKEY_*$ constants.

It is not necessary to call $FlushKey()$ to change a key. Registry changes are flushed to disk by the registry using its lazy flusher. Registry changes are also flushed to disk at system shutdown. Unlike $CloseKey()$, the $FlushKey()$ method returns only when all the data has been written to the registry. An application should only call $FlushKey()$ if it requires absolute certainty that registry changes are on disk.

**Note:** If you don’t know whether a $FlushKey()$ call is required, it probably isn’t.

winreg.LoadKey(key, sub_key, file_name)

Creates a subkey under the specified key and stores registration information from a specified file into that subkey.

$key$ is a handle returned by $ConnectRegistry()$ or one of the constants $HKEY_USERS$ or $HKEY_LOCAL_MACHINE$.

$sub_key$ is a string that identifies the subkey to load.

$file_name$ is the name of the file to load registry data from. This file must have been created with the $SaveKey()$ function. Under the file allocation table (FAT) file system, the filename may not have an extension.

A call to $LoadKey()$ fails if the calling process does not have the $SE_RESTORE_PRIVILEGE$ privilege. Note that privileges are different from permissions – see the RegLoadKey documentation for more details.

If $key$ is a handle returned by $ConnectRegistry()$, then the path specified in $file_name$ is relative to the remote computer.

winreg.OpenKey(key, sub_key, reserved=0, access=KEY_READ)

winreg.OpenKeyEx(key, sub_key, reserved=0, access=KEY_READ)

Opens the specified key, returning a handle object.

$key$ is an already open key, or one of the predefined $HKEY_*$ constants.

$sub_key$ is a string that identifies the sub_key to open.

$reserved$ is a reserved integer, and must be zero. The default is zero.

$access$ is an integer that specifies an access mask that describes the desired security access for the key. Default is $KEY_READ$. See Access Rights for other allowed values.

The result is a new handle to the specified key.

If the function fails, $OSError$ is raised. Changed in version 3.2: Allow the use of named arguments. Changed in version 3.3: See above.

winreg.QueryInfoKey(key)

Returns information about a key, as a tuple.

$key$ is an already open key, or one of the predefined $HKEY_*$ constants.

The result is a tuple of 3 items:

<table>
<thead>
<tr>
<th>Index</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>An integer giving the number of sub keys this key has.</td>
</tr>
<tr>
<td>1</td>
<td>An integer giving the number of values this key has.</td>
</tr>
<tr>
<td>2</td>
<td>An integer giving when the key was last modified (if available) as 100’s of nanoseconds since Jan 1, 1600.</td>
</tr>
</tbody>
</table>

winreg.QueryValue(key, sub_key)

Retrieves the unnamed value for a key, as a string.

$key$ is an already open key, or one of the predefined $HKEY_*$ constants.
*sub_key* is a string that holds the name of the subkey with which the value is associated. If this parameter is *None* or empty, the function retrieves the value set by the `SetValue()` method for the key identified by *key*.

Values in the registry have name, type, and data components. This method retrieves the data for a key’s first value that has a NULL name. But the underlying API call doesn’t return the type, so always use `QueryValueEx()` if possible.

```python
winreg.QueryValueEx(key, value_name)
```

Retrieves the type and data for a specified value name associated with an open registry key.

- **key** is an already open key, or one of the predefined `HKEY_* constants`.
- **value_name** is a string indicating the value to query.

The result is a tuple of 2 items:

<table>
<thead>
<tr>
<th>Index</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The value of the registry item.</td>
</tr>
<tr>
<td>1</td>
<td>An integer giving the registry type for this value (see table in docs for <code>SetValueEx()</code>)</td>
</tr>
</tbody>
</table>

```python
winreg.SaveKey(key, file_name)
```

Saves the specified key, and all its subkeys to the specified file.

- **key** is an already open key, or one of the predefined `HKEY_* constants`.
- **file_name** is the name of the file to save registry data to. This file cannot already exist. If this filename includes an extension, it cannot be used on file allocation table (FAT) file systems by the `LoadKey()` method.

If *key* represents a key on a remote computer, the path described by *file_name* is relative to the remote computer. The caller of this method must possess the `SeBackupPrivilege` security privilege. Note that privileges are different than permissions — see the Conflicts Between User Rights and Permissions documentation for more details.

This function passes NULL for `security_attributes` to the API.

```python
winreg.SetValue(key, sub_key, type, value)
```

 Associates a value with a specified key.

- **key** is an already open key, or one of the predefined `HKEY_* constants`.
- **sub_key** is a string that names the subkey with which the value is associated.
- **type** is an integer that specifies the type of the data. Currently this must be `REG_SZ`, meaning only strings are supported. Use the `SetValueEx()` function for support for other data types.
- **value** is a string that specifies the new value.

If the key specified by the *sub_key* parameter does not exist, the `SetValue` function creates it.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

The key identified by the *key* parameter must have been opened with `KEY_SET_VALUE` access.

```python
winreg.SetValueEx(key, value_name, reserved, type, value)
```

Stores data in the value field of an open registry key.

- **key** is an already open key, or one of the predefined `HKEY_* constants`.
- **value_name** is a string that names the subkey with which the value is associated.
- **reserved** can be anything – zero is always passed to the API.
- **type** is an integer that specifies the type of the data. See `Value Types` for the available types.
- **value** is a string that specifies the new value.

This method can also set additional value and type information for the specified key. The key identified by the *key* parameter must have been opened with `KEY_SET_VALUE` access.
To open the key, use the `CreateKey()` or `OpenKey()` methods.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

```python
winreg.DisableReflectionKey(key)
```
Disables registry reflection for 32-bit processes running on a 64-bit operating system.

- `key` is an already open key, or one of the predefined `HKEY_*` constants.
- Will generally raise `NotImplemented` if executed on a 32-bit operating system.
- If the key is not on the reflection list, the function succeeds but has no effect. Disabling reflection for a key does not affect reflection of any subkeys.

```python
winreg.EnableReflectionKey(key)
```
Restores registry reflection for the specified disabled key.

- `key` is an already open key, or one of the predefined `HKEY_*` constants.
- Will generally raise `NotImplemented` if executed on a 32-bit operating system.
- Restoring reflection for a key does not affect reflection of any subkeys.

```python
winreg.QueryReflectionKey(key)
```
Determines the reflection state for the specified key.

- `key` is an already open key, or one of the predefined `HKEY_*` constants.
- Returns `True` if reflection is disabled.
- Will generally raise `NotImplemented` if executed on a 32-bit operating system.

### 33.3.2 Constants

The following constants are defined for use in many `_winreg` functions.

#### HKEY_* Constants

```python
winreg.HKEY_CLASSES_ROOT
```
Registry entries subordinate to this key define types (or classes) of documents and the properties associated with those types. Shell and COM applications use the information stored under this key.

```python
winreg.HKEY_CURRENT_USER
```
Registry entries subordinate to this key define the preferences of the current user. These preferences include the settings of environment variables, data about program groups, colors, printers, network connections, and application preferences.

```python
winreg.HKEY_LOCAL_MACHINE
```
Registry entries subordinate to this key define the physical state of the computer, including data about the bus type, system memory, and installed hardware and software.

```python
winreg.HKEY_USERS
```
Registry entries subordinate to this key define the default user configuration for new users on the local computer and the user configuration for the current user.

```python
winreg.HKEY_PERFORMANCE_DATA
```
Registry entries subordinate to this key allow you to access performance data. The data is not actually stored in the registry; the registry functions cause the system to collect the data from its source.

```python
winreg.HKEY_CURRENT_CONFIG
```
Contains information about the current hardware profile of the local computer system.

```python
winreg.HKEY_DYN_DATA
```
This key is not used in versions of Windows after 98.
Access Rights

For more information, see Registry Key Security and Access.

**winreg.KEY_ALL_ACCESS**
- Combines the STANDARD_RIGHTS_REQUIRED, KEY_QUERY_VALUE, KEY_SET_VALUE, KEY_CREATE_SUB_KEY, KEY_ENUMERATE_SUB_KEYS, KEY_NOTIFY, and KEY_CREATE_LINK access rights.

**winreg.KEY_WRITE**
- Combines the STANDARD_RIGHTS_WRITE, KEY_SET_VALUE, and KEY_CREATE_SUB_KEY access rights.

**winreg.KEY_READ**
- Combines the STANDARD_RIGHTS_READ, KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS, and KEY_NOTIFY values.

**winreg.KEY_EXECUTE**
- Equivalent to KEY_READ.

**winreg.KEY_QUERY_VALUE**
- Required to query the values of a registry key.

**winreg.KEY_SET_VALUE**
- Required to create, delete, or set a registry value.

**winreg.KEY_CREATE_SUB_KEY**
- Required to create a subkey of a registry key.

**winreg.KEY_ENUMERATE_SUB_KEYS**
- Required to enumerate the subkeys of a registry key.

**winreg.KEY_NOTIFY**
- Required to request change notifications for a registry key or for subkeys of a registry key.

**winreg.KEY_CREATE_LINK**
- Reserved for system use.

64-bit Specific

For more information, see Accessing an Alternate Registry View.

**winreg.KEY_WOW64_64KEY**
- Indicates that an application on 64-bit Windows should operate on the 64-bit registry view.

**winreg.KEY_WOW64_32KEY**
- Indicates that an application on 64-bit Windows should operate on the 32-bit registry view.

Value Types

For more information, see Registry Value Types.

**winreg.REG_BINARY**
- Binary data in any form.

**winreg.REG_DWORD**
- 32-bit number.

**winreg.REG_DWORD_LITTLE_ENDIAN**
- A 32-bit number in little-endian format.

**winreg.REG_DWORD_BIG_ENDIAN**
- A 32-bit number in big-endian format.
winreg.REG_EXPAND_SZ
   Null-terminated string containing references to environment variables (%PATH%).

winreg.REG_LINK
   A Unicode symbolic link.

winreg.REG_MULTI_SZ
   A sequence of null-terminated strings, terminated by two null characters. (Python handles this termination
   automatically.)

winreg.REG_NONE
   No defined value type.

winreg.REG_RESOURCE_LIST
   A device-driver resource list.

winreg.REG_FULL_RESOURCE_DESCRIPTOR
   A hardware setting.

winreg.REG_RESOURCE_REQUIREMENTS_LIST
   A hardware resource list.

winreg.REG_SZ
   A null-terminated string.

### 33.3.3 Registry Handle Objects

This object wraps a Windows HKEY object, automatically closing it when the object is destroyed. To guarantee
cleanup, you can call either the Close() method on the object, or the CloseKey() function.

All registry functions in this module return one of these objects.

All registry functions in this module which accept a handle object also accept an integer, however, use of the
handle object is encouraged.

Handle objects provide semantics for __bool__() – thus

```python
if handle:
    print("Yes")
```

will print Yes if the handle is currently valid (has not been closed or detached).

The object also support comparison semantics, so handle objects will compare true if they both reference the same
underlying Windows handle value.

Handle objects can be converted to an integer (e.g., using the built-in int() function), in which case the under-
lying Windows handle value is returned. You can also use the Detach() method to return the integer handle,
and also disconnect the Windows handle from the handle object.

```
PyHKEY.Close()
   Closes the underlying Windows handle.

PyHKEY.Detach()
   Detaches the Windows handle from the handle object.
```

The result is an integer that holds the value of the handle before it is detached. If the handle is already
detached or closed, this will return zero.

After calling this function, the handle is effectively invalidated, but the handle is not closed. You would call
this function when you need the underlying Win32 handle to exist beyond the lifetime of the handle object.

```
PyHKEY.__enter__()
PyHKEY.__exit__(*exc_info)
```

The HKEY object implements __enter__() and __exit__() and thus supports the context protocol
for the with statement:
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
  ...
  # work with key

will automatically close key when control leaves the with block.

33.4 winsound — Sound-playing interface for Windows

**Platforms:** Windows

The winsound module provides access to the basic sound-playing machinery provided by Windows platforms. It includes functions and several constants.

winsound..Beep *(frequency, duration)*

Beep the PC’s speaker. The frequency parameter specifies frequency, in hertz, of the sound, and must be in the range 37 through 32,767. The duration parameter specifies the number of milliseconds the sound should last. If the system is not able to beep the speaker, RuntimeError is raised.

winsound.PlaySound *(sound, flags)*

Call the underlying PlaySound() function from the Platform API. The sound parameter may be a filename, audio data as a string, or None. Its interpretation depends on the value of flags, which can be a bitwise ORed combination of the constants described below. If the sound parameter is None, any currently playing waveform sound is stopped. If the system indicates an error, RuntimeError is raised.

winsound.MessageBeep *(type=MB_OK)*

Call the underlying MessageBeep() function from the Platform API. This plays a sound as specified in the registry. The type argument specifies which sound to play; possible values are -1, MB_ICONASTERISK, MB_ICONEXCLAMATION, MB_ICONHAND, MB_ICONQUESTION, and MB_OK, all described below. The value -1 produces a “simple beep”; this is the final fallback if a sound cannot be played otherwise.

winsound.SND_FILENAME

The sound parameter is the name of a WAV file. Do not use with SND_ALIAS.

winsound.SND_ALIAS

The sound parameter is a sound association name from the registry. If the registry contains no such name, play the system default sound unless SND_NODEFAULT is also specified. If no default sound is registered, raise RuntimeError. Do not use with SND_FILENAME.

All Win32 systems support at least the following; most systems support many more:

<table>
<thead>
<tr>
<th>PlaySound() name</th>
<th>Corresponding Control Panel Sound name</th>
</tr>
</thead>
<tbody>
<tr>
<td>'SystemAsterisk'</td>
<td>Asterisk</td>
</tr>
<tr>
<td>'SystemExclamation'</td>
<td>Exclamation</td>
</tr>
<tr>
<td>'SystemExit'</td>
<td>Exit Windows</td>
</tr>
<tr>
<td>'SystemHand'</td>
<td>Critical Stop</td>
</tr>
<tr>
<td>'SystemQuestion'</td>
<td>Question</td>
</tr>
</tbody>
</table>

For example:

```python
import winsound
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered (because
# "*" probably isn’t the registered name of any sound).
winsound.PlaySound("*", winsound.SND_ALIAS)
```

winsound.SND_LOOP

Play the sound repeatedly. The SND_ASYNC flag must also be used to avoid blocking. Cannot be used with SND_MEMORY.
winsound.SND_MEMORY
The `sound` parameter to `PlaySound()` is a memory image of a WAV file, as a string.

**Note:** This module does not support playing from a memory image asynchronously, so a combination of this flag and `SND_ASYNC` will raise `RuntimeError`.

winsound.SND_PURGE
Stop playing all instances of the specified sound.

**Note:** This flag is not supported on modern Windows platforms.

winsound.SND_ASYNC
Return immediately, allowing sounds to play asynchronously.

winsound.SND_NODEFAULT
If the specified sound cannot be found, do not play the system default sound.

winsound.SND_NOSTOP
Do not interrupt sounds currently playing.

winsound.SND_NOWAIT
Return immediately if the sound driver is busy.

**Note:** This flag is not supported on modern Windows platforms.

winsound.MB_ICONASTERISK
Play the SystemDefault sound.

winsound.MB_ICONEXCLAMATION
Play the SystemExclamation sound.

winsound.MB_ICONHAND
Play the SystemHand sound.

winsound.MB_ICONQUESTION
Play the SystemQuestion sound.

winsound.MB_OK
Play the SystemDefault sound.
The modules described in this chapter provide interfaces to features that are unique to the Unix operating system, or in some cases to some or many variants of it. Here’s an overview:

34.1 **posix** — The most common POSIX system calls

*Platforms:* Unix

This module provides access to operating system functionality that is standardized by the C Standard and the POSIX standard (a thinly disguised Unix interface).

**Do not import this module directly.** Instead, import the module *os*, which provides a *portable* version of this interface. On Unix, the *os* module provides a superset of the *posix* interface. On non-Unix operating systems the *posix* module is not available, but a subset is always available through the *os* interface. Once *os* is imported, there is *no* performance penalty in using it instead of *posix*. In addition, *os* provides some additional functionality, such as automatically calling *putenv()* when an entry in *os.environ* is changed.

Errors are reported as exceptions; the usual exceptions are given for type errors, while errors reported by the system calls raise *OSError*.

34.1.1 Large File Support

Several operating systems (including AIX, HP-UX, Irix and Solaris) provide support for files that are larger than 2 GiB from a C programming model where *int* and *long* are 32-bit values. This is typically accomplished by defining the relevant size and offset types as 64-bit values. Such files are sometimes referred to as *large files*.

Large file support is enabled in Python when the size of an *off_t* is larger than a *long* and the *long long* type is available and is at least as large as an *off_t*. It may be necessary to configure and compile Python with certain compiler flags to enable this mode. For example, it is enabled by default with recent versions of Irix, but with Solaris 2.6 and 2.7 you need to do something like:

```
CFLAGS='''getconf LFS_CFLAGS''' OPT="-g -O2 $CFLAGS" \
./configure
```

On large-file-capable Linux systems, this might work:

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT="-g -O2 $CFLAGS" \
./configure
```

34.1.2 Notable Module Contents

In addition to many functions described in the *os* module documentation, *posix* defines the following data item:

**posix.environ**

A dictionary representing the string environment at the time the interpreter was started. Keys and values
are bytes on Unix and str on Windows. For example, `environ[b'HOME']` (or `environ[’HOME’]` on Windows) is the pathname of your home directory, equivalent to `getenv("HOME")` in C.

Modifying this dictionary does not affect the string environment passed on by `execv()`, `popen()` or `system()`: if you need to change the environment, pass `environ` to `execve()` or add variable assignments and export statements to the command string for `system()` or `popen()`. Changed in version 3.2: On Unix, keys and values are bytes.

**Note:** The `os` module provides an alternate implementation of `environ` which updates the environment on modification. Note also that updating `os.environ` will render this dictionary obsolete. Use of the `os` module version of this is recommended over direct access to the `posix` module.

### 34.2 `pwd` — The password database

*Platforms:* Unix

This module provides access to the Unix user account and password database. It is available on all Unix versions.

Password database entries are reported as a tuple-like object, whose attributes correspond to the members of the `passwd` structure (Attribute field below, see `<pwd.h>`):

<table>
<thead>
<tr>
<th>Index</th>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>pw_name</code></td>
<td>Login name</td>
</tr>
<tr>
<td>1</td>
<td><code>pw_passwd</code></td>
<td>Optional encrypted password</td>
</tr>
<tr>
<td>2</td>
<td><code>pw_uid</code></td>
<td>Numerical user ID</td>
</tr>
<tr>
<td>3</td>
<td><code>pw_gid</code></td>
<td>Numerical group ID</td>
</tr>
<tr>
<td>4</td>
<td><code>pw_gecos</code></td>
<td>User name or comment field</td>
</tr>
<tr>
<td>5</td>
<td><code>pw_dir</code></td>
<td>User home directory</td>
</tr>
<tr>
<td>6</td>
<td><code>pw_shell</code></td>
<td>User command interpreter</td>
</tr>
</tbody>
</table>

The uid and gid items are integers, all others are strings. `KeyError` is raised if the entry asked for cannot be found.

**Note:** In traditional Unix the field `pw_passwd` usually contains a password encrypted with a DES derived algorithm (see module `crypt`). However most modern unices use a so-called shadow password system. On those unices the `pw_passwd` field only contains an asterisk (`*`) or the letter ‘x’ where the encrypted password is stored in a file `/etc/shadow` which is not world readable. Whether the `pw_passwd` field contains anything useful is system-dependent. If available, the `spwd` module should be used where access to the encrypted password is required.

It defines the following items:

- `pwd.getpwuid(uid)`
  - Return the password database entry for the given numeric user ID.

- `pwd.getpwnam(name)`
  - Return the password database entry for the given user name.

- `pwd.getpwall()`
  - Return a list of all available password database entries, in arbitrary order.

**See Also:**

- **Module `grp`** An interface to the group database, similar to this.
- **Module `spwd`** An interface to the shadow password database, similar to this.
34.3 **spwd — The shadow password database**

**Platforms:** Unix

This module provides access to the Unix shadow password database. It is available on various Unix versions.

You must have enough privileges to access the shadow password database (this usually means you have to be root).

Shadow password database entries are reported as a tuple-like object, whose attributes correspond to the members of the `spwd` structure (Attribute field below, see `<shadow.h>`):

<table>
<thead>
<tr>
<th>Index</th>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>sp_nam</code></td>
<td>Login name</td>
</tr>
<tr>
<td>1</td>
<td><code>sp_pwd</code></td>
<td>Encrypted password</td>
</tr>
<tr>
<td>2</td>
<td><code>sp_latchg</code></td>
<td>Date of last change</td>
</tr>
<tr>
<td>3</td>
<td><code>sp_min</code></td>
<td>Minimal number of days between changes</td>
</tr>
<tr>
<td>4</td>
<td><code>sp_max</code></td>
<td>Maximum number of days between changes</td>
</tr>
<tr>
<td>5</td>
<td><code>sp_warn</code></td>
<td>Number of days before password expires to warn user about it</td>
</tr>
<tr>
<td>6</td>
<td><code>sp_inact</code></td>
<td>Number of days after password expires until account is blocked</td>
</tr>
<tr>
<td>7</td>
<td><code>sp_expire</code></td>
<td>Number of days since 1970-01-01 until account is disabled</td>
</tr>
<tr>
<td>8</td>
<td><code>sp_flag</code></td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The `sp_nam` and `sp_pwd` items are strings, all others are integers. `KeyError` is raised if the entry asked for cannot be found.

The following functions are defined:

```python
def getspnam(name):
    # Return the shadow password database entry for the given user name.
```

```python
def getspall():
    # Return a list of all available shadow password database entries, in arbitrary order.
```

**See Also:**

- **Module grp**  An interface to the group database, similar to this.
- **Module pwd**  An interface to the normal password database, similar to this.

34.4 **grp — The group database**

**Platforms:** Unix

This module provides access to the Unix group database. It is available on all Unix versions.

Group database entries are reported as a tuple-like object, whose attributes correspond to the members of the `group` structure (Attribute field below, see `<pwd.h>`):

<table>
<thead>
<tr>
<th>Index</th>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>gr_name</code></td>
<td>the name of the group</td>
</tr>
<tr>
<td>1</td>
<td><code>gr_passwd</code></td>
<td>the (encrypted) group password; often empty</td>
</tr>
<tr>
<td>2</td>
<td><code>gr_gid</code></td>
<td>the numerical group ID</td>
</tr>
<tr>
<td>3</td>
<td><code>gr_mem</code></td>
<td>all the group member’s user names</td>
</tr>
</tbody>
</table>

The gid is an integer, name and password are strings, and the member list is a list of strings. (Note that most users are not explicitly listed as members of the group they are in according to the password database. Check both databases to get complete membership information. Also note that a `gr_name` that starts with a + or - is likely to be a YP/NIS reference and may not be accessible via `getgrnam()` or `getgrgid()`.)

It defines the following items:
The Python Library Reference, Release 3.3.3

```
grp.getgrgid(gid)
   Return the group database entry for the given numeric group ID. KeyError is raised if the entry asked for cannot be found.

grp.getgrnam(name)
   Return the group database entry for the given group name. KeyError is raised if the entry asked for cannot be found.

grp.getgrall()
   Return a list of all available group entries, in arbitrary order.

See Also:
Module pwd  An interface to the user database, similar to this.
Module spwd  An interface to the shadow password database, similar to this.

34.5 crypt — Function to check Unix passwords

Platforms: Unix

This module implements an interface to the crypt(3) routine, which is a one-way hash function based upon a modified DES algorithm; see the Unix man page for further details. Possible uses include storing hashed passwords so you can check passwords without storing the actual password, or attempting to crack Unix passwords with a dictionary.

Notice that the behavior of this module depends on the actual implementation of the crypt(3) routine in the running system. Therefore, any extensions available on the current implementation will also be available on this module.

34.5.1 Hashing Methods

New in version 3.3. The crypt module defines the list of hashing methods (not all methods are available on all platforms):

```
crypt.METHOD_SHA512
   A Modular Crypt Format method with 16 character salt and 86 character hash. This is the strongest method.

crypt.METHOD_SHA256
   Another Modular Crypt Format method with 16 character salt and 43 character hash.

crypt.METHOD_MD5
   Another Modular Crypt Format method with 8 character salt and 22 character hash.

crypt.METHOD_CRYPT
   The traditional method with a 2 character salt and 13 characters of hash. This is the weakest method.
```

34.5.2 Module Attributes

New in version 3.3.

```
crypt.methods
   A list of available password hashing algorithms, as crypt.METHOD_* objects. This list is sorted from strongest to weakest, and is guaranteed to have at least crypt.METHOD_CRYPT.
```

34.5.3 Module Functions

The crypt module defines the following functions:

```
import crypt

word will usually be a user’s password as typed at a prompt or in a graphical interface. The optional salt is either a string as returned from mksalt(), one of the crypt.METHOD_* values (though not all may be available on all platforms), or a full encrypted password including salt, as returned by this function. If salt is not provided, the strongest method will be used (as returned by methods()).

Checking a password is usually done by passing the plain-text password as word and the full results of a previous crypt() call, which should be the same as the results of this call.

salt (either a random 2 or 16 character string, possibly prefixed with $digit$ to indicate the method) which will be used to perturb the encryption algorithm. The characters in salt must be in the set [./a-zA-Z0-9], with the exception of Modular Crypt Format which prefixes a $digit$.

Returns the hashed password as a string, which will be composed of characters from the same alphabet as the salt.

Since a few crypt(3) extensions allow different values, with different sizes in the salt, it is recommended to use the full encrypted password as salt when checking for a password. Changed in version 3.3: Accept crypt.METHOD_* values in addition to strings for salt.

import mksalt

Return a randomly generated salt of the specified method. If no method is given, the strongest method available as returned by methods() is used.

The return value is a string either of 2 characters in length for crypt.METHOD_CRYPT, or 19 characters starting with $digit$ and 16 random characters from the set [./a-zA-Z0-9], suitable for passing as the salt argument to crypt(). New in version 3.3.

34.5.4 Examples

A simple example illustrating typical use (a constant-time comparison operation is needed to limit exposure to timing attacks. hmac.compare_digest() is suitable for this purpose):

```python
import pwd
import crypt
import getpass
from hmac import compare_digest as compare_hash

def login():
    username = input('Python login: ')
    cryptedpasswd = pwd.getpwnam(username)[1]
    if cryptedpasswd:
        if cryptedpasswd == 'x' or cryptedpasswd == '*':
            raise ValueError('no support for shadow passwords')
        cleartext = getpass.getpass()
        return compare_hash(crypt.crypt(cleartext, cryptedpasswd), cryptedpasswd)
    else:
        return True

To generate a hash of a password using the strongest available method and check it against the original:

```python
import crypt
from hmac import compare_digest as compare_hash

hashed = crypt.crypt(plaintext)
if not compare_hash(hashed, crypt.crypt(plaintext, hashed)):
    raise ValueError("hashed version doesn’t validate against original")"
34.6 termios — POSIX style tty control

Platforms: Unix

This module provides an interface to the POSIX calls for tty I/O control. For a complete description of these calls, see the POSIX or Unix manual pages. It is only available for those Unix versions that support POSIX termios style tty I/O control (and then only if configured at installation time).

All functions in this module take a file descriptor \( fd \) as their first argument. This can be an integer file descriptor, such as returned by `sys.stdin.fileno()`, or a `file object`, such as `sys.stdin` itself.

This module also defines all the constants needed to work with the functions provided here; these have the same name as their counterparts in C. Please refer to your system documentation for more information on using these terminal control interfaces.

The module defines the following functions:

- `termios.tcgetattr(fd)`
  Return a list containing the tty attributes for file descriptor \( fd \), as follows: `[iflag, oflag, cflag, lflag, ispeed, ospeed, cc]` where `cc` is a list of the tty special characters (each a string of length 1, except the items with indices VMIN and VTIME, which are integers when these fields are defined). The interpretation of the flags and the speeds as well as the indexing in the `cc` array must be done using the symbolic constants defined in the `termios` module.

- `termios.tcsetattr(fd, when, attributes)`
  Set the tty attributes for file descriptor \( fd \) from the `attributes`, which is a list like the one returned by `tcgetattr()`. The `when` argument determines when the attributes are changed: TCSANOW to change immediately, TCSADRAIN to change after transmitting all queued output, or TCSAFLUSH to change after transmitting all queued output and discarding all queued input.

- `termios.tcsendbreak(fd, duration)`
  Send a break on file descriptor \( fd \). A zero `duration` sends a break for 0.25 –0.5 seconds; a nonzero `duration` has a system dependent meaning.

- `termios.tcdrain(fd)`
  Wait until all output written to file descriptor \( fd \) has been transmitted.

- `termios.tcflush(fd, queue)`
  Discard queued data on file descriptor \( fd \). The `queue` selector specifies which queue: TCIFLUSH for the input queue, TCOFLUSH for the output queue, or TCIOFLUSH for both queues.

- `termios.tcflow(fd, action)`
  Suspend or resume input or output on file descriptor \( fd \). The `action` argument can be TCOFF to suspend output, TCOON to restart output, TCIOFF to suspend input, or TCION to restart input.

See Also:

Module `tty` Convenience functions for common terminal control operations.

34.6.1 Example

Here’s a function that prompts for a password with echoing turned off. Note the technique using a separate `tcgetattr()` call and a `try ... finally` statement to ensure that the old tty attributes are restored exactly no matter what happens:

```python
def getpass(prompt="Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        return input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
```

This function uses the `termios` module to set the echoing flag to off: the `termios.TCSADRAIN` argument to `tcsetattr()` waits until all output is transmitted, so that the echo characters are not displayed when the user types the password.
passwd = input(prompt)

finally:
    termios.tcsetattr(fd, termios.TCSADRAIN, old)
return passwd

34.7 tty — Terminal control functions

Platforms: Unix

The tty module defines functions for putting the tty into cbreak and raw modes. Because it requires the termios module, it will work only on Unix.

The tty module defines the following functions:

tty.setraw(fd, when=termios.TCSAFLUSH)
    Change the mode of the file descriptor fd to raw. If when is omitted, it defaults to termios.TCSAFLUSH, and is passed to termios.tcsetattr().

tty.setcbreak(fd, when=termios.TCSAFLUSH)
    Change the mode of file descriptor fd to cbreak. If when is omitted, it defaults to termios.TCSAFLUSH, and is passed to termios.tcsetattr().

See Also:
Module termios Low-level terminal control interface.

34.8 pty — Pseudo-terminal utilities

Platforms: Linux

The pty module defines operations for handling the pseudo-terminal concept: starting another process and being able to write to and read from its controlling terminal programmatically.

Because pseudo-terminal handling is highly platform dependent, there is code to do it only for Linux. (The Linux code is supposed to work on other platforms, but hasn’t been tested yet.)

The pty module defines the following functions:

pty.fork()
    Fork. Connect the child’s controlling terminal to a pseudo-terminal. Return value is (pid, fd). Note that the child gets pid 0, and the fd is invalid. The parent’s return value is the pid of the child, and fd is a file descriptor connected to the child’s controlling terminal (and also to the child’s standard input and output).

pty.openpty()
    Open a new pseudo-terminal pair, using os.openpty() if possible, or emulation code for generic Unix systems. Return a pair of file descriptors (master, slave), for the master and the slave end, respectively.

pty.spawn(argv, master_read, stdin_read)
    Spawn a process, and connect its controlling terminal with the current process’s standard io. This is often used to baffle programs which insist on reading from the controlling terminal.

    The functions master_read and stdin_read should be functions which read from a file descriptor. The defaults try to read 1024 bytes each time they are called.

34.8.1 Example

The following program acts like the Unix command script(1), using a pseudo-terminal to record all input and output of a terminal session in a “typescript”.

34.7. tty — Terminal control functions 1353
```
import sys, os, time, getopt
import pty

mode = 'wb'
shell = 'sh'
filename = 'typescript'
if 'SHELL' in os.environ:
    shell = os.environ['SHELL']

try:
    opts, args = getopt.getopt(sys.argv[1:], 'ap')
except getopt.error as msg:
    print('%s: %s' % (sys.argv[0], msg))
sys.exit(2)

for opt, arg in opts:
    # option -a: append to typescript file
    if opt == '-a':
        mode = 'ab'
    # option -p: use a Python shell as the terminal command
    elif opt == '-p':
        shell = sys.executable
    if args:
        filename = args[0]

script = open(filename, mode)

def read(fd):
    data = os.read(fd, 1024)
    script.write(data)
    return data

sys.stdout.write('Script started, file is %s
' % filename)
script.write(('Script started on %s' % time.asctime()).encode())
pty.spawn(shell, read)
script.write(('Script done on %s' % time.asctime()).encode())
sys.stdout.write('Script done, file is %s
' % filename)
```

34.9 fcntl — The fcntl and ioctl system calls

Platforms: Unix

This module performs file control and I/O control on file descriptors. It is an interface to the fcntl() and
ioctl() Unix routines.

All functions in this module take a file descriptor `fd` as their first argument. This can be an integer file descriptor,
such as returned by `sys.stdin.fileno()` , or a `io.IOBase` object, such as `sys.stdin` itself, which
provides a `fileno()` that returns a genuine file descriptor. Changed in version 3.3: Operations in this module
used to raise a `IOError` where they now raise a `OSError`. The module defines the following functions:

```python
fcntl.fcntl(fd, op[, arg])
```

Perform the requested operation on file descriptor `fd` (file objects providing a `fileno()` method are accepted
as well). The operation is defined by `op` and is operating system dependent. These codes are also found in the
`fcntl` module. The argument `arg` is optional, and defaults to the integer value 0. When present, it can either be an integer value, or a string. With the argument missing or an integer value, the return value of this function is the integer return value of the C `fcntl` call. When the argument is a string it represents a binary structure, e.g. created by `struct.pack()` . The binary data is copied to a buffer whose address is passed to the C `fcntl` call. The return value after a successful call is the contents of
the buffer, converted to a string object. The length of the returned string will be the same as the length of
the \texttt{arg} argument. This is limited to 1024 bytes. If the information returned in the buffer by the operating
system is larger than 1024 bytes, this is most likely to result in a segmentation violation or a more subtle
data corruption.

If the \texttt{fcntl()} fails, an \texttt{OSError} is raised.

\texttt{fcntl.ioctl(fd, op[, arg[, mutate_flag]])}

This function is identical to the \texttt{fcntl()} function, except that the argument handling is even more com-
plicated.

The \texttt{op} parameter is limited to values that can fit in 32-bits.

The parameter \texttt{arg} can be one of an integer, absent (treated identically to the integer 0), an object supporting
the read-only buffer interface (most likely a plain Python string) or an object supporting the read-write buffer
interface.

In all but the last case, behaviour is as for the \texttt{fcntl()} function.

If a mutable buffer is passed, then the behaviour is determined by the value of the \texttt{mutate_flag} parameter.

If it is false, the buffer’s mutability is ignored and behaviour is as for a read-only buffer, except that the
1024 byte limit mentioned above is avoided – so long as the buffer you pass is as least as long as what the
operating system wants to put there, things should work.

If \texttt{mutate_flag} is true (the default), then the buffer is (in effect) passed to the underlying \texttt{ioctl()} system
call, the latter’s return code is passed back to the calling Python, and the buffer’s new contents reflect the
action of the \texttt{ioctl()}. This is a slight simplification, because if the supplied buffer is less than 1024 bytes
long it is first copied into a static buffer 1024 bytes long which is then passed to \texttt{ioctl()} and copied back
into the supplied buffer.

An example:

\begin{verbatim}
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPGRP, " "))[0]
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPGRP, buf, 1)
0
>>> buf
array('h', [13341])
\end{verbatim}

\texttt{fcntl.flock(fd, op)}

Perform the lock operation \texttt{op} on file descriptor \texttt{fd} (file objects providing a \texttt{fileno()} method are accepted
as well). See the Unix manual \texttt{flock(2)} for details. (On some systems, this function is emulated using
\texttt{fcntl()}.)

\texttt{fcntl.lockf(fd, operation[, length[, start[, whence]]])}

This is essentially a wrapper around the \texttt{fcntl()} locking calls. \texttt{fd} is the file descriptor of the file to lock
or unlock, and \texttt{operation} is one of the following values:

- \texttt{LOCK_UN} – unlock
- \texttt{LOCK_SH} – acquire a shared lock
- \texttt{LOCK_EX} – acquire an exclusive lock

When \texttt{operation} is \texttt{LOCK_SH} or \texttt{LOCK_EX}, it can also be bitwise ORed with \texttt{LOCK_NB} to avoid blocking
on lock acquisition. If \texttt{LOCK_NB} is used and the lock cannot be acquired, an \texttt{OSError} will be raised and
the exception will have an \texttt{errno} attribute set to \texttt{EACCES} or \texttt{EAGAIN} (depending on the operating system;
for portability, check for both values). On at least some systems, \texttt{LOCK_EX} can only be used if the file
descriptor refers to a file opened for writing.
length is the number of bytes to lock, start is the byte offset at which the lock starts, relative to whence, and whence is as with io.IOBase.seek(), specifically:

- 0 – relative to the start of the file (os.SEEK_SET)
- 1 – relative to the current buffer position (os.SEEK_CUR)
- 2 – relative to the end of the file (os.SEEK_END)

The default for start is 0, which means to start at the beginning of the file. The default for length is 0 which means to lock to the end of the file. The default for whence is also 0.

Examples (all on a SVR4 compliant system):

```python
import struct, fcntl, os

f = open(...)  # open a file
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

Note that in the first example the return value variable `rv` will hold an integer value; in the second example it will hold a string value. The structure lay-out for the `lockdata` variable is system dependent — therefore using the `flock()` call may be better.

See Also:

Module `os` If the locking flags `O_SHLOCK` and `O_EXLOCK` are present in the `os` module (on BSD only), the `os.open()` function provides an alternative to the `lockf()` and `flock()` functions.

### 34.10 pipes — Interface to shell pipelines

**Platforms:** Unix

**Source code:** `Lib/pipes.py`

The `pipes` module defines a class to abstract the concept of a pipeline — a sequence of converters from one file to another.

Because the module uses `/bin/sh` command lines, a POSIX or compatible shell for `os.system()` and `os.popen()` is required.

The `pipes` module defines the following class:

```python
class pipes.Template
```

An abstraction of a pipeline.

Example:

```python
>>> import pipes
>>> t = pipes.Template()
>>> t.append('tr a-z A-Z', '--')
>>> f = t.open('pipefile', 'w')
>>> f.write('hello world')
>>> f.close()
>>> open('pipefile').read()
'HELLO WORLD'
```
34.10.1 Template Objects

Template objects following methods:

**Template.reset()**
- Restore a pipeline template to its initial state.

**Template.clone()**
- Return a new, equivalent, pipeline template.

**Template.debug(flag)**
- If `flag` is true, turn debugging on. Otherwise, turn debugging off. When debugging is on, commands to be executed are printed, and the shell is given `set -x` command to be more verbose.

**Template.append(cmd, kind)**
- Append a new action at the end. The `cmd` variable must be a valid bourne shell command. The `kind` variable consists of two letters.
  - The first letter can be either of ‘-’ (which means the command reads its standard input), ‘f’ (which means the commands reads a given file on the command line) or ‘.’ (which means the commands reads no input, and hence must be first.)
  - Similarly, the second letter can be either of ‘-’ (which means the command writes to standard output), ‘f’ (which means the command writes a file on the command line) or ‘.’ (which means the command does not write anything, and hence must be last.)

**Template.prepend(cmd, kind)**
- Add a new action at the beginning. See append() for explanations of the arguments.

**Template.open(file, mode)**
- Return a file-like object, open to `file`, but read from or written to by the pipeline. Note that only one of ‘r’, ‘w’ may be given.

**Template.copy(infile, outfile)**
- Copy `infile` to `outfile` through the pipe.

34.11 resource — Resource usage information

*Platforms: Unix*

This module provides basic mechanisms for measuring and controlling system resources utilized by a program. Symbolic constants are used to specify particular system resources and to request usage information about either the current process or its children.

An OSError is raised on syscall failure.

**exception resource.error**
- A deprecated alias of OSError. Changed in version 3.3: Following PEP 3151, this class was made an alias of OSError.

34.11.1 Resource Limits

Resources usage can be limited using the `setrlimit()` function described below. Each resource is controlled by a pair of limits: a soft limit and a hard limit. The soft limit is the current limit, and may be lowered or raised by a process over time. The soft limit can never exceed the hard limit. The hard limit can be lowered to any value greater than the soft limit, but not raised. (Only processes with the effective UID of the super-user can raise a hard limit.)

The specific resources that can be limited are system dependent. They are described in the `getrlimit(2)` man page. The resources listed below are supported when the underlying operating system supports them; resources which cannot be checked or controlled by the operating system are not defined in this module for those platforms.
resource.RLIMIT_INFINITY
Constant used to represent the the limit for an unlimited resource.

resource.getrlimit(resource)
Returns a tuple (soft, hard) with the current soft and hard limits of resource. Raises ValueError if an invalid resource is specified, or error if the underlying system call fails unexpectedly.

resource.setrlimit(resource, limits)
Sets new limits of consumption of resource. The limits argument must be a tuple (soft, hard) of two integers describing the new limits. A value of RLIMIT_INFINITY can be used to request a limit that is unlimited.

Raised ValueError if an invalid resource is specified, if the new soft limit exceeds the hard limit, or if a process tries to raise its hard limit. Specifying a limit of RLIMIT_INFINITY when the hard or system limit for that resource is not unlimited will result in a ValueError. A process with the effective UID of super-user can request any valid limit value, including unlimited, but ValueError will still be raised if the requested limit exceeds the system imposed limit.

setrlimit may also raise error if the underlying system call fails.

These symbols define resources whose consumption can be controlled using the setrlimit() and getrlimit() functions described below. The values of these symbols are exactly the constants used by C programs.

The Unix man page for getrlimit(2) lists the available resources. Note that not all systems use the same symbol or same value to denote the same resource. This module does not attempt to mask platform differences — symbols not defined for a platform will not be available from this module on that platform.

resource.RLIMIT_CORE
The maximum size (in bytes) of a core file that the current process can create. This may result in the creation of a partial core file if a larger core would be required to contain the entire process image.

resource.RLIMIT_CPU
The maximum amount of processor time (in seconds) that a process can use. If this limit is exceeded, a SIGXCPU signal is sent to the process. (See the signal module documentation for information about how to catch this signal and do something useful, e.g. flush open files to disk.)

resource.RLIMIT_FSIZE
The maximum size of a file which the process may create. This only affects the stack of the main thread in a multi-threaded process.

resource.RLIMIT_DATA
The maximum size (in bytes) of the process’s heap.

resource.RLIMIT_STACK
The maximum size (in bytes) of the call stack for the current process.

resource.RLIMIT_RSS
The maximum resident set size that should be made available to the process.

resource.RLIMIT_NPROC
The maximum number of processes the current process may create.

resource.RLIMIT_NOFILE
The maximum number of open file descriptors for the current process.

resource.RLIMIT_OFILE
The BSD name for RLIMIT_NOFILE.

resource.RLIMIT_MEMLOCK
The maximum address space which may be locked in memory.

resource.RLIMIT_VMEM
The largest area of mapped memory which the process may occupy.

resource.RLIMIT_AS
The maximum area (in bytes) of address space which may be taken by the process.
34.11.2 Resource Usage

These functions are used to retrieve resource usage information:

`resource.getrusage(who)`

This function returns an object that describes the resources consumed by either the current process or its children, as specified by the `who` parameter. The `who` parameter should be specified using one of the `RUSAGE_*` constants described below.

The fields of the return value each describe how a particular system resource has been used, e.g. amount of time spent running is user mode or number of times the process was swapped out of main memory. Some values are dependent on the clock tick internal, e.g. the amount of memory the process is using.

For backward compatibility, the return value is also accessible as a tuple of 16 elements.

The fields `ru_utime` and `ru_stime` of the return value are floating point values representing the amount of time spent executing in user mode and the amount of time spent executing in system mode, respectively. The remaining values are integers. Consult the `getrusage(2)` man page for detailed information about these values. A brief summary is presented here:

<table>
<thead>
<tr>
<th>Index</th>
<th>Field</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ru_utime</td>
<td>time in user mode (float)</td>
</tr>
<tr>
<td>1</td>
<td>ru_stime</td>
<td>time in system mode (float)</td>
</tr>
<tr>
<td>2</td>
<td>ru_maxrss</td>
<td>maximum resident set size</td>
</tr>
<tr>
<td>3</td>
<td>ru_ixrss</td>
<td>shared memory size</td>
</tr>
<tr>
<td>4</td>
<td>ru_idrss</td>
<td>unshared memory size</td>
</tr>
<tr>
<td>5</td>
<td>ru_isrss</td>
<td>unshared stack size</td>
</tr>
<tr>
<td>6</td>
<td>ru_minflt</td>
<td>page faults not requiring I/O</td>
</tr>
<tr>
<td>7</td>
<td>ru_majflt</td>
<td>page faults requiring I/O</td>
</tr>
<tr>
<td>8</td>
<td>ru_nswap</td>
<td>number of swap outs</td>
</tr>
<tr>
<td>9</td>
<td>ru_inblock</td>
<td>block input operations</td>
</tr>
<tr>
<td>10</td>
<td>ru_oublock</td>
<td>block output operations</td>
</tr>
<tr>
<td>11</td>
<td>ru_msgsnd</td>
<td>messages sent</td>
</tr>
<tr>
<td>12</td>
<td>ru_msgrcv</td>
<td>messages received</td>
</tr>
<tr>
<td>13</td>
<td>ru_nsignals</td>
<td>signals received</td>
</tr>
<tr>
<td>14</td>
<td>ru_nvcsw</td>
<td>voluntary context switches</td>
</tr>
<tr>
<td>15</td>
<td>ru_nivcsw</td>
<td>involuntary context switches</td>
</tr>
</tbody>
</table>

This function will raise a `ValueError` if an invalid `who` parameter is specified. It may also raise `error` exception in unusual circumstances.

`resource.getpagesize()`

Returns the number of bytes in a system page. (This need not be the same as the hardware page size.) This function is useful for determining the number of bytes of memory a process is using. The third element of the tuple returned by `getrusage()` describes memory usage in pages; multiplying by page size produces number of bytes.

The following `RUSAGE_*` symbols are passed to the `getrusage()` function to specify which processes information should be provided for.

- `resource.RUSAGE_SELF`
  Pass to `getrusage()` to request resources consumed by the calling process, which is the sum of resources used by all threads in the process.

- `resource.RUSAGE_CHILDREN`
  Pass to `getrusage()` to request resources consumed by child processes of the calling process which have been terminated and waited for.

- `resource.RUSAGE_BOTH`
  Pass to `getrusage()` to request resources consumed by both the current process and child processes. May not be available on all systems.

- `resource.RUSAGE_THREAD`
  Pass to `getrusage()` to request resources consumed by the current thread. May not be available on all systems.
systems. New in version 3.2.

## 34.12 nis — Interface to Sun’s NIS (Yellow Pages)

*Platforms:* Unix

The `nis` module gives a thin wrapper around the NIS library, useful for central administration of several hosts. Because NIS exists only on Unix systems, this module is only available for Unix.

The `nis` module defines the following functions:

### `nis.match(key, mapname, domain=default_domain)`

Return the match for `key` in map `mapname`, or raise an error (`nis.error`) if there is none. Both should be strings, `key` is 8-bit clean. Return value is an arbitrary array of bytes (may contain NULL and other joys).

Note that `mapname` is first checked if it is an alias to another name.

The `domain` argument allows to override the NIS domain used for the lookup. If unspecified, lookup is in the default NIS domain.

### `nis.cat(mapname, domain=default_domain)`

Return a dictionary mapping `key` to `value` such that `match(key, mapname)==value`. Note that both keys and values of the dictionary are arbitrary arrays of bytes.

Note that `mapname` is first checked if it is an alias to another name.

The `domain` argument allows to override the NIS domain used for the lookup. If unspecified, lookup is in the default NIS domain.

### `nis.maps(domain=default_domain)`

Return a list of all valid maps.

The `domain` argument allows to override the NIS domain used for the lookup. If unspecified, lookup is in the default NIS domain.

### `nis.get_default_domain()`

Return the system default NIS domain.

The `nis` module defines the following exception:

### `exception nis.error`

An error raised when a NIS function returns an error code.

## 34.13 syslog — Unix syslog library routines

*Platforms:* Unix

This module provides an interface to the Unix `syslog` library routines. Refer to the Unix manual pages for a detailed description of the `syslog` facility.

This module wraps the system `syslog` family of routines. A pure Python library that can speak to a syslog server is available in the `logging.handlers` module as `SysLogHandler`.

The module defines the following functions:

#### `syslog.syslog(message)`

Send the string `message` to the system logger. A trailing newline is added if necessary. Each message is tagged with a priority composed of a `facility` and a `level`. The optional `priority` argument, which defaults to `LOG_INFO`, determines the message priority. If the facility is not encoded in `priority` using logical-or (`LOG_INFO | LOG_USER`), the value given in the `openlog()` call is used.
If `openlog()` has not been called prior to the call to `syslog()`, `openlog()` will be called with no arguments.

```python
syslog.openlog([ident[, logoption[, facility]]])
```

Logging options of subsequent `syslog()` calls can be set by calling `openlog()`. `syslog()` will call `openlog()` with no arguments if the log is not currently open.

The optional `ident` keyword argument is a string which is prepended to every message, and defaults to `sys.argv[0]` with leading path components stripped. The optional `logoption` keyword argument (default is 0) is a bit field – see below for possible values to combine. The optional `facility` keyword argument (default is `LOG_USER`) sets the default facility for messages which do not have a facility explicitly encoded. Changed in version 3.2: In previous versions, keyword arguments were not allowed, and `ident` was required. The default for `ident` was dependent on the system libraries, and often was `python` instead of the name of the python program file.

```python
syslog.closelog()
```

Reset the syslog module values and call the system library `closelog()`.

This causes the module to behave as it does when initially imported. For example, `openlog()` will be called on the first `syslog()` call (if `openlog()` hasn’t already been called), and `ident` and other `openlog()` parameters are reset to defaults.

```python
syslog.setlogmask(maskpri)
```

Set the priority mask to `maskpri` and return the previous mask value. Calls to `syslog()` with a priority level not set in `maskpri` are ignored. The default is to log all priorities. The function `LOG_MASK(pri)` calculates the mask for the individual priority `pri`. The function `LOG_UPTO(pri)` calculates the mask for all priorities up to and including `pri`.

The module defines the following constants:

**Priority levels (high to low):** `LOG_EMERG`, `LOG_ALERT`, `LOG_CRIT`, `LOG_ERR`, `LOG_WARNING`, `LOG_NOTICE`, `LOG_INFO`, `LOG_DEBUG`.

**Facilities:** `LOG_KERN`, `LOG_USER`, `LOG_MAIL`, `LOG_DAEMON`, `LOG_AUTH`, `LOG_LPR`, `LOG_NEWS`, `LOG_UUCP`, `LOG_CRON`, `LOG_SYSLOG`, `LOG_LOCAL0` to `LOG_LOCAL7`, and, if defined in `<syslog.h>`, `LOG_AUTHPRIV`.

**Log options:** `LOG_PID`, `LOG_CONS`, `LOG_NDELAY`, and, if defined in `<syslog.h>`, `LOG_ODELAY`, `LOG_NOWAIT`, and `LOG_PERROR`.

### 34.13.1 Examples

#### Simple example

A simple set of examples:

```python
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

An example of setting some log options, these would include the process ID in logged messages, and write the messages to the destination facility used for mail logging:

```python
syslog.openlog(logoption=syslog.LOG_PID, facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```
UNDOCUMENTED MODULES

Here’s a quick listing of modules that are currently undocumented, but that should be documented. Feel free to contribute documentation for them! (Send via email to docs@python.org.)

The idea and original contents for this chapter were taken from a posting by Fredrik Lundh; the specific contents of this chapter have been substantially revised.

35.1 Platform specific modules

These modules are used to implement the \texttt{os.path} module, and are not documented beyond this mention. There’s little need to document these.

\texttt{ntpath} — Implementation of \texttt{os.path} on Win32, Win64, WinCE, and OS/2 platforms.

\texttt{posixpath} — Implementation of \texttt{os.path} on POSIX.
>>> The default Python prompt of the interactive shell. Often seen for code examples which can be executed interactively in the interpreter.

... The default Python prompt of the interactive shell when entering code for an indented code block or within a pair of matching left and right delimiters (parentheses, square brackets or curly braces).

2to3 A tool that tries to convert Python 2.x code to Python 3.x code by handling most of the incompatibilities which can be detected by parsing the source and traversing the parse tree.

2to3 is available in the standard library as lib2to3; a standalone entry point is provided as Tools/scripts/2to3. See 2to3 - Automated Python 2 to 3 code translation.

abstract base class Abstract base classes complement duck-typing by providing a way to define interfaces when other techniques like hasattr() would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don’t inherit from a class but are still recognized by isinstance() and issubclass(): see the abc module documentation. Python comes with many built-in ABCs for data structures (in the collections.abc module), numbers (in the numbers module), streams (in the io module), import finders and loaders (in the importlib.abc module). You can create your own ABCs with the abc module.

argument A value passed to a function (or method) when calling the function. There are two types of arguments:

• keyword argument: an argument preceded by an identifier (e.g. name=) in a function call or passed as a value in a dictionary preceded by **. For example, 3 and 5 are both keyword arguments in the following calls to complex():

    complex(real=3, imag=5)
    complex(**{'real': 3, 'imag': 5})

• positional argument: an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an iterable preceded by *. For example, 3 and 5 are both positional arguments in the following calls:

    complex(3, 5)
    complex(*[3, 5])

Arguments are assigned to the named local variables in a function body. See the calls section for the rules governing this assignment. Syntactically, any expression can be used to represent an argument; the evaluated value is assigned to the local variable.

See also the parameter glossary entry, the FAQ question on the difference between arguments and parameters, and PEP 362.

attribute A value associated with an object which is referenced by name using dotted expressions. For example, if an object o has an attribute a it would be referenced as o.a.

BDFL Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python’s creator.

bytes-like object An object that supports the bufferobjects, like bytes, bytearray or memoryview. Bytes-like objects can be used for various operations that expect binary data, such as compression, saving to a...
binary file or sending over a socket. Some operations need the binary data to be mutable, in which case not all bytes-like objects can apply.

**bytecode** Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` and `.pyo` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This “intermediate language” is said to run on a virtual machine that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

A list of bytecode instructions can be found in the documentation for the `dis` module.

**class** A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

**coercion** The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

**complex number** An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of `-1`), often written `i` in mathematics or `j` in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a `j` suffix, e.g., `3+1j`. To get access to complex equivalents of the `math` module, use `cmath`. Use of complex numbers is a fairly advanced mathematical feature. If you’re not aware of a need for them, it’s almost certain you can safely ignore them.

**context manager** An object which controls the environment seen in a `with` statement by defining `__enter__()`, `__exit__()`, and methods. See PEP 343.

**CPython** The canonical implementation of the Python programming language, as distributed on python.org. The term “CPython” is used when necessary to distinguish this implementation from others such as Jython or IronPython.

**decorator** A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. Common examples for decorators are `classmethod()` and `staticmethod()`.

The decorator syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

```python
def f(...):
    ...

f = staticmethod(f)
```

```python
@staticmethod
def f(...):
    ...
```

The same concept exists for classes, but is less commonly used there. See the documentation for `function definitions` and `class definitions` for more about decorators.

**descriptor** Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using `a.b` to get, set or delete an attribute looks up the object named `b` in the class dictionary for `a`, but if `b` is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

For more information about descriptors’ methods, see descriptors.

**dictionary** An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()`, `__eq__()` methods. Called a hash in Perl.
**docstring**  A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

**duck-typing**  A programming style which does not look at an object’s type to determine if it has the right interface; instead, the method or attribute is simply called or used (“If it looks like a duck and quacks like a duck, it must be a duck.”) By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using `type()` or `isinstance()`. (Note, however, that duck-typing can be complemented with abstract base classes.) Instead, it typically employs `hasattr()` tests or EAFP programming.

**EAFP**  Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the LBYL style common to many other languages such as C.

**expression**  A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also statements which cannot be used as expressions, such as `if`. Assignments are also statements, not expressions.

**extension module**  A module written in C or C++, using Python’s C API to interact with the core and with user code.

**file object**  An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called file-like objects or streams.

There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

**file-like object**  A synonym for `file object`.

**finder**  An object that tries to find the loader for a module. It must implement either a method named `find_loader()` or a method named `find_module()`. See PEP 302 and PEP 420 for details and `importlib.abc.Finder` for an abstract base class.

**floor division**  Mathematical division that rounds down to nearest integer. The floor division operator is `//`. For example, the expression `11 // 4` evaluates to `2` in contrast to the `2.75` returned by float true division. Note that `(-11) // 4` is `-3` because that is `-2.75` rounded downward. See PEP 238.

**function**  A series of statements which returns some value to a caller. It can also be passed zero or more arguments which may be used in the execution of the body. See also parameter, method, and the function section.

**function annotation**  An arbitrary metadata value associated with a function parameter or return value. Its syntax is explained in section function. Annotations may be accessed via the `__annotations__` special attribute of a function object.

Python itself does not assign any particular meaning to function annotations. They are intended to be interpreted by third-party libraries or tools. See PEP 3107, which describes some of their potential uses.

**__future__**  A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter.

By importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it becomes the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```
garbage collection  The process of freeing memory when it is not used anymore. Python performs garbage
collection via reference counting and a cyclic garbage collector that is able to detect and break reference
cycles.

generator  A function which returns an iterator. It looks like a normal function except that it contains yield
statements for producing a series a values usable in a for-loop or that can be retrieved one at a time with the
next() function. Each yield temporarily suspends processing, remembering the location execution
state (including local variables and pending try-statements). When the generator resumes, it picks-up where
it left-off (in contrast to functions which start fresh on every invocation).

generator expression  An expression that returns an iterator. It looks like a normal expression followed by a
for expression defining a loop variable, range, and an optional if expression. The combined expression
generates values for an enclosing function:

```python
>>> sum(i*i for i in range(10))  # sum of squares 0, 1, 4, ... 81
285
```

GIL  See global interpreter lock.

global interpreter lock  The mechanism used by the CPython interpreter to assure that only one thread executes
Python bytecode at a time. This simplifies the CPYthon implementation by making the object model (in-
cluding critical built-in types such as dict) implicitly safe against concurrent access. Locking the entire
interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism
afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when
doing computationally-intensive tasks such as compression or hashing. Also, the GIL is always released
when doing I/O.

Past efforts to create a “free-threaded” interpreter (one which locks shared data at a much finer granularity)
have not been successful because performance suffered in the common single-processor case. It is
believed that overcoming this performance issue would make the implementation much more complicated
and therefore costlier to maintain.

hashable  An object is hashable if it has a hash value which never changes during its lifetime (it needs a
__hash__() method), and can be compared to other objects (it needs an __eq__() method). Hash-
able objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use
the hash value internally.

All of Python’s immutable built-in objects are hashable, while no mutable containers (such as lists or dicion-
taries) are. Objects which are instances of user-defined classes are hashable by default; they all compare
unequal (except with themselves), and their hash value is their id().

IDLE  An Integrated Development Environment for Python. IDLE is a basic editor and interpreter environment
which ships with the standard distribution of Python.

immutable  An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object
cannot be altered. A new object has to be created if a different value has to be stored. They play an important
role in places where a constant hash value is needed, for example as a key in a dictionary.

import path  A list of locations (or path entries) that are searched by the path based finder for modules to import.
During import, this list of locations usually comes from sys.path, but for subpackages it may also come
from the parent package’s __path__ attribute.

importing  The process by which Python code in one module is made available to Python code in another module.

importer  An object that both finds and loads a module; both a finder and loader object.

interactive  Python has an interactive interpreter which means you can enter statements and expressions at the
interpreter prompt, immediately execute them and see their results. Just launch python with no arguments
(possibly by selecting it from your computer’s main menu). It is a very powerful way to test out new ideas
or inspect modules and packages (remember help(x)).

interpreted  Python is an interpreted language, as opposed to a compiled one, though the distinction can be
blurry because of the presence of the bytecode compiler. This means that source files can be run directly
without explicitly creating an executable which is then run. Interpreted languages typically have a shorter
development/debug cycle than compiled ones, though their programs generally also run more slowly. See
also interactive.

iterable  An object capable of returning its members one at a time. Examples of iterables include all sequence
types (such as list, str, and tuple) and some non-sequence types like dict, file objects, and objects
of any classes you define with an __iter__() or __getitem__() method. Iterables can be used in
a for loop and in many other places where a sequence is needed (zip(), map(), ...). When an iterable
object is passed as an argument to the built-in function iter(), it returns an iterator for the object. This
iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call
iter() or deal with iterator objects yourself. The for statement does that automatically for you, creating
a temporary unnamed variable to hold the iterator for the duration of the loop. See also iterator, sequence,
and generator.

iterator  An object representing a stream of data. Repeated calls to the iterator’s __next__() method (or
passing it to the built-in function next()) return successive items in the stream. When no more data are
available a StopIteration exception is raised instead. At this point, the iterator object is exhausted and
any further calls to its __next__() method just raise StopIteration again. Iterators are required to
have an __iter__() method that returns the iterator object itself so every iterator is also iterable and may
be used in most places where other iterables are accepted. One notable exception is code which attempts
multiple iteration passes. A container object (such as a list) produces a fresh new iterator each time you
pass it to the iter() function or use it in a for loop. Attempting this with an iterator will just return the
same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.
More information can be found in Iterator Types.

key function  A key function or collation function is a callable that returns a value used for sorting or ordering.
For example, locale.strxfrm() is used to produce a sort key that is aware of locale specific sort
conventions.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They in-
clude min(), max(), sorted(), list.sort(), heapq.nsmallest(), heapq.nlargest(), and
itertools.groupby().

There are several ways to create a key function. For example, the str.lower() method can serve as a
key function for case insensitive sorts. Alternatively, an ad-hoc key function can be built from a lambda
expression such as lambda r: (r[0], r[2]). Also, the operator module provides three key
function constructors: attrgetter(), itemgetter(), and methodcaller(). See the Sorting
HOW TO for examples of how to create and use key functions.

keyword argument  See argument.

lambda  An anonymous inline function consisting of a single expression which is evaluated when the function is
called. The syntax to create a lambda function is lambda [arguments]: expression

LBYL  Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups.
This style contrasts with the EAFP approach and is characterized by the presence of many if statements.
In a multi-threaded environment, the LBYL approach can risk introducing a race condition be-
tween “the looking” and “the leaping”. For example, the code, if key in mapping: return
mapping[key] can fail if another thread removes key from mapping after the test, but before the lookup.
This issue can be solved with locks or by using the EAFP approach.

list  A built-in Python sequence. Despite its name it is more akin to an array in other languages than to a linked
list since access to elements are O(1).

list comprehension  A compact way to process all or part of the elements in a sequence and return a list
with the results. result = [’{:04x}’ . format(x) for x in range(256) if x % 2
== 0] generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The
if clause is optional. If omitted, all elements in range(256) are processed.

loader  An object that loads a module. It must define a method named load_module(). A loader is typically
returned by a finder. See PEP 302 for details and importlib.abc.Loader for an abstract base class.
mapping  A container object that supports arbitrary key lookups and implements the methods specified in the `Mapping` or `MutableMapping` abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

meta path finder  A finder returned by a search of `sys.meta_path`. Meta path finders are related to, but different from `path entry finders`.

metaclass  The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

More information can be found in `metaclasses`.

method  A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first argument (which is usually called `self`). See `function` and `nested scope`.

method resolution order  Method Resolution Order is the order in which base classes are searched for a member during lookup. See `The Python 2.3 Method Resolution Order`.

module  An object that serves as an organizational unit of Python code. Modules have a namespace containing arbitrary Python objects. Modules are loaded into Python by the process of `importing`.

See also `package`.

MRO  See `method resolution order`.

mutable  Mutable objects can change their value but keep their `id()`. See also `immutable`.

named tuple  Any tuple-like class whose indexable elements are also accessible using named attributes (for example, `time.localtime()` returns a tuple-like object where the `year` is accessible either with an index such as `t[0]` or with a named attribute like `t.tm_year`).

A named tuple can be a built-in type such as `time.struct_time`, or it can be created with a regular class definition. A full featured named tuple can also be created with the factory function `collections.namedtuple()`. The latter approach automatically provides extra features such as a self-documenting representation like `Employee(name='jones', title='programmer')`.

namespace  The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `builtins.open` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.islice()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

namespace package  A `PEP 420` package which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a regular package because they have no `__init__.py` file.

See also `module`.

nested scope  The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes by default work only for reference and not for assignment. Local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace. The `nonlocal` allows writing to outer scopes.

new-style class  Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python’s newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()` , class methods, and static methods.

object  Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any new-style class.
package  A Python module which can contain submodules or recursively, subpackages. Technically, a package is a Python module with an\_
\_path\_
\_ attribute.

See also regular package and namespace package.

parameter  A named entity in a function (or method) definition that specifies an argument (or in some cases, arguments) that the function can accept. There are five types of parameters:

- **positional-or-keyword**: specifies an argument that can be passed either *positionally* or as a *keyword argument*. This is the default kind of parameter, for example foo and bar in the following:

  ```python
def func(foo, bar=None): ...```

- **positional-only**: specifies an argument that can be supplied only by position. Python has no syntax for defining positional-only parameters. However, some built-in functions have positional-only parameters (e.g. abs()).

- **keyword-only**: specifies an argument that can be supplied only by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare * in the parameter list of the function definition before them, for example kw\_only1 and kw\_only2 in the following:

  ```python
def func(arg, *, kw\_only1, kw\_only2): ...```

- **var-positional**: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with *, for example args in the following:

  ```python
def func(*args, **kwargs): ...```

- **var-keyword**: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with **, for example kwargs in the example above.

Parameters can specify both optional and required arguments, as well as default values for some optional arguments.

See also the argument glossary entry, the FAQ question on the difference between arguments and parameters, the inspect.Parameter class, the function section, and PEP 362.

path entry  A single location on the import path which the path based finder consults to find modules for importing.

path entry finder  A finder returned by a callable on sys.path_hooks (i.e. a path entry hook) which knows how to locate modules given a path entry.

path entry hook  A callable on the sys.path_hook list which returns a path entry finder if it knows how to find modules on a specific path entry.

path based finder  One of the default meta path finders which searches an import path for modules.

portion  A set of files in a single directory (possibly stored in a zip file) that contribute to a namespace package, as defined in PEP 420.

positional argument  See argument.

provisional package  A provisional package is one which has been deliberately excluded from the standard library’s backwards compatibility guarantees. While major changes to such packages are not expected, as long as they are marked provisional, backwards incompatible changes (up to and including removal of the package) may occur if deemed necessary by core developers. Such changes will not be made gratuitously – they will occur only if serious flaws are uncovered that were missed prior to the inclusion of the package.

This process allows the standard library to continue to evolve over time, without locking in problematic design errors for extended periods of time. See PEP 411 for more details.

Python 3000  Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated “Py3k”.

Pythonic  An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom
The Python Library Reference, Release 3.3.3

in Python is to loop over all elements of an iterable using a for statement. Many other languages don’t have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
for i in range(len(food)):
    print(food[i])
```

As opposed to the cleaner, Pythonic method:

```
for piece in food:
    print(piece)
```

qualified name A dotted name showing the “path” from a module’s global scope to a class, function or method defined in that module, as defined in

PEP 3155. For top-level functions and classes, the qualified name is the same as the object’s name:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...         
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

When used to refer to modules, the fully qualified name means the entire dotted path to the module, including any parent packages, e.g. email.mime.text:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Reference counting is generally not visible to Python code, but it is a key element of the CPython implementation. The sys module defines a getrefcount() function that programmers can call to return the reference count for a particular object.

regular package A traditional package, such as a directory containing an __init__.py file.

See also namespace package.

__slots__ A declaration inside a class that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequence An iterable which supports efficient element access using integer indices via the __getitem__() special method and defines a __len__() method that returns the length of the sequence. Some built-in sequence types are list, str, tuple, and bytes. Note that dict also supports __getitem__() and __len__(), but is considered a mapping rather than a sequence because the lookups use arbitrary immutable keys rather than integers.

slice An object usually containing a portion of a sequence. A slice is created using the subscript notation, [] with colons between numbers when several are given, such as in variable_name[1:3:5]. The bracket (subscript) notation uses slice objects internally.

special method A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with underscores. Special methods are documented in specialnames.

statement A statement is part of a suite (a “block” of code). A statement is either an expression or one of several constructs with a keyword, such as if, while or for.

struct sequence A tuple with named elements. Struct sequences expose an interface similar to named tuple in that elements can either be accessed either by index or as an attribute. However, they do not have
any of the named tuple methods like _make() or _asdict(). Examples of struct sequences include
sys.float_info and the return value of os.stat().

**triple-quoted string** A string which is bound by three instances of either a quotation mark (" ) or an apostrophe ('). While they don’t provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

**type** The type of a Python object determines what kind of object it is; every object has a type. An object’s type is accessible as its __class__ attribute or can be retrieved with type(obj).

**universal newlines** A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention ‘\n’, the Windows convention ‘\r\n’, and the old Macintosh convention ‘\r’. See PEP 278 and PEP 3116, as well as str.splitlines() for an additional use.

**view** The objects returned from dict.keys(), dict.values(), and dict.items() are called dictionary views. They are lazy sequences that will see changes in the underlying dictionary. To force the dictionary view to become a full list use list(dictview). See Dictionary view objects.

**virtual machine** A computer defined entirely in software. Python’s virtual machine executes the bytecode emitted by the bytecode compiler.

**Zen of Python** Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing ”import this” at the interactive prompt.
ABOUT THESE DOCUMENTS

These documents are generated from reStructuredText sources by Sphinx, a document processor specifically written for the Python documentation.

Development of the documentation and its toolchain takes place on the docs@python.org mailing list. We’re always looking for volunteers wanting to help with the docs, so feel free to send a mail there!

Many thanks go to:

• Fred L. Drake, Jr., the creator of the original Python documentation toolset and writer of much of the content;
• the Docutils project for creating reStructuredText and the Docutils suite;
• Fredrik Lundh for his Alternative Python Reference project from which Sphinx got many good ideas.

See reporting-bugs for information how to report bugs in this documentation, or Python itself.

B.1 Contributors to the Python Documentation

Many people have contributed to the Python language, the Python standard library, and the Python documentation. See Misc/ACKS in the Python source distribution for a partial list of contributors.

It is only with the input and contributions of the Python community that Python has such wonderful documentation – Thank You!
C.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python’s principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen Python-Labs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Note: GPL-compatible doesn’t mean that we’re distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don’t.

Thanks to the many outside volunteers who have worked under Guido’s direction to make these releases possible.

C.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 3.3.3
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 3.3.3 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 3.3.3 alone or in any derivative version, provided, however, that PSF’s License Agreement and PSF’s notice of copyright, i.e., “Copyright © 2001-2013 Python Software Foundation; All Rights Reserved” are retained in Python 3.3.3 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 3.3.3 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 3.3.3.

4. PSF is making Python 3.3.3 available to Licensee on an “AS IS” basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 3.3.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.3.3 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.3.3, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 3.3.3, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an “AS IS” basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License
Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the “BeOpen Python” logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 (“CNRI”), and the Individual or Organization (“Licensee”) accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI’s License Agreement and CNRI’s notice of copyright, i.e., “Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved” are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI’s License Agreement, Licensee may substitute the following text (omitting the quotes): “Python 1.6.1 is made available subject to the terms and conditions in CNRI’s License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013.”

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an “AS IS” basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia’s conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the “ACCEPT” button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.
Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The _random module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

C.3.2 Sockets

The socket module uses the functions, getaddrinfo(), and getnameinfo(), which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ‘‘AS IS’’ AND GAI ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE FOR GAI ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON GAI ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN GAI ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.3 Floating point exception control

The source for the fpectl module includes the following notice:

```
/                               \                           
|                               |                           
|                              The Regents of the University of California.       |
| All rights reserved.          |                           |
|                              |                           |
| Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software. |
|                              |                           |
| This work was produced at the University of California, Lawrence Livermore National Laboratory under contract no. W-7405-ENG-48 |
```

C.3. Licenses and Acknowledgements for Incorporated Software 1383
C.3.4 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

Copyright 1996 by Sam Rushing

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Sam Rushing not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.5 Cookie management

The `http.cookies` module contains the following notice:

Copyright 2000 by Timothy O’Malley <timo@alum.mit.edu>

All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Timothy O’Malley not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

Timothy O’Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL Timothy O’Malley BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.6 Execution tracing

The `trace` module contains the following notice:

portions copyright 2001, Autonomous Zones Industries, Inc., all rights... err... reserved and offered to the public under the terms of the Python 2.2 license.
Author: Zooko O’Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of neither Automatrix, Bioreason or Mojam Media be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

C.3.7 UUencode and UUdecode functions

The `uu` module contains the following notice:

Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America. All Rights Reserved
Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Lance Ellinghouse not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion between ascii and binary. This results in a 1000-fold speedup. The C version is still 5 times faster, though.
- Arguments more compliant with Python standard

### C.3.8 XML Remote Procedure Calls

The `xmlrpc.client` module contains the following notice:

```
Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and its associated documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Secret Labs AB or the author not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

### C.3.9 test_epoll

The `test_epoll` contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

C.3.10 Select kqueue

The select and contains the following notice for the kqueue interface:

Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.11 strtod and dtoa

The file Python/dtoa.c, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from http://www.netlib.org/fp/. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

/****************************************************************
* * The author of this software is David M. Gay. * */
* Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
* Permission to use, copy, modify, and distribute this software for any
* purpose without fee is hereby granted, provided that this entire notice
* is included in all copies of any software which is or includes a copy
* or modification of this software and in all copies of the supporting
* documentation for such software.
* THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
* WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
* REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
* OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
*
***************************************************************/

C.3.12 OpenSSL

The modules \texttt{hashlib}, \texttt{posix}, \texttt{ssl}, \texttt{crypt} use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows installers for Python include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here:

**LICENSE ISSUES**

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit. See below for the actual license texts. Actually both licenses are BSD-style Open Source licenses. In case of any license issues related to OpenSSL please contact openssl-core@openssl.org.

OpenSSL License

/* ====================================================================
 * Copyright (c) 1998-2008 The OpenSSL Project. All rights reserved.
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in
 * the documentation and/or other materials provided with the
 * distribution.
 * 3. All advertising materials mentioning features or use of this
 * software must display the following acknowledgment:
 * This product includes software developed by the OpenSSL Project
 * for use in the OpenSSL Toolkit. (http://www.openssl.org/)
 * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
 * endorse or promote products derived from this software without
 * prior written permission. For written permission, please contact
 * openssl-core@openssl.org.
 */
5. Products derived from this software may not be called "OpenSSL"
   nor may "OpenSSL" appear in their names without prior written
   permission of the OpenSSL Project.

6. Redistributions of any form whatsoever must retain the following
   acknowledgment:
   "This product includes software developed by the OpenSSL Project
   for use in the OpenSSL Toolkit (http://www.openssl.org/"

THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT "AS IS" AND ANY
EXRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR
ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
 STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.

Original SSLeay License
-----------------------
/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
 * All rights reserved.
 *
 * This package is an SSL implementation written
 * by Eric Young (eay@cryptsoft.com).
 * The implementation was written so as to conform with Netscapes SSL.
 *
 * This library is free for commercial and non-commercial use as long as
 * the following conditions are aheared to. The following conditions
 * apply to all code found in this distribution, be it the RC4, RSA,
 * lhash, DES, etc., code; not just the SSL code. The SSL documentation
 * included with this distribution is covered by the same copyright terms
 * except that the holder is Tim Hudson (tjh@cryptsoft.com).
 *
 * Copyright remains Eric Young’s, and as such any Copyright notices in
 * the code are not to be removed.
 * If this package is used in a product, Eric Young should be given attribution
 * as the author of the parts of the library used.
 * This can be in the form of a textual message at program startup or
 * in documentation (online or textual) provided with the package.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the copyright
 *    notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgement:
   "This product includes cryptographic software written by Eric Young (eay@cryptsoft.com)"
   The word 'cryptographic' can be left out if the routines from the library being used are not cryptographic related :-).

4. If you include any Windows specific code (or a derivative thereof) from the apps directory (application code) you must include an acknowledgement:
   "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"

   THIS SOFTWARE IS PROVIDED BY ERIC YOUNG '``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

   The licence and distribution terms for any publically available version or derivative of this code cannot be changed. i.e. this code cannot simply be copied and put under another distribution licence [including the GNU Public Licence.]

C.3.13 expat

The pyexpat extension is built using an included copy of the expat sources unless the build is configured --with-system-expat:

Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
C.3.14 libffi

The \_ctypes extension is built using an included copy of the libffi sources unless the build is configured
--with-system-libffi:

Copyright (c) 1996-2008 Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
`Software`), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED `AS IS`, WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.

C.3.15 zlib

The \zlib extension is built using an included copy of the zlib sources if the zlib version found on the system is
too old to be used for the build:

Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided `as-is`, without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly  Mark Adler
jloup@gzip.org     madler@alumni.caltech.edu
COPYRIGHT

Python and this documentation is:
Copyright © 2001-2013 Python Software Foundation. All rights reserved.
Copyright © 2000 BeOpen.com. All rights reserved.
Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.
Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

See *History and License* for complete license and permissions information.
PYTHON MODULE INDEX

__future__, 1251
__main__, 1231
_dummy_thread, 670
_thread, 669

a
abc, 1243
aifc, 968
argparse, 438
array, 187
ast, 1301
asyncio, 708
asyncio, 704
atexit, 1247
audioop, 965

c
base64, 787
bdb, 1189
binascii, 789
binhex, 789
bisect, 185
builtins, 1230
bz2, 334

calendar, 162
cgi, 847
cgit, 853
chunk, 974
cmath, 210
cmd, 1031
code, 1271
codes, 121
codeop, 1273
collections, 165
collections.abc, 179
colorsys, 975
compileall, 1315
concurrent.futures, 643
configparser, 360
contextlib, 1236
copy, 195
copyreg, 304
cProfile, 1202
crypt (Unix), 1350
csv, 355
ctypes, 553
curses (Unix), 525
curses.ascii, 541
curses.panel, 543
curses.textpad, 540
d
datetime, 137
dbm, 308
dbm.dumb, 311
dbm.gnu (Unix), 309
dbm.ndbm (Unix), 310
decimal, 212
difflib, 98
dis, 1317
distutils, 1269
doctest, 1078
dummy_threading, 668
e
e-mail, 719
e-mail.charset, 745
e-mail.encoders, 747
e-mail.errors, 748
e-mail.generator, 728
e-mail.header, 743
e-mail.headerregistry, 736
e-mail.iterators, 751
e-mail.message, 719
e-mail.mime, 741
e-mail.parser, 725
e-mail.policy, 731
e-mail.utils, 749
encodings.idna, 136
encodings.mbc, 136
encodings.utf_8_sig, 136
errno, 548
f
faulthandler, 1193
fcntl (Unix), 1354
flecmp, 278
fileinput, 272

---

1395
The Python Library Reference, Release 3.3.3

resource (Unix), 1357
rlcompleter, 116
runpy, 1284

S
sched, 660
select, 663
shelve, 305
shlex, 1036
shutil, 285
signal, 710
site, 1265
smtpd, 914
smtpplib, 909
sndhdr, 977
socket, 673
socketserver, 921
spwd (Unix), 1349
sqlite3, 311
ssl, 688
stat, 274
string, 73
stringprep, 112
struct, 117
subprocess, 647
sunau, 970
symbol, 1307
syntable, 1305
sys, 1215
sysconfig, 1227
syslog (Unix), 1360

tabnanny, 1313
tarfile, 346
telnetlib, 916
tempfile, 280
termios (Unix), 1352
test, 1173
test.support, 1175
textwrap, 107
threading, 583
time, 431
timeit, 1207
tkinter, 1041
tkinter.scrolledtext (Tk), 1071
tkinter.tix, 1066
tkinter.ttk, 1051
token, 1308
tokenize, 1309
trace, 1211
traceback, 1248
tty (Unix), 1353
turtle, 999
types, 193

U
unicodedata, 110
unittest, 1099
unittest.mock, 1121
urllib, 862
urllib.error, 884
urllib.parse, 878
urllib.request, 862
urllib.response, 878
urllib.robotparser, 884
uu, 792
uuid, 918

V
venv, 1180

W
warnings, 1231
wave, 972
weakref, 190
webbrowser, 845
winreg (Windows), 1337
winsound (Windows), 1345
wsgiref, 854
wsgiref.handlers, 859
wsgiref.headers, 856
wsgiref.simple_server, 857
wsgiref.util, 854
wsgiref.validate, 858

X
xdrlib, 377
xml, 798
xml.dom, 810
xml.dom.minidom, 820
xml.dom.pulldom, 842
xml.etree.ElementTree, 799
xml.parsers.expat, 836
xml.parsers.expat.errors, 843
xml.parsers.expat.model, 842
xml.sax, 826
xml.sax.handler, 827
xml.sax.saxutils, 832
xml.sax.xmlreader, 833
xmlrpc.client, 943
xmlrpc.server, 950

Z
zipfile, 340
zipimport, 1279
zlib, 329
Symbols

* operator, 27
** operator, 27
+ operator, 27
- operator, 27

--help
trace command line option, 1211
-ignore-dir=<dir>
trace command line option, 1212
-ignore-modules=<mod>
trace command line option, 1212
--user-base
site command line option, 1267
--user-site
site command line option, 1267
--version
trace command line option, 1211
-C, --coverdir=<dir>
trace command line option, 1212
-R, --no-report
trace command line option, 1212
-T, --trackcalls
trace command line option, 1212
-a, --annotate
pickletools command line option, 1325
-b
compileall command line option, 1316
-b, --buffer
unittest command line option, 1102
-c, --catch
unittest command line option, 1102
-c, --clock
timeit command line option, 1209
-c, --count
trace command line option, 1211
-d destdir
compileall command line option, 1315
-e, --exact
tokenize command line option, 1311
-f
compileall command line option, 1315

-f, --failfast
unittest command line option, 1102
-f, --file=<file>
trace command line option, 1212
-g, --timing
trace command line option, 1212
-h, --help
timeit command line option, 1209
tokenize command line option, 1311
-i list
compileall command line option, 1315
-l
compileall command line option, 1315
-l, --indentlevel=<num>
pickletools command line option, 1325
-l, --listfuncs
trace command line option, 1211
-m, --memo
pickletools command line option, 1325
-m, --missing
trace command line option, 1212
-n N, --number=N
timeit command line option, 1209
-o, --output=<file>
pickletools command line option, 1325
-p, --pattern pattern
unittest-discover command line option, 1102
-p, --preamble=<preamble>
pickletools command line option, 1325
-p, --process
timeit command line option, 1209
-q
compileall command line option, 1315
-r N, --repeat=N
timeit command line option, 1209
-r, --report
trace command line option, 1211
-s S, --setup=S
timeit command line option, 1209
-s, --start-directory directory
unittest-discover command line option, 1102
-s, --summary
trace command line option, 1212
-t, --time
timeit command line option, 1209
-t, --top-level-directory directory
unittest-discover command line option, 1102
-t, --trace
  trace command line option, 1211
-v, --verbose
  trace command line option, 1209
unittest-discover command line option, 1102
-x regex
  compileall command line option, 1315
... , 1365
.ini
  file, 360
.pdbrc
  file, 1197
/ operator, 27
// operator, 27
== operator, 26
% operator, 27
% formatting, 43
% interpolation, 43
& operator, 28
_CData (class in ctypes), 577
.FuncPtr (class in ctypes), 572
.SimpleCData (class in ctypes), 578
.abs() (in module operator), 262
.add() (in module operator), 262
.and() (in module operator), 262
.bases (class attribute), 63
.call() (email.headerregistry.HeaderRegistry method), 740
.ceil() (fractions.Fraction method), 239
class (instance attribute), 63
class (unittest.mock.Mock attribute), 1129
code (function object attribute), 61
concat() (in module operator), 263
contains() (email.message.Message method), 721
contains() (in module operator), 263
contains() (mailbox.Mailbox method), 771
copy() (copy protocol), 196
decode (base64 module), 209
decode () (in module, 209
decopy() (copy protocol), 196
delitem() (email.message.Message method), 722
delitem() (in module operator), 263
delitem() (mailbox.MH method), 774
delitem() (mailbox.Mailbox method), 769
dict (object attribute), 63
dir() (unittest.mock.Mock method), 1126
displayhook (in module sys), 1217
der() (contextmanager method), 60
der() (winreg.PyHKEY method), 1344
eq() (email.charset.Charset method), 747
eq() (email.header.Header method), 745
eq() (in module operator), 261
eq() (instance method), 26
_eq_() (memoryview method), 49
_excepthook (in module sys), 1217
_exit_() (contextmanager method), 60
_exit_() (winreg.PyHKEY method), 1344
_floor_() (fractions.Fraction method), 239
_floordiv_() (in module operator), 262
_format_ (1) (datetime.date method), 143
_format_ (datetime.date method), 149
_format_ (datetime.time method), 153
_future_ (module), 1367
_future_ (module), 1251
ge() (in module operator), 261
g() (instance method), 26
_getitem_() (email.headerregistry.HeaderRegistry method), 740
_getitem_() (email.message.Message method), 722
_getitem_() (in module operator), 263
_getitem_() (mailbox.Mailbox method), 770
_getnewargs_() (object method), 298
_getstate_() (copy protocol), 301
_getstate_() (object method), 298
gt() (in module operator), 261
gt() (instance method), 26
_iadd_() (in module operator), 266
_iand_() (in module operator), 266
_concat_() (in module operator), 266
_floordiv_() (in module operator), 264
_ilshift_() (in module operator), 267
_imod_() (in module operator), 266
_import_() (built-in function), 21
_import_() (in module importlib), 1286
_imul_() (in module operator), 266
_index_() (in module operator), 262
_init_() (difflib.HtmlDiff method), 99
_init_() (logging.Handler method), 495
_inv_() (in module operator), 262
_invert_() (in module operator), 262
_ior_() (in module operator), 266
_ipow_() (in module operator), 267
_lshift_() (in module operator), 267
_isub_() (in module operator), 267
_iter_() (container method), 32
_iter_() (iterator method), 32
_iter_() (mailbox.Mailbox method), 770
_iter_() (unittest.TestSuite method), 1114
_itruediv_() (in module operator), 267
_ixor_() (in module operator), 267
_le_() (in module operator), 261
_le_() (instance method), 26
_len_() (email.message.Message method), 721
_len_() (mailbox.Mailbox method), 771
_lshift_() (in module operator), 262
_lt_() (in module operator), 261
_lt_() (instance method), 26
_main_ (module), 1231
_missing_() (collections.defaultdict method), 173
_mod_() (in module operator), 262

The Python Library Reference, Release 3.3.3

__mro__ (class attribute), 63
__mul__ () (in module operator), 262
__name__ (class attribute), 63
__ne__() (email.charset.Charset method), 747
__ne__() (email.header.Header method), 745
__ne__() (in module operator), 261
__ne__() (instance method), 26
__neg__ () (in module operator), 262
__next__() (csv.csvreader method), 359
__next__() (iterator method), 32
__not__ () (in module operator), 262
__or__() (in module operator), 262
__pos__() (in module operator), 263
__pow__() (in module operator), 263
__qualname__ (class attribute), 63
__reduce__ () (object method), 298
__reduce_ex__() (object method), 299
__repr__ () (multiprocessing.managers.BaseProxy method), 613
__repr__() (netrc.netrc method), 377
__round__() (fractions.Fraction method), 239
__rshift__() (in module operator), 263
__setitem__() (email.message.Message method), 722
__setitem__() (in module operator), 263
__setitem__() (mailbox.Mailbox method), 769
__setitem__() (mailbox.Maildir method), 772
__setstate__() (copy protocol), 301
__setstate__() (object method), 298
__slots__, 1372
__stderr__ (in module sys), 1226
__stdin__ (in module sys), 1226
__stdout__ (in module sys), 1226
__str__() (datetime.date method), 143
__str__() (datetime.datetime method), 149
__str__() (datetime.time method), 152
__str__() (email.charset.Charset method), 747
__str__() (email.header.Header method), 745
__str__() (email.headerregistry.Address method), 741
__str__() (email.headerregistry.Group method), 741
__str__() (email.message.Message method), 720
__str__() (multiprocessing.managers.BaseProxy method), 613
__sub__() (in module operator), 263
__subclasses__() (class method), 63
__subclasses_hook__() (abc.ABCMeta method), 1243
__truediv__() (in module operator), 263
__xor__() (in module operator), 263
__anonymous__ (ctypes.Structure attribute), 581
__dict__ () (collections.somenamedtuple method), 175
__base__ (ctypes._CData attribute), 577
__callmethod__ (multiprocessing.managers.BaseProxy method), 613
__parse__ (gettext.NullTranslations method), 985
__replace__ (collections.somenamedtuple method), 175
__setroot__() (xml.etree.ElementTree.ElementTree method), 807
__source__ (collections.somenamedtuple attribute), 176
__structure__ (module email.iterators), 752
__thread__ (module), 669
__write__() (wsgiref.handlers.BaseHandler method), 860
__xoptions__ (module sys), 1227
^ operator, 28
> operator, 26
>= operator, 26
>> operator, 28
>>>, 1365
< operator, 26
<= operator, 26
<< operator, 28
<protocol>.__proxy__, 865
2to3, 1365

A
A (in module re), 87
A-LAW, 969, 977
a-LAW, 965
a2b_base64() (module binascii), 790
a2b_hex() (module binascii), 791
a2b_hqx() (module binascii), 790
a2b_qp() (module binascii), 790
a2b_uu() (module binascii), 789
abc (module), 1243
ABCMeta (class in abc), 1243
abiflags (module sys), 1215
abort() (ftplib.FTP method), 893
abort() (module os), 411

Index 1401
The Python Library Reference, Release 3.3.3

abort() (threading.Barrier method), 593
above() (curses.panel.Panel method), 544
abs() (built-in function), 5
abs() (decimal.Context method), 225
abs() (in module operator), 262
abspath() (in module os.path), 269
abstract base class, 1365
AbstractBasicAuthHandler (class in urllib.request), 865
abstractclassmethod() (in module abc), 1245
AbstractDigestAuthHandler (class in urllib.request), 866
AbstractFormatter (class in formatter), 1329
abstractmethod() (in module abc), 1244
abstractproperty() (in module abc), 1246
abstractstaticmethod() (in module abc), 1246
AbstractWriter (class in formatter), 1330
accept() (asyncore.dispatcher method), 706
accept() (multiprocessing.connection.Listener method), 617
accept() (socket.socket method), 680
access() (in module os), 399
accumulate() (in module itertools), 246
acos() (in module cmath), 211
acos() (in module math), 208
acosh() (in module cmath), 211
acosh() (in module math), 209
acquire() (_thread.lock method), 669
acquire() (logging.Handler method), 495
acquire() (threading.Condition method), 589
acquire() (threading.Lock method), 587
acquire() (threading.RLock method), 587
acquire() (threading.Semaphore method), 590
acquire_lock() (in module imp), 1277
ACTION (optparse.Option attribute), 488
active_children() (in module multiprocessing), 602
active_count() (in module threading), 583
add() (decimal.Context method), 225
add() (in module audiioop), 965
add() (in module operator), 262
add() (mailbox.Mailbox method), 769
add() (mailbox.Maildir method), 772
add() (msilib.RadioButtonGroup method), 1335
add() (pstats.Stats method), 1204
add() (set method), 56
add() (tarfile.TarFile method), 349
add() (tkinter.ttk.Notebook method), 1056
add_alias() (in module email.charset), 747
add_argument() (argparse.ArgumentParser method), 447
add_argument_group() (argparse.ArgumentParser method), 462
add_cgi_var() (wsgiref.handlers.BaseHandler method), 860
add_charset() (in module email.charset), 747
add_cookie_header() (http.cookiejar.CookieJar method), 937
add_data() (in module msilib), 1331
add_data() (urllib.request.Request method), 867
add_done_callback() (concurrent.futures.Future method), 646
add_fallback() (gettext.NullTranslations method), 985
add_file() (msilib.Directory method), 1334
addフラグ() (mailbox.MaildirMessage method), 777
addフラグ() (mailbox.mboxMessage method), 779
addフラグ() (mailbox.MMDFMessage method), 782
add_flowing_data() (formatter.formatter method), 1328
add_folder() (mailbox.Maildir method), 772
add_folder() (mailbox.MH method), 774
add_handler() (urllib.request.OpenerDirector method), 868
add_header() (email.message.Message method), 722
add_header() (urllib.request.Request method), 867
add_header() (wsgiref.headers.Headers method), 856
add_history() (in module readline), 115
add_ho_rule() (formatter.formatter method), 1327
add_label() (mailbox.BabylMessage method), 781
add_label() (formatter.formatter method), 1328
add_line_break() (formatter.formatter method), 1327
add_mutable_group() (argparse.ArgumentParser method), 462
add_option() (optparse.OptionParser method), 476
add_paren() (urllib.request.BaseHandler method), 869
add_password() (urllib.request.HTTPPasswordMgr method), 871
add_section() (configparser.ConfigParser method), 372
add_section() (configparser.RawConfigParser method), 375
add_sequence() (mailbox.MHMessage method), 780
add_stream() (in module msilib), 1332
add_subparsers() (argparse.ArgumentParser method), 458
add_table() (in module msilib), 1332
add_type() (in module mimetypes), 785
add_unredirected_header() (urllib.request.Request method), 867
addch() (curses.window method), 531
addCleanup() (unittest.TestCase method), 1113
addComponent() (turtle.Shape method), 1026
addError() (unittest.TestResult method), 1117
addExpectedFailure() (unittest.TestResult method), 1117
addFailure() (unittest.TestCase method), 1117
addFile() (tarfile.TarFile method), 349
addFilter() (logging.Handler method), 495
addFilter() (logging.Logger method), 494
addHandler() (logging.Logger method), 494
addLevelName() (in module logging), 503
addstr() (curses.window method), 531
AddPackagePath() (in module modulefinder), 1283
addr() (smtpd.SMTPChannel attribute), 915
addr_spec() (email.headerregistry.Address attribute), 741
Address (class in email.headerregistry), 740

1402 Index
address (email.headerregistry.SingleAddressHeader attribute), 739
address (multiprocessing.connection.Listener attribute), 617
address (multiprocessing.managers.BaseManager attribute), 609
address_exclude() (ipaddress.IpV4Network method), 959
address_exclude() (ipaddress.IpV6Network method), 961
address_family (socketserver.BaseServer attribute), 923
address_string() (http.server.BaseHTTPRequestHandler method), 930
addresses (email.headerregistry.AddressHeader attribute), 738
addresses (email.headerregistry.Group attribute), 741
AddressHeader (class in email.headerregistry), 738
address() (exception module types), 575
AddressValueError, 964
addshape() (module turtle), 1024
addsite_dir() (module site), 1267
addSkip() (unittest.TestResult method), 1117
addstr() (curses.window method), 531
addSuccess() (unittest.TestResult method), 1117
addTest() (unittest.TestSuite method), 1114
addTests() (unittest.TestSuite method), 1114
addTypeEqualityFunc() (unittest.TestCase method), 1111
addUnexpectedSuccess() (unittest.TestResult method), 1117
adjusted() (decimal.Decimal method), 218
adler32() (module zlib), 329
ادي (module audioop), 965
ADPCM, Intel/DVI, 965
ADPCM2LIN() (module audioop), 965
AES
algorithm, 386
AF_CAN (module socket), 676
AF_INET (module socket), 675
AF_INET6 (module socket), 675
AF_RDS (module socket), 676
AF_UNIX (module socket), 675
aifc (module), 968
aifc() (aifc.aifc method), 969
AIFF, 968, 974
aiff() (aifc.aifc method), 969
AIF-C, 968, 974
alarm() (module signal), 712
alaw2lin() (module audioop), 965
algorithm
AES, 386
algorithms_available (module hashlib), 384
algorithms_guaranteed (module hashlib), 384
alias (module pdb), 1199
alignment() (module types), 575
all() (module function), 5
all_errors (module ftplib), 892
all_features (module xml.sax.handler), 829
all_properties (module xml.sax.handler), 829
all_suffixes() (module importlib.machinery), 1292
allocate_lock() (module _thread), 669
allow_reuse_address (socketserver.BaseServer attribute), 923
allowed_domains() (http.cookiejar.DefaultCookiePolicy method), 940
alt() (module curses.ascii), 543
ALT_DIGITS (module locale), 994
altsep (module os), 420
altszone (module time), 432
ALWAYS_TYPED_ACTIONS (optparse.Option attribute), 488
AMPER (module token), 1308
AMPEREQUAL (module token), 1308
and
operator, 25
and_() (module operator), 262
annotation (inspect.Parameter attribute), 1260
answer_challenge() (module multiprocessing.connection), 616
anticipate_failure() (module test.support), 1178
ANY (module unittest.mock), 1146
any() (module function), 5
api_version (module sys), 1226
apop() (poplib.POP3 method), 896
append() (collections.deque method), 170
append() (collections.array method), 188
append() (collections.deque method), 170
append() (email.header.Header method), 744
append() (imaplib.IMAP4 method), 899
append() (msilib.CAB method), 1334
append() (pipes.Template method), 1357
append() (sequence method), 34
append() (xml.etree.ElementTree.Element method), 806
appendChild() (xml.dom.Node method), 814
appendleft() (collections.deque method), 170
application_uri() (module wsgiref.util), 854
apply (2to3 fixer), 1170
apply() (multiprocessing.pool.Pool method), 614
apply_async() (multiprocessing.pool.Pool method), 614
architecture() (module platform), 545
archive (zipimport.zipimporter attribute), 1280
aRepr (module reprlib), 200
argparse (module), 438
args (BaseException attribute), 65
args (functools.partial attribute), 261
args (inspect.BoundArguments attribute), 1261
args (module pdb command), 1199
argtypes (ctypes._FuncPtr attribute), 572
argument, 1365
ArgumentDefaultsHelpFormatter (class in argparse), 443
ArgumentError, 572
ArgumentParser (module argparse), 440
arguments (inspect.BoundArguments attribute), 1261
argv (module sys), 1215
arithmetic, 27

Index 1403
ArithmeticError, 66
array
    module, 45
array (class in array), 187
array (module), 187
Array() (in module multiprocessing), 606
Array() (in module multiprocessing.sharedctypes), 607
Array() (multiprocessing.managers.SyncManager method), 610
arrays, 187
article() (ntplib.NNTP method), 907
as_completed() (in module concurrent.futures), 647
as_integer_ratio() (float method), 29
AS_IS (in module formatter), 1327
as_string() (email.message.Message method), 720
as_tuple() (decimal.Decimal method), 218
ASCII (in module re), 87
ascii() (built-in function), 5
ascii() (in module curses.ascii), 543
ascii_lowercase (in module string), 73
ascii_uppercase (in module string), 73
asctime() (in module time), 432
asin() (in module cmath), 211
asin() (in module math), 208
asinh() (in module cmath), 211
asinh() (in module math), 209
assert
    statement, 66
assert_any_call() (unittest.mock.Mock method), 1124
assert_called_once_with() (unittest.mock.Mock method), 1124
assert_called_with() (unittest.mock.Mock method), 1124
assert_data() (unittest.mock.Mock method), 1125
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertEqual() (unittest.TestCase method), 1108
assertNotIn() (unittest.TestCase method), 1108
assertNotIsInstance() (unittest.TestCase method), 1108
assertNotRegexp() (unittest.TestCase method), 1111
assertRaises() (unittest.TestCase method), 1111
assertRaisesRegexp() (unittest.TestCase method), 1110
assertRegexp() (unittest.TestCase method), 1111
assertSequenceEqual() (unittest.TestCase method), 1111
assertSetEqual() (unittest.TestCase method), 1112
assertTrue() (unittest.TestCase method), 1108
assertTupleEqual() (unittest.TestCase method), 1111
assertWarns() (unittest.TestCase method), 1109
assertWarnsRegexp() (unittest.TestCase method), 1110
assignment
    slice, 34
    subscript, 34
AST (class in ast), 1301
ast (module), 1301
astimezone() (datetime.datetime method), 147
async_chat (class in asynchat), 708
async_chat.ac_in_buffer_size (in module asynchat), 708
async_chat.ac_out_buffer_size (in module asynchat), 708
async_chat (module), 708
asynccore (module), 704
AsyncResult (class in multiprocessing.pool), 615
AT (in module token), 1308
At() (in module tkinter), 1050
atan() (in module cmath), 211
atan() (in module math), 208
atan2() (in module math), 208
atan2() (in module math), 209
AtEnd() (in module tkinter), 1050
atexit (module), 1247
AtInsert() (in module tkinter), 1050
AtSelFirst() (in module tkinter), 1050
AtSelLast() (in module tkinter), 1050
attach() (email.message.Message method), 720
attach() (email.message.Message method), 720
attach_mock() (unittest.mock.Mock method), 1125
AttlistDeclHandler() (xml.parsers.expat.xmlparser method), 839
attrgetter() (in module operator), 263
attrib (xml.etree.ElementTree.Element attribute), 806
attribute, 1365
AttributeError, 66
attributes (xml.dom.Node attribute), 813
AttributesImpl (class in xml.sax.xmlreader), 833
AttributesNSImpl (class in xml.sax.xmlreader), 833
attroff() (curses.window method), 531
attron() (curses.window method), 531
attrs() (curses.window method), 531
Audio Interchange File Format, 968, 974
AUDIO_FILE_ENCODING_ADPCM_G721 (in module sunau), 971
The Python Library Reference, Release 3.3.3

BINARY_OR (opcode), 1320
BINARY_POWER (opcode), 1319
BINARY_RSHIFT (opcode), 1319
BINARY_SUBSCR (opcode), 1319
BINARY_SUBTRACT (opcode), 1319
BINARY_TRUE_DIVIDE (opcode), 1319
BINARY_XOR (opcode), 1320
binascii (module), 789
bind (widgets), 1049
bind() (asyncore.dispatcher method), 706
bind() (inspect.Signature method), 1259
bind() (socket.socket method), 680
bind_partial() (inspect.Signature method), 1259
bind_port() (in module test.support), 1179
bind_textdomain_codeset() (in module gettext), 983
bindtextdomain() (in module gettext), 983
binhex module, 789
binhex (module), 789
binhex() (in module binhex), 789
bisect (module), 185
bisect() (in module bisect), 185
bisect_left() (in module bisect), 185
bisect_right() (in module bisect), 185
bit_length() (int method), 28
bitmap() (msilib.Dialog method), 1335
bitwise operations, 28
bk() (in module turtle), 1004
bgd() (curses.window method), 531
bkgdset() (curses.window method), 532
blocked_domains() (http.cookiejar.DefaultCookiePolicy method), 940
BlockingIOError, 69, 422
body() (nntplib.NNTP method), 908
body_encoding (email.charset.Charset attribute), 747
body_line_iterator() (in module email.iterators), 751
BOM (in module codecs), 124
BOM_BE (in module codecs), 124
BOM_UTF16 (in module codecs), 124
BOM_UTF16_BE (in module codecs), 124
BOM_UTF16_LE (in module codecs), 124
BOM_UTF32 (in module codecs), 124
BOM_UTF32_BE (in module codecs), 124
BOM_UTF32_LE (in module codecs), 124
BOM_UTF8 (in module codecs), 124
bool() (built-in function), 6
Boolean
object, 26
operations, 25
type, 6
values, 62
BOOLEAN_STATES (in module configparser), 369
border() (curses.window method), 532
bottom() (curses.panel.Panel method), 544
bottom_panel() (in module curses.panel), 543
BoundArguments (class in inspect), 1261
BoundaryError, 748
BoundedSemaphore (class in multiprocessing), 605
BoundedSemaphore (class in threading), 590
BoundedSemaphore() (multiprocessing.managers.SyncManager method), 609
box() (curses.window method), 532
bpformat() (bdb.Breakpoint method), 1189
bpprint() (bdb.Breakpoint method), 1190
break (pdb command), 1197
break_anywhere() (bdb.Bdb method), 1191
break_here() (bdb.Bdb method), 1191
break_long_words (textwrap.TextWrapper attribute), 110
BREAK_LOOP (opcode), 1320
break_on_hyphens (textwrap.TextWrapper attribute), 110
Breakpoint (class in bdb), 1189
breakpoints, 1073
broadcast_address (ipaddress.IPv4Network attribute), 959
broadcast_address (ipaddress.IPv6Network attribute), 961
broken (threading.Barrier attribute), 593
BrokenBarrierError, 593
BrokenPipeError, 69
BrokenProcessPool, 647
BROWSER, 845, 846
BsdDbShelf (class in shelve), 306
buffer (2to3 fixer), 1170
buffer (io.TextIOBase attribute), 429
buffer (unittest.TestResult attribute), 1116
buffer protocol
binary sequence types, 45
str (built-in class), 38
buffer size, I/O, 16
buffer_info() (array.array method), 188
buffer_size (xml.parsers.expat.xmlparser attribute), 838
buffer_text (xml.parsers.expat.xmlparser attribute), 838
buffer_used (xml.parsers.expat.xmlparser attribute), 838
BufferedIOBase (class in io), 425
BufferedReader (class in io), 428
BufferedReader (class in io), 427
BufferedRWPair (class in io), 428
BufferedWriter (class in io), 428
BufferError, 66
BufferingHandler (class in logging.handlers), 522
BufferTooShort, 600
bufsize() (ossaudiodev.oss_audio_device method), 980
BUILD_LIST (opcode), 1322
BUILD_MAP (opcode), 1322
build_opener() (in module urllib.request), 864
BUILD_SET (opcode), 1322
BUILD_SLICE (opcode), 1324
BUILD_TUPLE (opcode), 1322
built-in
types, 25
built-in function
  compile, 61, 194, 1299
  complex, 27
  eval, 62, 198, 1299
  exec, 62, 1299
  float, 27
  hash, 34
  int, 27
  len, 32, 56
  max, 32
  min, 32
  slice, 1324
  type, 62

builtin_module_names (in module sys), 1215
BuiltinFunctionType (in module types), 194
BuiltinImporter (class in importlib.machinery), 1292
BuiltinMethodType (in module types), 194
builtins (module), 1230
ButtonBox (class in tkinter.tix), 1067
byte() (in module turtle), 1025
byref() (in module ctypes), 575
byte-code
  file, 1275, 1314
bytearray
  methods, 47
  object, 34, 45, 46
bytearray() (built-in function), 6
bytecode
  object, 1366
bytecode_path() (importlib.abc.PyPycLoader method), 1291
BYTECODE_SUFFIXES (in module importlib.machinery), 1292
byteorder (in module sys), 1215
bytes
  methods, 47
  object, 45, 46
  str (built-in class), 38
bytes (uuid.UUID attribute), 919
bytes() (built-in function), 6
bytes-like object, 1365
bytes_le (uuid.UUID attribute), 919
BytesFeedParser (class in email.parser), 726
BytesGenerator (class in email.generator), 729
BytesIO (class in io), 427
BytesParser (class in email.parser), 727
byteswap() (array.array method), 188
BytesWarning, 71
bz2 (module), 334
BZ2Compressor (class in bz2), 335
BZ2Decompressor (class in bz2), 335
BZ2File (class in bz2), 334

C
  language, 26, 27
  structures, 117
  c_bool (class in ctypes), 579
C_BUILTIN (in module imp), 1278
  c_byte (class in ctypes), 578
  c_char (class in ctypes), 578
  c_char_p (class in ctypes), 578
  c_contiguous (memoryview attribute), 54
  c_double (class in ctypes), 578
C_EXTENSION (in module imp), 1278
  c_float (class in ctypes), 578
  c_int (class in ctypes), 578
  c_int16 (class in ctypes), 578
  c_int32 (class in ctypes), 578
  c_int64 (class in ctypes), 578
  c_int8 (class in ctypes), 578
  c_long (class in ctypes), 579
  c_longdouble (class in ctypes), 578
  c_longlong (class in ctypes), 579
  c_short (class in ctypes), 579
  c_size_t (class in ctypes), 579
  c_ssize_t (class in ctypes), 579
  c_ubyte (class in ctypes), 578
  c_ulong (class in ctypes), 579
  c_ulonglong (class in ctypes), 579
  c_ushort (class in ctypes), 579
  c_void_p (class in ctypes), 579
  c_wchar (class in ctypes), 579
  c_wchar_p (class in ctypes), 579
CAB (class in msilib), 1334
cache_from_source() (in module imp), 1277
CacheFTPHandler (class in urllib.request), 866
calcsize() (in module struct), 118
Calendar (class in calendar), 162
calendar (module), 162
calendar() (in module calendar), 165
call() (in module subprocess), 648
  callback (in module unittest.mock), 1145
call_args (unittest.mock.Mock attribute), 1128
  callback_args (optparse.Option attribute), 477
callback() (in module weakref), 192
callable (built-in function), 6
  CallableProxyType (in module weakref), 192
callback (contextlib.ExitStack method), 1239
callable() (in module weakref), 192
call_count (unittest.mock.Mock attribute), 1126
CALL_FUNCTION (opcode), 1324
CALL_FUNCTION_KW (opcode), 1324
CALL_FUNCTION_VAR (opcode), 1324
CALL_FUNCTION_VAR_KW (opcode), 1324
call_list() (unittest.mock.call method), 1145
call_tracing() (in module sys), 1215
callable (2to3 fixer), 1170
Callable (class in collections.abc), 180
  callback (in module weakref), 192
callable (built-in function), 6
  CallableProxyType (in module weakref), 192
  callback (contextlib.ExitStack method), 1239
  callback_args (optparse.Option attribute), 477
  callback_kwargs (optparse.Option attribute), 477
callbacks (in module gc), 1254
called (unittest.mock.Mock attribute), 1126
CalledProcessError, 650
can_change_color() (in module curses), 525
can_fetch() (urllib.robotparser.RobotFileParser method), 884
can_symlink() (in module test.support), 1178
cancel() (concurrent.futures.Future method), 646
cancel() (sched.scheduler method), 661
cancel() (threading.Timer method), 592
cancel_dump_traceback_later() (in module fault-handler), 1194
cancel_join_thread() (multiprocessing.Queue method), 602
cancelled() (concurrent.futures.Future method), 646
CannotSendHeader, 886
CannotSendRequest, 886
canonic() (bdb.Bdb method), 1190
canonical() (decimal.Context method), 225
canonical() (decimal.Decimal method), 218
capitalize() (str method), 38
captured_stderr() (in module test.support), 1177
captured_stdin() (in module test.support), 1177
captured_stdout() (in module test.support), 1177
captureWarnings() (in module logging), 505
capwords() (in module string), 82
casefold() (str method), 38
cast() (in module ctypes), 575
cast() (memoryview method), 51
cat() (in module nis), 1360
catch_warnings (class in warnings), 1235
category() (in module unicodedata), 111
cbreak() (in module curses), 525
cccd() (ftplib.FTP_TLS method), 895
CDLL (class in ctypes), 570
cell() (in module math), 27, 206
center() (str method), 38
CERT_NONE (in module ssl), 692
CERT_OPTIONAL (in module ssl), 692
CERT_REQUIRED (in module ssl), 692
cert_time_to_seconds() (in module ssl), 692
CertificateError, 689
certificates, 698
CFUNCTYPE() (in module ctypes), 573
CGI
debugging, 852
exceptions, 853
protocol, 847
security, 851
tracebacks, 853
cgi (module), 847
cgi_directories (http.server.CGIHTTPRequestHandler attribute), 932
CGIHandler (class in wsgiref.handlers), 859
CGIHTTPRequestHandler (class in http.server), 932
cgith (module), 853
CGIXMLRPCRequestHandler (class in xmlrpc.server), 950
chain() (in module itertools), 247
chaining
comparisons, 26
ChainMap (class in collections), 166
change_cwd() (in module test.support), 1178
CHANNEL_BINDING_TYPES (in module ssl), 694
channel_class (smtpd.SMTPServer attribute), 914
channels() (naissancedev.oss_audio_device method), 979
CHAR_MAX (in module locale), 995
caracter, 110
CharacterDataHandler() (xml.parsers.expat.xmlparser method), 840
characters() (xml.sax.handler.ContentHandler method), 830
characters_written (BlockingIOError attribute), 69
Charset (class in email.charset), 745
charset() (gettext.NullTranslations method), 986
chdir() (in module os), 400
check (lzma.LZMADataCompressor attribute), 338
check() (imaplib.IMAP4 method), 899
check() (in module tabnanny), 1313
check_call() (in module subprocess), 648
check_output() (doctest.OutputChecker method), 1096
check_output() (in module subprocess), 649
check_warnings() (in module test.support), 1176
checkcache() (in module linecache), 285
checkfuncname() (in module bdb), 1193
CheckList (class in tkinter.tix), 1068
checksum
  Cyclic Redundancy Check, 330
chflags() (in module os), 400
chgat() (curses.window method), 532
childNodes (xml.dom.Node attribute), 813
ChildProcessError, 69
chmod() (xml.dom.Node attribute), 813
choice() (in module random), 240
choices (optparse.Option attribute), 477
chown() (in module os), 401
chown() (in module shutil), 288
chr() (built-in function), 6
chroot() (in module os), 401
Chunk (class in chunk), 975
chunk (module), 974
cipher
  DES, 1350
cipher() (ssl.SSLSocket method), 695
circle() (in module turtle), 1006
CIRCUMFLEX (in module token), 1308
CIRCUMFLEXEQUAL (in module token), 1308
Clamped (class in decimal), 229
class, 1366
Class (class in smtplib), 1306
Class browser, 1071
classmethod() (built-in function), 6
CLD_CONTINUED (in module os), 417
CLD_DUMPED (in module os), 417
CLD_EXITED (in module os), 417
module, 1195
Cmd (class in cmd), 1031
cmd (module), 1031
cmd (subprocess.CalledProcessError attribute), 650
cmd (subprocess.TimeoutExpired attribute), 650
cmdloop() (cmd.Cmd method), 1032
cmp() (in module filecmp), 278
cmp_op (in module subprocess), 650
cmp_to_key() (in module filecmp), 278
CMMSG_LEN() (in module socket), 679
CMMSG_SPACE() (in module socket), 679
code
  object, 61, 307
code (module), 1271
code (urllib.error.HTTPError attribute), 884
code (xml.etree.ElementTree.ParseError attribute), 810
code (xml.parsers.expat.ExpatError attribute), 841
code_info() (in module dis), 1317
Codes, 121
decode, 121
codecs (module), 121
coded_value (http.cookies.Morsel attribute), 934
codeop (module), 1273
codemod (module), 1273
codemod2name (module html.entities), 798
codes (module xml.parsers.expat.errors), 843
CODESET (in module locale), 992
CodeType (in module types), 194
cocreation, 1366
col_offset (ast.AST attribute), 1301
collapse_addresses() (in module ipaddress), 964
collapse_rfc2231_value() (in module email.utils), 751
collect() (in module gc), 1253
collect_incoming_data() (asynchat.async_chat method), 708
collections (module), 165
collections.abc (module), 179
COLOR (in module token), 1308
color() (in module turtle), 1012
color_content() (in module curses), 526
color_pair() (in module curses), 526
colormode() (in module turtle), 1024
colorsys (module), 975
column() (tkinter.ttk.Treeview method), 1060
COLUMNS, 531
columns (os.terminal_size attribute), 398
combinations() (in module itertools), 248
combinations_with_replacement() (in module itertools), 248
combine() (datetime.datetime class method), 145
combing() (in module unicodedata), 111
ComboBox (class in tkinter.tix), 1067
Combobox (class in tkinter.ttk), 1055
COMMA (in module token), 1308
command (http.server.BaseHTTPServer class, 928
CommandCompiler (class in codeop), 1273
commands (pdb command), 1198
comment (http.cookiejar.Cookie attribute), 942
COMMENT (in module tokenize), 1310
comment (zipfile.ZipFile attribute), 344
comment (zipfile.ZipInfo attribute), 345
Comment() (in module xml.etree.ElementTree), 804
comment_url (http.cookiejar.Cookie attribute), 942
commenters (shlex.shlex attribute), 1038
CommentHandler() (xml.parsers.expat.xmlparser method), 840
commit() (msilib.CAB method), 1334
Commit() (msilib.Database method), 1332
commit() (sqlite3.Connection method), 315
common (filecmp.dircmp attribute), 279
Common Gateway Interface, 847
common_dirs (filecmp.dircmp attribute), 279
common_files (filecmp.dircmp attribute), 279
common_funny (filecmp.dircmp attribute), 279
common_types (in module mimetypes), 786
commonprefix() (in module os.path), 269
communicate() (subprocess.Popen method), 654
compare() (decimal.Context method), 225
compare() (decimal.Decimal method), 218
compare() (difflib.Differ method), 105
compare_digest() (in module hmac), 385
compare_networks() (ipaddress.IPv4Network method), 960
compare_networks() (ipaddress.IPv6Network method), 961
COMPARE_OP (opcode), 1322
compare_signal() (decimal.Context method), 225
compare_signal() (decimal.Decimal method), 218
compare_total() (decimal.Context method), 225
compare_total() (decimal.Decimal method), 218
compare_total_mag() (decimal.Context method), 226
compare_total_mag() (decimal.Decimal method), 218
comparing
  objects, 26
comparison
  operator, 26
COMPARISON_FLAGS (in module doctest), 1085
comparisons
  chaining, 26
Compat32 (class in email.policy), 734
compile
  built-in function, 61, 194, 1299
Compile (class in codeop), 1273
compile() (built-in function), 7
compile() (in module py_compile), 1315
compile() (in module code), 1273
compile() (parser.ST method), 1300
compile_command() (in module code), 1271
compile_command() (in module code), 1273
compile_dir() (in module compileall), 1316
compile_file() (in module compileall), 1316
compile_path() (in module compileall), 1316
compileall (module), 1315
compileall command line option
-b, 1316
-d destdir, 1315
-f, 1315
-i list, 1315
-l, 1315
-q, 1315
-x regex, 1315
compilest() (in module parser), 1299
complete() (rlcompleter.Completer method), 116
completedefault() (cmd.Cmd method), 1032
complex
    built-in function, 27
Complex (class in numbers), 203
complex number, 1366
    literals, 26
    object, 26
complex() (built-in function), 7
compress() (bz2.BZ2Compressor method), 335
compress() (in module bz2), 335
compress() (in module gzip), 333
compress() (in module itertools), 249
compress() (in module lzma), 338
compress() (in module zlib), 329
compress() (lzma.LZMACompressor method), 337
compress() (zlib.Compress method), 321
compress_size (zipfile.ZipInfo attribute), 345
compress_type (zipfile.ZipInfo attribute), 345
compressed (ipaddress.IPv4Address attribute), 955
compressed (ipaddress.IPv4Network attribute), 959
compressed (ipaddress.IPv6Address attribute), 956
compressed (ipaddress.IPv6Network attribute), 961
compression() (ssl.SSLSocket method), 695
CompressionError, 347
compressobj() (in module zlib), 329
COMSPEC, 416, 652
concat() (in module operator), 263
concatenation
    operation, 32
concurrent.futures (module), 643
Condition (class in multiprocessing), 605
Condition (class in threading), 589
    condition (pdb command), 1198
condition() (msilib.Control method), 1335
Condition() (multiprocessing.managers.SyncManager method), 610
ConfigParser (class in configparser), 372
configparser (module), 360
configuration
    file, 360
    file, debugger, 1197
    file, path, 1266
configuration information, 1227
configure() (tkinter.ttk.Style method), 1063
configure_mock() (unittest.mock.Mock method), 1125
confstr() (in module os), 420
confstr_names (in module os), 420
conjugate() (complex number method), 27
    conjugate() (decimal.Decimal method), 218
    conjugate() (numbers.Complex method), 203
    conn (smtp.SMTPChannel attribute), 915
    connect() (asyncore.dispatcher method), 706
    connect() (ftplib.FTP method), 893
    connect() (http.client.HTTPConnection method), 888
    connect() (in module sqlite3), 313
    connect() (multiprocessing.managers.BaseManager method), 609
    connect() (smtplib.SMTP method), 911
    connect() (socket.socket method), 680
    connect_ex() (socket.socket method), 680
    Connection (class in multiprocessing), 603
    Connection (class in sqlite3), 315
    ConnectionAbortedError, 70
    ConnectionError, 69
    ConnectionRefusedError, 70
    ConnectionResetError, 70
    ConnectRegistry() (in module winreg), 1338
    const (optparse.Option attribute), 477
    constructor() (in module copyreg), 304
    container
        iteration over, 32
        Container (class in collections.abc), 180
        contains() (in module operator), 263
content type
    MIME, 784
    content_type (email.headerregistry.ContentTypeHeader attribute), 739
    ContentDispositionHeader (class in email.headerregistry), 739
    ContenHandler (class in xml.sax.handler), 827
    ContentTooShortError, 884
    ContentTransferEncoding (class in email.headerregistry), 739
    ContentTypeHeader (class in email.headerregistry), 739
    Context (class in decimal), 224
    context (ssl.SSLSocket attribute), 696
    context management protocol, 60
    context manager, 60, 1366
    context_diff() (in module difflib), 99
    ContextDecorator (class in contextlib), 1237
    contextlib (module), 1236
    contextmanager() (in module contextlib), 1236
    contiguous (memoryview attribute), 54
    continue (pdb command), 1198
    CONTINUE_LOOP (opcode), 1320
    Control (class in msilib), 1335
    Control (class in tkinter.tix), 1067
    control() (msilib.Dialog method), 1335
    control() (select.queue method), 667
    controlnames (in module curses.ascii), 543
    controls() (ossaudiodev.oss_mixer_device method), 980
    ConversionError, 379
    conversions
        numeric, 27

Index
| convert_arg_line_to_args() (argparse.ArgumentParser method) | 464 |
| convert_field() (string.Formatter method) | 75 |
| Cookie (class in http.cookiejar) | 936 |
| CookieError | 932 |
| CookieJar (class in http.cookiejar) | 936 |
| cookiejar (urllib.request.HTTPCookieProcessor attribute) | 870 |
| CookiePolicy (class in http.cookiejar) | 936 |
| Coordinated Universal Time | 432 |
| Copy | 1073 |
| Copy module | 304 |
| Copy protocol | 298 |
| copy (module) | 195 |
| copy() (decimal.Context method) | 224 |
| copy() (dict method) | 58 |
| copy() (hashlib.hash method) | 384 |
| copy() (hmac.HMAC method) | 385 |
| copy() (imaplib.IMAP4 method) | 899 |
| copy() (in module copy) | 196 |
| copy() (in module multiprocessing.sharedctypes) | 607 |
| copy() (in module shutil) | 286 |
| copy() (pipes.Template method) | 1357 |
| copy() (sequence method) | 34 |
| copy() (set method) | 55 |
| copy() (types.MappingProxyType method) | 195 |
| copy() (zlib.Compress method) | 331 |
| copy() (zlib.Decompress method) | 331 |
| copy2() (in module shutil) | 286 |
| copy_abs() (decimal.Decimal method) | 226 |
| copy_abs() (decimal.Decimal method) | 226 |
| copy_decimal() (decimal.Decimal method) | 224 |
| copy_location() (in module ast) | 1304 |
| copy_negate() (decimal.Decimal method) | 226 |
| copy_negate() (decimal.Decimal method) | 219 |
| copy_sign() (decimal.Decimal method) | 226 |
| copy_sign() (decimal.Decimal method) | 219 |
| copyfile() (in module shutil) | 285 |
| copyfileobj() (in module shutil) | 285 |
| copying files | 285 |
| copyingmode() (in module shutil) | 285 |
| copyreg (module) | 304 |
| copyright (built-in variable) | 23 |
| copyright (in module sys) | 1215 |
| copytree() (in module math) | 206 |
| copytree() (in module shutil) | 286 |
| copytree() (in module shutil) | 286 |
| cos() (in module math) | 211 |
| cos() (in module math) | 208 |
| cosh() (in module math) | 212 |
| cosh() (in module math) | 209 |
| count() (array.array method) | 188 |
| count() (collections.deque method) | 171 |
| count() (in module iter tools) | 249 |
| count() (sequence method) | 32 |
| count() (str method) | 38 |
| Counter (class in collections) | 168 |
| countOf() (in module operator) | 263 |
| countTestCases() (unittest.TestCase method) | 1112 |
| countTestCases() (unittest.TestSuite method) | 1114 |
| CoverageResults (class in trace) | 1213 |
| cProfile (module) | 1202 |
| CPU time | 433 |
| cpu_count() (in module multiprocessing) | 602 |
| CPython | 1366 |
| CRC (zipfile.ZipInfo attribute) | 345 |
| crc32() (in module binascii) | 790 |
| crc32() (in module zlib) | 330 |
| crc_hqx() (in module binascii) | 790 |
| create() (imaplib.IMAP4 method) | 899 |
| create() (in module venv) | 1183 |
| create() (venv.EnvBuilder method) | 1182 |
| create_aggregate() (sqlite3.Connection method) | 316 |
| create_autospec() (in module unittest.mock) | 1146 |
| create_collation() (sqlite3.Connection method) | 316 |
| create_configuration() (venv.EnvBuilder method) | 1182 |
| create_connection() (in module socket) | 676 |
| create_decimal() (decimal.Context method) | 225 |
| create_decimal_from_float() (decimal.Context method) | 225 |
| create_function() (sqlite3.Connection method) | 315 |
| CREATE_NEW_CONSOLE (in module subprocess) | 656 |
| CREATE_NEW_PROCESS_GROUP (in module subprocess) | 656 |
| create_socket() (asyncore.dispatcher method) | 706 |
| create_stats() (profile.Profile method) | 1203 |
| create_string_buffer() (in module ctypes) | 575 |
| create_system (zipfile.ZipInfo attribute) | 345 |
| create_unicode_buffer() (in module cTypes) | 575 |
| create_version (zipfile.ZipInfo attribute) | 345 |
| createAttribute() (xml.dom.Document method) | 815 |
| createAttributeNS() (xml.dom.Document method) | 816 |
| createComment() (xml.dom.Document method) | 815 |
| createDocument() (xml.dom.DOMImplementation method) | 812 |
| createDocumentType() (xml.dom.DOMImplementation method) | 812 |
| createElement() (xml.dom.Document method) | 815 |
| createElementNS() (xml.dom.Document method) | 815 |
| CreateKey() (in module winreg) | 1338 |
| CreateKeyEx() (in module winreg) | 1338 |
| createLock() (logging.Handler method) | 495 |
| createLock() (logging.NullHandler method) | 515 |
| createProcessingInstruction() (xml.dom.Document method) | 815 |
| CreateRecord() (in module msilib) | 1331 |
| createSocket() (logging.handlers.SocketHandler method) | 518 |
| createTextNode() (xml.dom.Document method) | 815 |
| createTextNode() (xml.dom.Document method) | 815 |
| credits (built-in variable) | 23 |
| critical() (in module logging) | 502 |
| critical() (logging.Logger method) | 494 |
| CRNCYSTR (in module locale) | 993 |
cross() (in module audioop), 965
crypt
  module, 1348
crypt (module), 1350
crypt() (in module crypt), 1350
crypt3(), 1350, 1351
cryptography, 383, 386
csv, 355
csv (module), 355
cte (email.headerregistry.ContentTransferEncoding attribute), 739
cte_type (email.policy.Policy attribute), 732
cтверd() (in module os), 388
ctime() (datetime.date method), 143
ctime() (datetime.datetime method), 149
ctime() (in module time), 433
ctrl() (in module curses.ascii), 543
CTRL_BREAK_EVENT (in module signal), 711
CTRL_C_EVENT (in module signal), 711
ctypes (module), 553
curdir (in module os), 420
currency() (in module locale), 995
current() (tkinter.ttk.Combobox method), 1055
current_process() (in module multiprocessing), 602
current_thread() (in module threading), 583
CurrentByteIndex (xml.parsers.expat.xmlparser attribute), 839
CurrentColumnNumber (xml.parsers.expat.xmlparser attribute), 839
currentframe() (in module inspect), 1264
CurrentLineNumber (xml.parsers.expat.xmlparser attribute), 839
curs_set() (in module curses), 526
curses (module), 525
curses.ascii (module), 541
curses.panel (module), 543
curses.textpad (module), 540
Cursor (class in sqlite3), 319
cursor() (sqlite3.Connection method), 315
cursyncup() (curses.window method), 532
CUT, 1073
cwd() (ftplib.FTP method), 895
cycle() (in module itertools), 249
Cyclic Redundancy Check, 330

D
D_FMT (in module locale), 993
D_T_FMT (in module locale), 993
daemon (multiprocessing.Process attribute), 598
daemon (threading.Thread attribute), 586
data
  packing binary, 117
tabular, 355
Data (class in plistlib), 380
data (collections.UserDict attribute), 178
data (collections.UserList attribute), 179
data (select.kevent attribute), 668
data (urllib.request.Request attribute), 866
data (xml.dom.Comment attribute), 817
data (xml.dom.ProcessingInstruction attribute), 818
data (xml.dom.Text attribute), 817
data (xmlrpc.client.Binary attribute), 946
data() (xml.etree.ElementTree.TreeBuilder method), 809
database
  Unicode, 110
databases, 311
DatagramHandler (class in logging.handlers), 519
date (class in datetime), 141
date() (datetime.datetime method), 147
date() (nntplib.NNTP method), 908
date_time (zipfile.ZipInfo attribute), 345
date_time_string() (http.server.BaseHTTPRequestHandler method), 930
DateHeader (class in email.headerregistry), 738
datetime (class in datetime), 144
datetime (email.headerregistry.DateHeader attribute), 738
datetime (module), 137
day (datetime.datetime attribute), 142
day (datetime.datetime attribute), 146
day_abbr (in module calendar), 165
day_name (in module calendar), 165
daylight (in module time), 433
Daylight Saving Time, 432
DbfilenameShelf (class in shelve), 306
dbm (module), 308
dbm.dumb (module), 311
dbm.gnu
  module, 306
dbm.gnu (module), 309
dbm.ndbm
  module, 306
dbm.ndbm (module), 310
dbdebug (imaplib.IMAP4 attribute), 902
DEBUG (in module re), 87
dbdebug (shlex.shlex attribute), 1038
dbdebug (zipfile.ZipFile attribute), 344
dbdebug() (in module doctest), 1097
dbdebug() (in module logging), 501
dbdebug() (logging.Logger method), 493
dbdebug() (pipes.Template method), 1357
dbdebug() (unittest.TestCase method), 1107
dbdebug() (unittest.TestSuite method), 1114
DEBUG_BYTECODE_SUFFIXES (in module importlib.machinery), 1292
DEBUG_COLLECTABLE (in module gc), 1255
DEBUG_LEAK (in module gc), 1255
DEBUG_SAVEALL (in module gc), 1255
dbdebug_src() (in module doctest), 1098
DEBUG_STATS (in module gc), 1255
DEBUG_UNCOLLECTABLE (in module gc), 1255
dbuser, 1073, 1220, 1224
  configuration file, 1197
dbuser, 1195
CGI, 852
The Python Library Reference, Release 3.3.3

DebuggingServer (class in smtpd), 914
debugelevel (http.client.HTTPResponse attribute), 889
DebugRunner (class in doctest), 1098
Decimal (class in decimal), 216
decimal (module), 212
decimal() (in module unicodedata), 110
DecimalException (class in decimal), 229
decode
Codecs, 121
decode() (bytearray method), 47
decode() (bytes method), 47
decode() (codecs.Codec method), 125
decode() (in module base64), 788
decode() (in module codecs), 121
decode() (in module quopri), 791
decode() (in module uu), 792
decode() (json.JSONDecoder method), 765
decode() (xmlrpc.client.Binary method), 946
decode() (xmlrpc.client.DateTime method), 945
decode_header() (in module email.header), 745
decode_header() (in module ntplib), 908
decode_params() (in module email.utils), 751
decode_rfc2231() (in module email.utils), 751
decodebytes() (in module base64), 788
DecodedGenerator (class in email.generator), 730
decodestring() (in module base64), 788
decodestring() (in module codecs), 121
decomposition() (in module unicodedata), 111
decompress() (bz2.BZ2Compressor method), 335
decompress() (in module bz2), 335
decompress() (in module gzip), 333
decompress() (in module lzma), 338
decompress() (in module zlib), 330
decompress() (lzma.LZMADecompressor method), 338
decompress() (zlib.decompress method), 331
decompressobj() (in module zlib), 330
decorator, 1366
DEDENT (in module token), 1308
dedent() (in module textwrap), 108
decopy() (in module copy), 196
def_prog_mode() (in module curses), 526
def_shell_mode() (in module curses), 526
default (in module email.policy), 736
DEFAULT (in module unittest.mock), 1145
default (inspect.Parameter attribute), 1260
default (optparse.Option attribute), 477
default() (cmd.Cmd method), 1032
default() (json.JSONEncoder method), 766
DEFAULT_BUFFER_SIZE (in module io), 422
default_bfs() (xml.dom.pulldom), 825
default_factory (collections.defaultdict attribute), 173
DEFAULT_FORMAT (in module tarfile), 347
default_open() (urllib.request.BaseHandler method), 869
DEFAULT_PROTOCOL (in module pickle), 294
default_timer() (in module timeit), 1208
DefaultContext (class in decimal), 224
DefaultCookiePolicy (class in http.cookiejar), 936
defaultdict (class in collections), 173
DefaultHandler() (xml.parsers.expat.xmlparser method), 840
DefaultHandlerExpand() (xml.parsers.expat.xmlparser method), 840
defaults() (configparser.ConfigParser method), 372
defaultTestLoader (in module unittest), 1117
defaultTestResult (unittest.TestResult method), 1112
defects (email.headerregistry.BaseHeader attribute), 737
defects (email.message.Message attribute), 725
defpath (in module os), 421
DefragResult (class in urllib.parse), 882
DefragResultBytes (class in urllib.parse), 882
degrees() (in module math), 209
degrees() (in module turtle), 1009
del
statement, 34, 56
del_param() (email.message.Message method), 724
delattr() (built-in function), 7
delay() (in module turtle), 1021
delay_output() (in module curses), 526
delayload (http.cookiejar.FileCookieJar attribute), 938
delch() (curses.window method), 532
delet() (poplib.POP3 method), 897
delete() (ftplib.FTP method), 895
delete() (imaplib.IMAP4 method), 899
delete() (tkinter.ttk.Treeview method), 1061
DELETE_ATTR (opcode), 1322
DELETE_DEREF (opcode), 1323
DELETE_FAST (opcode), 1323
DELETE_GLOBAL (opcode), 1322
DELETE_NAME (opcode), 1322
DELETE_SUBSCR (opcode), 1320
deleatext() (imaplib.IMAP4 method), 899
DeleteKey() (in module winreg), 1338
DeleteKeyEx() (in module winreg), 1338
deletel() (curses.window method), 532
deleteMe() (bdb.Breakpoint method), 1189
DeleteValue() (in module winreg), 1339
delimiter (csv.Dialect attribute), 358
delim() (in module operator), 263
delivery_challenge() (in module multiprocessing.connection), 616
demo_app() (in module wsgiref.simple_server), 857
denominator (fractions.Fraction attribute), 238
denominator (numbers.Rational attribute), 204
DeprecationWarning, 70
decode (class in collections), 170
dequeue() (logging.handlers.QueueListener method), 523
DER_cert_to_PEM_cert() (in module ssl), 692
derwin() (curses.window method), 532
DES
cipher, 1350
description (sqlite3.Cursor attribute), 322
description() (nntplib.NNTP method), 906
descriptions() (nntplib.NNTP method), 906
descriptor, 1366
dest (optparse.Option attribute), 477
detach() (io.BufferedIOBase method), 426
detach() (io.TextIOBase method), 429
detach() (socket.socket method), 680
detach() (tkinter.ttk.Treeview method), 1061
detach() (winreg.PyHKEY method), 1344
detect_encoding() (in module tokenize), 1310
deterministic profiling, 1200
device_encoding() (in module os), 393
devnull (in module os), 421
DEVNULL (in module subprocess), 650
devpoll() (in module select), 663
dgettext() (in module gettext), 984
dictionary, 1366
dict (2to3 fixer), 1170
dict (built-in class), 56
dict() (multiprocessing.managers.SyncManager method), 610
dictConfig() (in module logging.config), 505
dictionary, 1366
   type, operations on, 56
DictReader (class in csv), 357
DictWriter (class in csv), 357
diff_files (filecmp.dircmp attribute), 279
differ (class in difflib), 98, 105
difference() (set method), 55
difference_update() (set method), 56
difflib (module), 98
digest() (hashlib.hash method), 384
digest() (hmac.HMAC method), 385
digit() (in module unicodedata), 110
digits (in module string), 73
dir() (built-in function), 8
dir() (ftpplib.FTP method), 894
dircmp (class in filecmp), 278
directory
   changing, 400
creating, 403
deleting, 287, 404
site-packages, 1266
site-python, 1266
traversal, 409, 410
walking, 409, 410
Directory (class in msilib), 1334
DirList (class in tkinter.tix), 1068
dirname() (in module os.path), 270
DirSelectBox (class in tkinter.tix), 1068
DirSelectDialog (class in tkinter.tix), 1068
DirTree (class in tkinter.tix), 1068
dis (module), 1317
dis() (in module dis), 1317
dis() (in module pickletools), 1325
disable (pdb command), 1197
disable() (bdb.Breakpoint method), 1189
disable() (in module faulthandler), 1194
disable() (in module gc), 1253
disable() (in module logging), 503
disable() (profile.Profile method), 1203
disable_interspersed_args() (optparse.OptionParser method), 481
DisableReflectionKey() (in module winreg), 1342
disassemble() (in module dis), 1318
discard (http.cookiejar.Cookie attribute), 942
discard() (mailbox.Mailbox method), 769
discard() (mailbox.MH method), 774
discard() (set method), 56
discard_buffers() (asyncio.async_chat method), 708
disco() (in module dis), 1318
discover() (unittest.TestLoader method), 1115
disk_usage() (in module shutil), 288
dispatch_call() (bdb.Bdb method), 1190
dispatch_exception() (bdb.Bdb method), 1191
dispatch_line() (bdb.Bdb method), 1190
dispatch_return() (bdb.Bdb method), 1191
dispatch_table (pickle.Pickler attribute), 296
dispatcher (class in asyncio), 705
dispatcher_with_send (class in asyncio), 706
display (pdb command), 1199
display_name (email.headerregistry.Address attribute), 740
display_name (email.headerregistry.Group attribute), 741
displayhook() (in module sys), 1216
dist() (in module platform), 547
distance() (in module turtle), 1009
distb() (in module dis), 1317
distutils (module), 1269
divide() (decimal.Context method), 226
divide_int() (decimal.Context method), 226
DivisionByZero (class in decimal), 229
divmod() (built-in function), 8
divmod() (decimal.Context method), 226
DllCanUnloadNow() (in module ctypes), 575
DllGetClassObject() (in module ctypes), 575
dllhandle (in module sys), 1216
dngettext() (in module gettext), 984
do_clear() (bdb.Bdb method), 1191
do_command() (curses.textpad.Textbox method), 541
do_GET() (http.server.SimpleHTTPRequestHandler method), 931
do_handshake() (ssl.SSLSocket method), 695
do_HEAD() (http.server.SimpleHTTPRequestHandler method), 931
do_POST() (http.server.CGIHTTPRequestHandler method), 932
doc_header (cmd.Cmd attribute), 1033
DocCGIXMLRPCRequestHandler (class in xmlrpc.server), 953
DocFileSuite() (in module doctest), 1090
doublequote (csv.Dialect attribute), 358
DOUBLESLASH (in module token), 1308
DOUBLESLASHEQUAL (in module token), 1308
DOUBLESTAR (in module token), 1308
DOUBLESTAREQUAL (in module token), 1308
doupdate() (in module curses), 526
down (pdb command), 1197
down() (in module turtle), 1010
drop_whitespace (textwrap.TextWrapper attribute), 109
dropwhile() (in module itertools), 250
dst() (datetime.datetime method), 148
dst() (datetime.datetime method), 159
dst() (datetime.timezone method), 159
Duck (class in dummy_threading), 668
dump() (in module ast), 1305
dump() (in module json), 763
dump() (in module marshal), 308
dump() (in module pickle), 295
dump() (pickle.Pickler method), 296
dump_stats() (profile.Profile method), 1203
dump_stats() (pstats.Stats method), 1204
dump_traceback() (in module faulthandler), 1193
dump_traceback_later() (in module faulthandler), 1194
dumps() (in module json), 763
dumps() (in module marshal), 308
dumps() (in module pickle), 295
dumps() (in module xmlrpc.client), 949
dup() (in module os), 393
dup2() (in module os), 393
DUP_TOP (opcode), 1319
DUP_TOP_TWO (opcode), 1319
DuplicateOptionError, 376
DuplicateSectionError, 375
dwFlags (subprocess.STARTUPINFO attribute), 655
e (in module cmath), 212
e (in module math), 209
E2BIG (in module errno), 548
EACCESS (in module errno), 548
EADDRINUSE (in module errno), 552
EADDRNOTAVAIL (in module errno), 552
EADV (in module errno), 551
EAFNOSUPPORT (in module errno), 552
EAGAIN (in module errno), 548
EALREADY (in module errno), 553
east_asian_width() (in module unicodedata), 111
EBADBE (in module errno), 550
EBADF (in module errno), 548
EBADFD (in module errno), 551
EBADMSG (in module errno), 551
EBAD (in module errno), 550
Index 1417

EBADRQC (in module errno), 550
EBADSLT (in module errno), 550
EBFONT (in module errno), 550
EBUSY (in module errno), 548
ECHILD (in module errno), 548
echo() (in module curses), 526
echochar() (curses.window method), 533
ECHRNNG (in module errno), 550
ECOMM (in module errno), 551
ECONNABORTED (in module errno), 552
ECONNREFUSED (in module errno), 552
ECONNRESET (in module errno), 552
EDEADLK (in module errno), 549
EDEADLOCK (in module errno), 550
EDESTADDRREQ (in module errno), 552
EDESTADDRREQ (in module errno), 552
edit() (curses.textpad.Textbox method), 540
EDOM (in module errno), 549
EDOTDOT (in module errno), 551
EQUOT (in module errno), 553
EEXIST (in module errno), 548
EFAULT (in module errno), 548
EFOREST (in module errno), 549
effective() (in module bdb), 1193
ehlo() (smtplib.SMTP method), 911
ehlo_or_helo_if_needed() (smtplib.SMTP method), 911
EHOSTDOWN (in module errno), 553
EHOSTUNREACH (in module errno), 553
EIDRM (in module errno), 550
EILSEQ (in module errno), 551
EINPROGRESS (in module errno), 553
EINTR (in module errno), 548
EINVAL (in module errno), 551
EIP (in module errno), 549
EIO (in module errno), 548
EISCONN (in module errno), 552
EISDIR (in module errno), 549
EISNAM (in module errno), 553
EL2HLT (in module errno), 550
EL2NSYNC (in module errno), 550
EL3HLT (in module errno), 550
EL3RST (in module errno), 550
Element (class in xml.etree.ElementTree), 805
ElementDeclHandler() (xml.parsers.expat.xmlparser method), 839
elements() (collections.Counter method), 168
ElementTree (class in xml.etree.ElementTree), 807
ELIBACC (in module errno), 551
ELIBBAD (in module errno), 551
ELIBEXEC (in module errno), 551
ELIBMAX (in module errno), 551
ELIBSCN (in module errno), 551
Ellinghouse, Lance, 792
Ellipsis (built-in variable), 23
ELLIPSIS (in module doctest), 1085
ELLIPSIS (in module token), 1308
ELNRNG (in module errno), 550
ELOOP (in module errno), 550
email (module), 719
email.charset (module), 745
email.encoders (module), 747
email.errors (module), 748
email.generator (module), 728
email.header (module), 743
email.headerregistry (module), 736
email.iterators (module), 751
email.message (module), 719
email.mime (module), 741
email.parser (module), 725
email.policy (module), 731
email.utils (module), 749
EmailPolicy (class in email.policy), 734
EMFILE (in module errno), 549
emit() (logging.FileHandler method), 515
emit() (logging.Handler method), 496
emit() (logging.handlers.BufferingHandler method), 522
emit() (logging.handlers.DatagramHandler method), 519
emit() (logging.handlers.HTTPHandler method), 523
emit() (logging.handlers.NTEventLogHandler method), 521
emit() (logging.handlers.QueueHandler method), 523
emit() (logging.handlers.RotatingFileHandler method), 517
emit() (logging.handlers.SMTPHandler method), 522
emit() (logging.handlers.SocketHandler method), 518
emit() (logging.handlers.SysLogHandler method), 520
emit() (logging.handlers.TimedRotatingFileHandler method), 518
emit() (logging.handlers.WatchedFileHandler method), 516
emit() (logging.NullHandler method), 515
emit() (logging.StreamHandler method), 515
EMLINK (in module errno), 549
Empty, 662
empty (inspect.Parameter attribute), 1260
empty (inspect.Signature attribute), 1259
empty() (multiprocessing.Queue method), 601
empty() (multiprocessing.SimpleQueue method), 602
empty() (queue.Queue method), 662
empty() (sched.scheduler method), 661
EMPTY_NAMESPACE (in module xml.dom), 811
emptyline() (cmd.Cmd method), 1032
EMSGSIZE (in module errno), 552
EMULTIHOP (in module errno), 551
enable (pdb command), 1198
enable() (bdb.Breakpoint method), 1189
enable() (in module cgitb), 853
enable() (in module faulthandler), 1194
enable() (not module gc), 1253
enable() (profile.Profile method), 1203
enable_callback_tracebacks() (in module sqlite3), 314
enable_interspersed_args() (optparse.OptionParser method), 481
enable_load_extension() (sqlite3.Connection method), 317
enable_traversal() (tkinter.ttk.Notebook method), 1057
ENABLE_USER_SITE (in module site), 1267
EnableReflectionKey() (in module winreg), 1342
ENAMETOOLONG (in module errno), 549
ENAVAIL (in module errno), 553
enclose() (curses.window method), 533
encode
Codecs, 121
encode() (codecs.Codec method), 125
encode() (codecs.IncrementalEncoder method), 126
encode() (email.header.Header method), 744
encode() (in module base64), 788
encode() (in module codecs), 121
encode() (in module quopri), 791
encode() (in module uu), 792
encode() (json.JSONEncoder method), 766
encode() (str method), 39
encode() (xmlrpc.client.Binary method), 946
encode() (xmlrpc.client.DateTime method), 945
encode() (xmlrpc.client.Enumeration method), 748
encode_base64() (in module email.encoders), 748
encode_base64_url() (in module email.encoders), 748
encode_noop() (in module email.encoders), 748
encode_quopri() (in module email.encoders), 748
encode_rfc2231() (in module email.mime.multipart), 751
encodebytes() (in module base64), 788
EncodedFile() (in module codecs), 124
encodePriority() (logging.handlers.SysLogHandler method), 520
encodestring() (in module base64), 788
encodestring() (in module codecs), 124
encoding
base64, 787
quoted-printable, 791
encoding (curses.window attribute), 533
ENCODING (in module tarfile), 347
ENCODING (in module tokenize), 1310
encoding (io.TextIOWrapper attribute), 429
encoding (UnicodeError attribute), 68
encodings.idna (module), 136
encodings.mbcs (module), 136
encodings.utf_8_sig (module), 136
encodings_map (in module mimetypes), 785
encodings_map (mimetypes.Attribute), 786
end (UnicodeError attribute), 69
end() (re.match attribute), 93
endpos (re.match method), 39
endswitch() (str method), 39
endwin() (in module curses), 526
ENETDOWN (in module errno), 552
ENETRESET (in module errno), 552
ENETUNREACH (in module errno), 552
ENFILE (in module errno), 552
ENOANO (in module errno), 550
ENOBUFS (in module errno), 552
ENOCSI (in module errno), 550
ENODATA (in module errno), 550
ENODEV (in module errno), 548
ENOENT (in module errno), 548
ENOEXEC (in module errno), 548
ENOLCK (in module errno), 549
ENOLINK (in module errno), 551
ENOMEM (in module errno), 548
ENOMSG (in module errno), 549
ENONET (in module errno), 550
ENOPKG (in module errno), 550
ENOPROTOOPT (in module errno), 552
ENOSPC (in module errno), 549
ENOSPC (in module errno), 552
ENOTBLK (in module errno), 548
ENOTCONN (in module errno), 552
ENOTEMPTY (in module errno), 549
ENOTFOUND (in module errno), 549
ENOTPERM (in module errno), 549
ENOTSOCK (in module errno), 550
ENOTTY (in module errno), 549
ENOTUNIQ (in module errno), 551
enqueue() (logging.handlers.QueueHandler method), 523
enqueue_sentinel() (logging.handlers.QueueListener method), 524
environment variables
  deleting, 392
  setting, 390
EnvironmentError, 69
Environments
  virtual, 1180
EnvironmentVarGuard (class in test.support), 1179
ENXIO (module errno), 548
eof (bz2.BZ2Decompressor attribute), 335
eof (lzma.LZMADecompressor attribute), 338
eof (shlex.shlex attribute), 1039
eof (zlib.Decompress attribute), 331
EOFError, 66
EOPNOTSUPP (module errno), 552
EOVERFLOW (module errno), 551
EPERM (module errno), 548
EPFNOSUPPORT (module errno), 552
epilogue (email.message.Message attribute), 725
EPipe (module errno), 549
epoch, 432
epoll() (module select), 664
EPROTO (module errno), 551
EPROTONOSUPPORT (module errno), 552
EPROTOTYPE (module errno), 552
eq() (module operator), 261
EQUEQUAL (module token), 1308
EQUAL (module token), 1308
ERA (module locale), 993
ERA_D_FMT (module locale), 994
ERA_D_T_FMT (module locale), 993
ERA_T_FMT (module locale), 994
ERANGE (module errno), 549
erase() (curses.window method), 533
erasechar() (module curses), 526
EREMCHG (module errno), 551
EREMOTE (module errno), 551
EREMOTEIO (module errno), 553
ERESTART (module errno), 551
erf() (module math), 209
erfc() (module math), 209
EROFS (module errno), 549
ERR (module curses), 536
errcheck (ctypes._FuncPtr attribute), 572
errcode (xmlrpc.client.ProtocolError attribute), 947
ermsg (xmlrpc.client.ProtocolError attribute), 947
erno
  module, 67
erno (module), 548
Error, 288, 358, 375, 379, 783, 789, 791, 792, 845, 970, 973, 991
error, 90, 117, 196, 308–311, 329, 387, 490, 525, 663, 669, 674, 837, 965
error() (argparse.ArgumentParser method), 465
error() (module logging), 502
error() (logging.Logger method), 494
error() (urllib.request.OpenerDirector method), 868
error() (xml.sax.handler.ErrorHandler method), 831
error_body (wsgiref.handlers.BaseHandler attribute), 861
def error_content_type (http.server.BaseHTTPRequestHandler attribute), 861
def error_headers (wsgiref.handlers.BaseHandler attribute), 861
def error_leader() (shlex.shlex method), 1038
def error_message_format (http.server.BaseHTTPRequestHandler attribute), 929
def error_output() (wsgiref.handlers.BaseHandler method), 861
def error_perm, 892
def error_proto, 892, 896
def error_reply, 892
def error_status (wsgiref.handlers.BaseHandler attribute), 861
def error_temp, 892

ErrorByteIndex (xml.parsers.expat.xmlparser attribute), 839
def errorcode (in module errno), 548

ErrorCode (xml.parsers.expat.xmlparser attribute), 839

ErrorColumnNumber (xml.parsers.expat.xmlparser attribute), 839

ErrorHandler (class in xml.sax.handler), 827

ErrorLineNumber (xml.parsers.expat.xmlparser attribute), 839

Errors
logging, 491
errors (io.TextIOBase attribute), 429
ersors (unittest.TestCase attribute), 1116

ErrorString() (in module xml.parsers.expat), 837

ERRORTOKEN (in module token), 1308
def escape (shlex.shlex attribute), 1038
def escape() (in module cgi), 851

def escape() (in module html), 793

def escape() (in module re), 90

def escape() (in module xml.sax.saxutils), 832

def escapechar (csv.Dialect attribute), 358

def escapedquotes (shlex.shlex attribute), 1038

ESHUTDOWN (in module errno), 552

ESOCKTNOSUPPORT (in module errno), 552

ESPIPE (in module errno), 549

ESRCCH (in module errno), 548

ESRMNT (in module errno), 551

ESTALE (in module errno), 553

ESTRPIPE (in module errno), 551

ETIME (in module errno), 550

ETIMEDOUT (in module errno), 552

Etnity() (decimal.Context method), 225

ETOOMANYREFS (in module errno), 552

Et voluptas (decimal.Context method), 225

ETXTBSY (in module errno), 549

EUCLean (in module errno), 553

EUNATCH (in module errno), 550

EUSERS (in module errno), 551

def eval

    built-in function, 62, 198, 1299
def eval() (built-in function), 9

def Event (class in multiprocessing), 605
def Event (class in threading), 591
def event() (msilib.Control method), 1335
def event() (multiprocessing.managers.SyncManager method), 610
def events (widgets), 1049

def ETIMEDOUT (in module errno), 552

EX_CANTCREAT (in module os), 413

EX_CONFIG (in module os), 413

EX_DATAERR (in module os), 412

EX_IOERR (in module os), 413

EX_NOHOST (in module os), 413

EX_INPUT (in module os), 412

EX_NOPERM (in module os), 413

EX_NOTFOUND (in module os), 413

EX_OK (in module os), 412

EX_OSERR (in module os), 413

EX_OSFILE (in module os), 413

EX_PROTOCOL (in module os), 413

EX_SOFTWARE (in module os), 413

EX_TEMPFAIL (in module os), 413

EXUnavailable (in module os), 413

EX_USAGE (in module os), 412

Example (class in doctest), 1093
def example (doctest.DocTestFailure attribute), 1098
def example (doctest.UnexpectedException attribute), 1098
def examples (doctest.DocTest attribute), 1092

def exc_info (doctest.UnexpectedException attribute), 1098

def exc_info() (in module sys), 1217

def exc_msg (doctest.Example attribute), 1093
def excel (class in csv), 357

def excel_table (class in csv), 357

except

    statement, 65
except (2to3 fixer), 1170

excepthook() (in module sys), 853, 1217

Exception, 66

exception() (concurrent.futures.Future method), 646

exception() (in module logging), 502

exception() (logging.Logger method), 494

exceptions

    in CGI scripts, 853

EXDEV (in module errno), 548

EXECUTE (in module errno), 548

exec

    built-in function, 62, 1299

exec (2to3 fixer), 1170

exec() (built-in function), 9

exec_prefix (in module sys), 1217

execfile (2to3 fixer), 1170

execl() (in module os), 411

execl() (in module os), 411

execlp() (in module os), 411

execlpe() (in module os), 411

executable (in module sys), 1217

Execute() (msilib.View method), 1332
execute() (sqlite3.Connection method), 315
execute() (sqlite3.Cursor method), 319
executeMany() (sqlite3.Connection method), 315
executeMany() (sqlite3.Cursor method), 320
executeScript() (sqlite3.Connection method), 315
executeScript() (sqlite3.Cursor method), 321
ExecutionLoader (class in importlib.abc), 1289
Executor (class in concurrent.futures), 643
evcc() (in module os), 411
evcc() (module os), 411
evccp() (module os), 411
evccve() (module os), 411
ExFileSelectBox (class in tkinter.tix), 1068
EXFULL (in module errno), 550
exists() (module os.path), 270
exists() (tkinter.ttk.Treeview method), 1061
exit (built-in variable), 23
exit() (argparse.ArgumentParser method), 465
exit() (module os, 669
exit() (module sys), 1217
exitcode (multiprocessing.Process attribute), 599
exitfunc (2to3 fixer), 1171
exitonclick() (module turtle), 1025
ExitStack (class in contextlib), 1238
exp() (decimal.Context method), 226
exp() (decimal.Decimal method), 219
exp() (module cmath), 211
exp() (module math), 207
expand() (re.match method), 91
expand_tabs (textwrap.TextWrapper attribute), 109
ExpandEnvironmentStrings() (module winreg), 1339
expandNode() (xml.dom.pulldom.DOMEventStream method), 825
expandtabs() (str method), 39
expanduser() (module os.path), 270
expandvars() (module os.path), 270
Expt, 836
ExpatError, 836
expect() (telnetlib.Telnet method), 917
expectedFailure() (module unittest), 1106
expectedFailures (unittest.TestCase method), 1116
expires (http.cookiejar.Cookie attribute), 942
exploded (ipaddress.IPv4Address attribute), 955
exploded (ipaddress.IPv4Network attribute), 959
explored (ipaddress.IPv6Address attribute), 956
exploded (ipaddress.IPv6Network attribute), 961
expm1() (module math), 207
exp() (module random), 241
expr() (module parser), 1298
expression, 1367
expunge() (imaplib.IMAP4 method), 900
extend() (array.array method), 188
extend() (collections.deque method), 171
extend() (sequence method), 34
extend() (xml.etree.ElementTree.Element method), 806
extend_path() (module pkgutil), 1280
EXTENDED_ARG (opcode), 1324
ExtendedContext (class in decimal), 223
ExtendedInterpolation (class in configparser), 364
extendleft() (collections.deque method), 171
extension module, 1367
EXTENSION_SUFFIXES (module importlib.machinery), 1292
ExtensionFileLoader (class in importlib.machinery), 1294
extensions_map (module http.server.SimpleHTTPRequestHandler attribute), 931
External Data Representation, 294, 377
external_attr (module zipfile.ZipInfo attribute), 345
ExternalExchangeError, 783
ExternalEntityParserCreate() (xml.parsers.expat.xmlparser method), 838
ExternalEntityRefHandler() (xml.parsers.expat.xmlparser method), 841
extra (module zipfile.ZipInfo attribute), 345
extract() (tarfile.TarFile method), 349
extract() (zipfile.ZipFile method), 343
extract_cookies() (module http.cookiejar.CookieJar method), 937
extract_stack() (module traceback), 1249
extract_tlb() (module traceback), 1249
extract_version (module zipfile.ZipInfo attribute), 345
extractall() (tarfile.TarFile method), 349
extractall() (zipfile.ZipFile method), 343
ExtractError, 347
extractfile() (tarfile.TarFile method), 349
extsep (module os), 420
F
f_contiguous (module memoryview attribute), 54
F_LOCK (module os), 395
F_OK (module os), 400
F_TEST (module os), 395
F_TLOCK (module os), 395
F_ULOCK (module os), 395
fabs() (module math), 206
factorial() (module math), 206
fail() (unittest.TestCase method), 1112
failfast (unittest.TestCase method), 1116
failureException (unittest.TestCase method), 1112
failures (unittest.TestCase method), 1116
False, 25, 62
false, 25
False (Built-in object), 25
False (built-in variable), 23
family (module socket.socket attribute), 684
FancyURLopener (class in urllib.request), 876
fast (pickle.Pickler attribute), 296
fatalError() (xml.sax.handler.ErrorHandler method), 831
faultCode (xmlrpc.client.Fault attribute), 947
fauhtandler (module), 1193
faultString (xmlrpc.client.Fault attribute), 947
fchdir() (module os), 401
Index 1421
fchmod() (in module os), 394
fchown() (in module os), 394
FCICreate() (in module msilib), 1331
fcntl (module), 1354
fcntl() (in module fcntl), 1354
fd() (in module turtle), 1003
fdatasync() (in module os), 394
fdopen() (in module os), 393
Feature (class in msilib), 1334
feature_external_ges (in module xml.sax.handler), 828
feature_external_pes (in module xml.sax.handler), 828
feature_namespace_prefixes (in module xml.sax.handler), 828
feature_namespaces (in module xml.sax.handler), 828
feature_string_interning (in module xml.sax.handler), 828
feature_validation (in module xml.sax.handler), 828
feed() (email.parser.FeedParser method), 726
feed() (html.parser.HTMLParser method), 794
feed() (xml.etree.ElementTree.XMLParser method), 809
feed() (xml.sax.xmlreader.IncrementalParser method), 834
FeedParser (class in email.parser), 726
fetch() (imaplib.IMAP4 method), 900
Fetch() (msilib.View method), 1332
fetchall() (sqlite3.Cursor method), 322
fetchmany() (sqlite3.Cursor method), 321
fetchone() (sqlite3.Cursor method), 321
fflags (select.kevent attribute), 667
field_size_limit() (in module csv), 356
fieldnames (csv.csvreader attribute), 359
fields (uuid.UUID attribute), 919
fileno (class in asynchat), 709
file
  .ini, 360
  .pdbrc, 1197
byte-code, 1275, 1314
copying, 285
debugger configuration, 1197
global files, 1347
mime.types, 785
path configuration, 1266
plist, 380
temporary, 280
file (pyclbr.Class attribute), 1314
file (pyclbr.Function attribute), 1314
file control
  UNIX, 1354
file name
temporary, 280
file object, 1367
  io module, 421
  open() built-in function, 14
file-like object, 1367
file_dispatcher (class in asyncore), 706
file_open() (urllib.request.FileHandler method), 872
file_size (zipfile.ZipInfo attribute), 346
file_wrapper (class in asyncio), 706
filecmp (module), 278
fileConfig() (in module logging.config), 506
FileCookieJar (class in http.cookiejar), 936
FileEntry (class in tkinter.tix), 1068
FileExistsError, 70
FileFinder (class in importlib.machinery), 1293
FileHandler (class in logging), 515
FileHandler (class in urllib.request), 866
FileInput (class in fileinput), 273
fileinput (module), 272
FileIO (class in io), 426
filelenino() (in module fileinput), 273
FileLoader (class in importlib.module), 1289
filenode() (in module stat), 275
filename (doctest.DocTest attribute), 1092
filename (http.cookiejar.FileCookieJar attribute), 938
filename (zipfile.ZipInfo attribute), 345names (in module fileinput), 273
filename_only (in module tabnanny), 1313
filenames
  pathname expansion, 283
  wildcard expansion, 283
filen() (http.client.HTTPResponse method), 889
fileno() (in module fileinput), 273
filenode() (io.IOBase method), 424
fileno() (multiprocessing.Connection method), 603
fileno() (ossaudiodev.oss_audio_device method), 978
fileno() (ossaudiodev.oss_mixer_device method), 980
fileno() (select.epoll method), 666
fileno() (select.kqueue method), 667
fileno() (socket.socket method), 680
fileno() (socketserver.BaseServer method), 922
fileno() (telnetlib.Telnet method), 917
FileNotFound, 70
FileSelectBox (class in tkinter.tix), 1068
FileType (class in argparser), 461
FileWrapper (class in wsgiref.util), 855
fill() (in module textwrap), 108
fill() (textwrap.TextWrapper method), 110
filcolor() (in module turtle), 1012
filling() (in module turtle), 1013
filter (2to3 fixer), 1171
Filter (class in logging), 497
filter (select.kevent attribute), 667
filter() (built-in function), 10
filter() (in module curses), 526
filter() (in module fnmatch), 284
filter() (logging.Filter method), 497
filter() (logging.Handler method), 495
filter() (logging.Logger method), 494
FILTER_DIR (in module unittest.mock), 1147
filterfalse() (in module itertools), 250
filterwarnings() (in module warnings), 1235
find() (doctest.DocTestFinder method), 1093
find() (in module getext), 984
find() (in module mmap), 717
fold_binary() (email.policy.Compat32 method), 734
fold_binary() (email.policy.EmailPolicy method), 735
fold_binary() (email.policy.Policy method), 734
FOR_ITER (opcode), 1323
forget() (in module test.support), 1176
forget() (tkinter.ttk.Notebook method), 1056
fork() (in module os), 413
fork() (in module pty), 1353
forkpty() (in module os), 414
Form (class in tkinter.tix), 1069
format (memoryview attribute), 53
format (struct.Struct attribute), 121
format() (built-in function), 11
format() (in module locale), 994
format() (logging.Formatter method), 496
format() (logging.Handler method), 496
format() (pprint.PrettyPrinter method), 198
format() (str method), 39
format() (string.Formatter method), 75
format_datetime() (in module email.utils), 750
format_exc() (in module traceback), 1249
format_exception() (in module traceback), 1249
format_exception_only() (in module traceback), 1249
format_field() (string.Formatter method), 75
format_help() (argparse.ArgumentParser method), 464
format_list() (in module traceback), 1249
format_map() (str method), 39
format_stack() (in module traceback), 1249
format_string() (in module locale), 995
format_time() (logging.Formatter method), 496
formatting, string (%), 43
formatting, string (with conversion flags), 43
formatting, string (without conversion flags), 43
formataddr() (in module email.utils), 749
formatargspec() (in module inspect), 1262
formatargvalues() (in module inspect), 1262
formatdate() (in module email.utils), 750
FormatError, 783
FormatError() (in module ctypes), 578
formatException() (logging.Formatter method), 497
formatmonth() (calendar.HTMLCalendar method), 164
formatmonth() (calendar.TextCalendar method), 163
formatStack() (logging.Formatter method), 497
Formatter (class in logging), 496
Formatter (class in string), 74
formatter (module), 1327
formatTime() (logging.Formatter method), 496
formatting, string (%), 43
formatwarning() (in module warnings), 1235
formatyear() (calendar.HTMLCalendar method), 164
formatyear() (calendar.TextCalendar method), 163
formatyearpage() (calendar.HTMLCalendar method), 164
forward() (in module turtle), 1003
found_terminator() (asynchat.async_chat method), 709
fpathconf() (in module os), 1394
fpectl (module), 1268
fqdn (smtpd.SMTPChannel attribute), 915
Fraction (class in fractions), 237
fractions (module), 237
FrameType (in module types), 194
freeze_support() (in module multiprocessing), 603
dexp() (module math), 206
from_address() (ctypes._CData method), 577
from_buffer() (ctypes._CData method), 577
from_buffer_copy() (ctypes._CData method), 577
from_bytes() (int class method), 29
from_decimal() (fractions.Fraction method), 238
from_float() (decimal.Decimal method), 219
from_floating_point() (fractions.Fraction method), 238
from_iterable() (itertools.chain class method), 247
from_param() (ctypes._CData method), 577
forkbuf() (tarfile.TarInfo method), 350
frombytes() (array.array method), 188
fromfd() (in module socket), 678
fromfd() (select.epoll method), 666
fromfd() (select.kqueue method), 667
fromfile() (array.array method), 188
fromhex() (bytearray class method), 47
fromhex() (bytes class method), 47
fromhex() (float class method), 30
fromkeys() (collections.Counter method), 169
fromkeys() (dict class method), 58
fromlist() (array.array method), 188
fromlist() (array.array method), 188
fromordinal() (datetime.date class method), 141
fromordinal() (datetime.datetime class method), 145
fromshare() (in module socket), 680
fromstring() (array.array method), 188
fromstring() (in module xml.etree.ElementTree), 804
fromstringlist() (in module xml.etree.ElementTree), 804
fromtarfile() (tarfile.TarInfo method), 350
fromtimestamp() (datetime.date class method), 141
fromtimestamp() (datetime.datetime class method), 145
fromunichr() (array.array method), 189
fromutc() (datetime.timezone method), 159
fromutc() (datetime.timezone method), 159
FrozenImporter (class in importlib.machinery), 1292
frozenset (built-in class), 54
fsdecode() (in module os), 389
fsencode() (in module os), 388
fstat() (in module os), 394
fstatvfs() (in module os), 394
fsum() (in module math), 206
fsync() (in module os), 394
FTP, 877
ftpbin (standard module), 891
protocol, 877, 891
FTP (class in ftplib), 891
ftp_open() (urllib.request.FTPHandler method), 872
FTP_TLS (class in ftplib), 892
FTPHandler (class in urllib.request), 866
ftplib (module), 891
ftpmirror.py, 893
ftruncate() (in module os), 394
get_errno() (in module ctypes), 576
get_examples() (doctest.DocTestParser method), 1094
get_exec_path() (in module os), 389
get_field() (mailbox.Formatter method), 74
get_field() (mailbox.Babyl method), 775
get_field() (mailbox.Mailbox method), 770
get_field() (mailbox.Maildir method), 773
get_field() (mailbox mbox method), 773
get_field() (mailbox.MH method), 774
get_field() (mailbox.MMDM method), 776
get_file_breaks() (bdb.Bdb method), 1192
get_filename() (email.message.Message method), 724
get_filename() (importlib.abc.ExecutionLoader method), 1289
get_filename() (importlib.abc.FileLoader method), 1289
get_filename() (importlib.abc.PyLoader method), 1291
get_filename() (importlib.abc.PyPycLoader method), 1291
get_filename() (zipimport.zipimporter method), 1280
get_flags() (mailbox.MaildirMessage method), 777
get_flags() (mailbox.mboxMessage method), 778
get_flags() (mailbox.MMDFMessage method), 782
get_folder() (mailbox.Maildir method), 772
get_folder() (mailbox.MH method), 774
get_free() (mailbox.mboxMessage method), 782
get_free() (mailbox.MMDFMessage method), 782
get_full_url() (urllib.request.Request method), 867
get_globals() (symtable.Function method), 1306
get_grouped_opcodes() (difflib.SequenceMatcher method), 104
get_header() (urllib.request.Request method), 867
get_history_item() (in module readline), 113
get_history_length() (in module readline), 114
get_host() (urllib.request.Request method), 867
get_id() (symtable.SymbolTable method), 1306
get_ident() (in module _thread), 669
get_ident() (in module threading), 583
get_identifiers() (symtable.SymbolTable method), 1306
get_importer() (in module pkgutil), 1281
get_initials() (mailbox.MaildirMessage method), 777
get_initials() (mailbox.mboxMessage method), 782
get_initials() (mailbox.MMDFMessage method), 782
get_label() (mailbox.Maildir method), 777
get_label() (mailbox.MH method), 774
get_labels() (mailbox.MaildirMessage method), 775
get_labels() (mailbox.mboxMessage method), 778
get_labels() (mailbox.MMDFMessage method), 781
get_last_error() (in module ctypes), 576
get_line_buffer() (in module readline), 113
get_lineno() (symtable.SymbolTable method), 1306
get_magic() (in module imp), 1275
get_method() (urllib.request.Request method), 867
get_methods() (symtable.Class method), 1306
get_mixed_type_key() (in module ipaddress), 964
get_name() (symtable.Symbol method), 1307
get_name() (symtable.SymbolTable method), 1306
get_namespace() (symtable.Symbol method), 1307
get_namespaces() (symtable.Symbol method), 1307
get_nonstandard_attr() (http.cookiejar.Cookie method), 942
get_nowait() (multiprocessing.Queue method), 601
get_nowait() (queue.Queue method), 662
get_objects() (in module gc), 1253
get_opcodes() (difflib.SequenceMatcher method), 103
get_option() (optparse.OptionParser method), 481
get_option_group() (optparse.OptionParser method), 473
get_origin_req_host() (urllib.request.Request method), 868
get_oshandle() (in module msvcrt), 1337
get_output_charset() (email.charset.Charset method), 746
get_path() (in module sysconfig), 1228
get_path_names() (in module sysconfig), 1228
get_paths() (in module sysconfig), 1229
get_platform() (in module sysconfig), 1229
get_poly() (in module turtle), 1018
get_position() (xdrlib.Unpacker method), 378
get_python_version() (in module sysconfig), 1229
get_recsrc() (ossaudiodev.oss_mixer_device method), 981
get_ref() (in module gc), 1254
get_refers() (in module gc), 1253
get_request() (socketserver.BaseServer method), 923
get_scheme() (wsgiref.handlers.BaseHandler method), 860
get_scheme_names() (in module sysconfig), 1228
get_selector() (urllib.request.Request method), 867
get_sequences() (mailbox.MHMessage method), 780
get_sequences() (mailbox.MH method), 780
get_server() (multiprocessing.managers.BaseManager method), 608
get_server_certificate() (in module ssl), 692
get_shapepoly() (in module turtle), 1017
get_socket() (telnetlib.Telnet method), 917
get_source() (importlib.abc.InspectLoader method), 1289
get_source() (importlib.abc.PyLoader method), 1291
get_source() (importlib.abc.SourceLoader method), 1290
get_source() (importlib.machinery.ExtensionFileLoader method), 1294
get_source() (importlib.machinery.SourcelessFileLoader method), 1294
get_source() (importlib.machinery.SourcelessFileLoader method), 1294
get_stack() (bdb.Bdb method), 1192
get_starttag_text() (html.parser.HTMLParser method), 795
get_stderr() (wsgiref.handlers.BaseHandler method), 860
get_stderr() (wsgiref.simple_server.WSGIRequestHandler method), 858
get.stdin() (wsgiref.handlers.BaseHandler method), 860
get_string() (mailbox.Mailbox method), 770
get_subdir() (mailbox.MaildirMessage method), 777
get_suffixes() (in module imp), 1275
get_symbols() (symtable.SymbolTable method), 1306
get_tag() (in module imp), 1277
get_terminal_size() (in module os), 398
get_terminal_size() (in module shutil), 291
get_geterminator() (asyncchat.async_chat method), 709
get_threshold() (in module gc), 1253
get_type() (symtable.SymbolTable method), 1306
get_type() (urllib.request.Request method), 867
get_unixfrom() (email.message.Message method), 720
get_unpack_formats() (in module shutil), 290
get_usage() (optparse.OptionParser method), 483
get_value() (string.Formatter method), 74
get_version() (optparse.OptionParser method), 473
get_visible() (mailbox.BabylMessage method), 781
get_wch() (curses.window method), 533
getacl() (imaplib.IMAP4 method), 900
getaddresses() (in module email.utils), 749
getaddrinfo() (in module socket), 676
get_annotation() (imaplib.IMAP4 method), 900
getacl() (in module email.utils), 749
getaddrinfo() (in module socket), 676
get_annotation() (imaplib.IMAP4 method), 900
getargspec() (in module inspect), 1262
getargvalues() (in module inspect), 1263
getatime() (in module os.path), 270
getbgcolor() (curses.window method), 533
getcomptype() (aifc.aifc method), 968
getcomptype() (sunau.AU_read method), 971
getcomptype() (wave.Wave_read method), 973
getcontext() (in module decimal), 1253
getcwd() (in module os), 401
getcode() (in module msvcrt), 1337
get_event() (xml.dom.pulldom.DOMEventStream method), 1337
getEventCategory() (logging.handlers.NTEventLogHandler method), 521
getEventCategory() (logging.handlers.NTEventLogHandler method), 521
getException() (xml.sax.SAXException method), 827
getFeature() (xml.sax.xmlreader.XMLReader method), 834
GetFieldCount() (msilib.Record method), 1333
getfile() (in module inspect), 1258
getfilesystemencoding() (in module sys), 1219
getfirst() (cgi.FieldStorage method), 850
getfloat() (configparser.ConfigParser method), 374
getfmts() (ossaudiodev.oss_audio_device method), 979
getfqdn() (in module socket), 677
getframeinfo() (in module inspect), 1264
getframerate() (aifc.aifc method), 968
getframerate() (sunau.AU_read method), 971
getframerate() (wave.Wave_read method), 973
getfullargspec() (in module inspect), 1262
getgeneratorlocals() (in module inspect), 1265
getgid() (in module os), 389
getgrall() (in module grp), 1349
getgrnam() (in module grp), 1350
groupe() (in module os), 389
groups() (in module os), 389
getheader() (http.client.HTTPResponse method), 889
getheaders() (http.client.HTTPResponse method), 889
gethostbyaddr() (in module socket), 392, 677
gethostbyname() (in module socket), 677
gethostbyname() (in module socket), 677
gethostname() (in module socket), 392, 677
getincrementaldecoder() (in module codecs), 123
getincrementalencoder() (in module codecs), 123
getinfo() (zipfile.ZipFile method), 342
getinnerframes() (in module inspect), 1264
GetInputContext() (xml.parsers.expat.xmlparser method), 838
getint() (configparser.ConfigParser method), 374
GetInteger() (msilib.Record method), 1333
ggetitem() (in module operator), 263
getiterator() (xml.etree.ElementTree.Element method), 807
getiterator() (xml.etree.ElementTree.ElementTree method), 808
getitimer() (in module signal), 713
getkey() (curses.window method), 533
GetLastError() (in module ctypes), 576
getLength() (xml.sax.xmlreader.Attributes method), 836
getLevelName() (in module logging), 503
getime() (in module linecache), 284
geline() (xml.sax.xmlreader.Locator method), 835
gelist() (cgi.FieldStorage method), 850
getlookavg() (in module os), 420
getlocal() (in module locale), 994
getLogger() (in module logging), 501
getLoggerClass() (in module logging), 501
getloginfo() (in module os), 389
getLogRecordFactory() (in module logging), 501
getmark() (aifc.aifc method), 969
getmark() (sunau.AU_read method), 972
getmark() (wave.Wave_read method), 973
getmarkers() (aifc.aifc method), 969
getmarkers() (sunau.AU_read method), 971
getmarkers() (wave.Wave_read method), 973
getmaxyx() (curses.window method), 533
getmember() (tarfile.TarFile method), 348
getmembers() (in module inspect), 1256
getmembers() (tarfile.TarFile method), 348
getMessage() (logging.LogRecord method), 498
getMessage() (xml.sax.SAXException method), 827
getmessageID() (logging.handlers.NTEventLogHandler method), 521
getmodule() (in module inspect), 1258
getmoduleinfo() (in module inspect), 1256
getmoduleuname() (in module inspect), 1257
getmouse() (in module curses), 527
getmut() (in module inspect), 1262
getmtime() (in module os.path), 270
getname() (chunk.Chunk method), 975
getName() (threading.Thread method), 586
getNameByQName() (xml.sax.xmlreader.AttributesNS method), 836
getnameinfo() (in module socket), 677
getnames() (tarfile.TarFile method), 349
getNames() (xml.sax.xmlreader.Attributes method), 836
getnchannels() (aifc.aifc method), 968
getnchannels() (sunau.AU_read method), 971
genchannels() (wave.Wave_read method), 973
getnframes() (aifc.aifc method), 968
getnframes() (sunau.AU_read method), 971
genframes() (wave.Wave_read method), 973
genode(), 919
getnode() (in module uuid), 919
getopt() (in module getopt), 489
getopt() (in module getopt), 489
GetoptError, 490
getouterframes() (in module inspect), 1264
getoutput() (in module subprocess), 659
getpagesize() (in module resource), 1359
getparams() (aifc.aifc method), 969
getparams() (sunau.AU_read method), 971
getparams() (wave.Wave_read method), 973
getparyx() (curses.window method), 533
getpass() (in module getpass), 524
getppid() (in module os), 390
getpreferredencoding() (in module locale), 994
getpriority() (in module os), 390
getprofile() (in module sys), 1220
GetProperty() (msilib.SummaryInformation method), 1333
getProperty() (xml.sax.xmlreader.XMLReader method), 834
GetPropertyCount() (msilib.SummaryInformation method), 1333
getproposedname() (in module socket), 677
getproxies() (in module urllib.request), 864
getPublicId() (xml.sax.xmlreader.InputSource method), 835
getPublicId() (xml.sax.xmlreader.Locator method), 835
getQNameByName() (xml.sax.xmlreader.AttributesNS method), 836
getQNames() (xml.sax.xmlreader.AttributesNS method), 836
getquota() (imaplib.IMAP4 method), 900
getquotaroot() (imaplib.IMAP4 method), 900
getrandbits() (in module random), 240
getreader() (in module codecs), 123
getrecursionlimit() (in module sys), 1219
getrefcount() (in module sys), 1219
getresgid() (in module os), 390
getresponse() (http.client.HTTPConnection method), 888
getresuid() (in module os), 390
getrlimit() (in module resource), 1358
getroot() (xml.etree.ElementTree.ElementTree method), 808
getrusage() (in module resource), 1359
getsample() (in module audioop), 966
getsampwidth() (aifc.aifc method), 968
getsampwidth() (sunau.AU_read method), 971
getsampwidth() (wave.Wave_read method), 973
getscreen() (in module turtle), 1018
getsevbyname() (in module socket), 677
getsevbynameport() (in module socket), 678
GetSetDescriptorType (in module types), 194
getshapes() (in module turtle), 1024
getsid() (in module os), 392
getsignal() (in module signal), 712
getsitepackages() (in module site), 1267
gettempdir() (in module tempfile), 282
gettempnam() (in module tempfile), 282
gettempdir() (xml.sax.xmlreader.InputSource method), 835
getSystemId() (xml.sax.xmlreader.InputSource method), 835
getsys() (in module curses), 527
getterminal() (tarfile.TarFile method), 349
gettimedir() (in module tempdir), 282
getTestCases() (unittest.TestLoader method), 1115
ggettext (module), 983
ggettext() (gettext.GNUTranslations method), 987
ggettext() (gettext.NullTranslations method), 986
ggettext() (in module gettext), 984
gettimeout() (socket.socket method), 681
gettrace() (in module sys), 1220
gettuplesize() (in module sys), 1219
gettdirs() (in module sys), 1219
getunicode() (in module sys), 1219
gettext() (xml.sax.xmlreader.InputSource method), 835
getvalue() (io.BytesIO method), 427
getvalue() (io.StringIO method), 430
getTestCaseNames() (unittest.TestLoader method), 1115
glob (module), 283
glob() (in module glob), 283
glob() (msilib.Directory method), 1334
global interpreter lock, 1368
The Python Library Reference, Release 3.3.3

globals() (built-in function), 11
globs (doctest.DocTest attribute), 1092
gmtime() (in module time), 434
gname (tarfile.TarInfo attribute), 350
GNOME, 987
GNU_FORMAT (in module tarfile), 347
gnu_getopt() (in module getopt), 490
got (doctest.DocTestFailure attribute), 1098
goto() (in module turtle), 1004
Graphical User Interface, 1041
GREATER (in module token), 1308
GREATEREQUAL (in module token), 1308
Greenwich Mean Time, 432
Group (class in email.headerregistry), 741
group() (ntplib.NNTP method), 906
group() (re.match method), 91
groupby() (in module itertools), 250
groupdict() (re.match method), 92

groupindex (re.regex attribute), 91
groups (email.headerregistry.AddressHeader attribute), 738
groups (re.regex attribute), 91
groups() (re.match method), 92
grp (module), 1349
gt() (in module operator), 261
guess_all_extensions() (in module mimetypes), 785
guess_all_extensions() (mimetypes.MimeTypes method), 786
guess_extension() (in module mimetypes), 785
guess_extension() (mimetypes.MimeTypes method), 786
guess_scheme() (in module wsgiref.util), 854
guess_type() (in module mimetypes), 784
guess_type() (mimetypes.MimeTypes method), 786
GUI, 1041
gzip (module), 332
GzipFile (class in gzip), 332

H

halfdelay() (in module curses), 527
handle() (http.server.BaseHTTPRequestHandler method), 929
handle() (logging.Handler method), 496
handle() (logging.handlers.QueueListener method), 524
handle() (logging.Logger method), 495
handle() (logging.NullHandler method), 515
handle() (socketserver.RequestHandler method), 924
handle() (wsgiref.simple_server.WSGIRequestHandler method), 858
handle_accept() (asyncore.dispatcher method), 705
handle_accepted() (asyncore.dispatcher method), 705
handle_charref() (html.parser.HTMLParser method), 795
handle_close() (asyncore.dispatcher method), 705
handle_comment() (html.parser.HTMLParser method), 795
handle_connect() (asyncore.dispatcher method), 705
handle_data() (html.parser.HTMLParser method), 795
handle_decl() (html.parser.HTMLParser method), 795
handle_defect() (email.policy.Policy method), 733
handle_endtag() (html.parser.HTMLParser method), 795
handle_entityref() (html.parser.HTMLParser method), 795
handle_error() (asyncore.dispatcher method), 705
handle_error() (socketserver.ProxyServer method), 923
handle_expect_100() (http.server.BaseHTTPRequestHandler method), 929
handle_expt() (asyncore.dispatcher method), 705
handle_one_request() (http.server.BaseHTTPRequestHandler method), 929
handle_pgi() (html.parser.HTMLParser method), 796
handle_read() (asyncore.dispatcher method), 705
handle_request() (socketserver.BaseServer method), 922
handle_request() (xmlrpc.server.CGIXMLRPCRequestHandler method), 952
handle_startelement() (html.parser.HTMLParser method), 795
handle_starttag() (html.parser.HTMLParser method), 795
handle_timeout() (socketserver.BaseServer method), 923
handle_write() (asyncore.dispatcher method), 705
handleError() (logging.Handler method), 496
handleError() (logging.handlers.SocketHandler method), 518
handler() (in module cgi), 853
has_children() (symtable.SymbolTable method), 1306
has_colors() (in module curses), 527
has_data() (urllib.request.Request method), 867
HAS_ECDH (in module ssl), 693
has_exec() (symtable.SymbolTable method), 1306
has_extn() (smtplib.SMTP method), 911
has_header() (csv.Sniffer method), 357
has_header() (urllib.request.Request method), 867
has_ic() (in module curses), 527
has_il() (in module curses), 527
has_import_star() (sys.modules.SymbolTable method), 1306
has_ipv6 (in module socket), 676
has_key (2to3 fixer), 1171
has_key() (in module curses), 527
has_nonstandard_attr() (http.cookiejar.Cookie method), 942
HAS_NPN (in module ssl), 693
has_option() (configparser.ConfigParser method), 373
has_option() (optparse.OptionParser method), 482
has_section() (configparser.ConfigParser method), 373
HAS_SNI (in module ssl), 693
hasattr() (built-in function), 11
hasAttribute() (xml.dom.Element method), 816
hasAttributeNS() (xml.dom.Element method), 816
hasAttributes() (xml.dom.Node method), 813
hasChildNodes() (xml.dom.Node method), 814
hascompare (in module dis), 1318
hasconst (in module dis), 1318
hasFeature() (xml.dom.DOMImplementation method), 812
hasfree (in module dis), 1318
hash
  built-in function, 34
hash() (built-in function), 11
hash.block_size (in module hashlib), 384
hash.digest_size (in module hashlib), 384
hash_info (in module sys), 1220
hashable, 1368
Hashable (class in collections.abc), 180
hasHandlers() (logging.Logger method), 495
hashlib (module), 383
hasjabs (in module dis), 1318
hasjrel (in module dis), 1318
haslocal (in module dis), 1318
hasname (in module dis), 1318
HAVE_ARGUMENT (opcode), 1324
HAVE_THREADS (in module decimal), 228
head() (nntplib.NNTP method), 907
Header (class in email.header), 744
header_encode() (email.charset.Charset method), 746
header_encode_lines() (email.charset.Charset method), 746
header_encoding (email.charset.Charset attribute), 746
header_factory (email.policy.EmailPolicy attribute), 735
header_fetch_parse() (email.policy.Compat32 method), 734
header_fetch_parse() (email.policy.EmailPolicy method), 735
header_fetch_parser() (email.policy.EmailPolicy method), 733
header_items() (urllib.request.Request method), 867
header_max_count() (email.policy.EmailPolicy method), 735
header_max_count() (email.policy.Policy method), 733
header_offset (zipfile.ZipInfo attribute), 345
header_source_parse() (email.policy.Compat32 method), 734
header_source_parse() (email.policy.EmailPolicy method), 735
header_source_parse() (email.policy.Policy method), 733
header_store_parse() (email.policy.Compat32 method), 734
header_store_parse() (email.policy.EmailPolicy method), 735
header_store_parse() (email.policy.Policy method), 733
HeaderError, 347
HeaderParseError, 748
HeaderRegistry (class in email.headerregistry), 739
headers
  (class in email.mime.multipart), 696
  (class in email.mime.text), 750
headers (http.server.BaseHTTPRequestHandler attribute), 928
headers (xmlrpc.client.ProtocolError attribute), 948
heading() (in module turtle), 1009
heading() (tkinter.ttk.Treeview method), 1061
heapify() (in module heapq), 182
heappop() (in module heapq), 182
heapreplace() (in module heapq), 182
heappush() (in module heapq), 182
heappushpop() (in module heapq), 182
heapq (module), 181
heareplace() (in module heapq), 182
helot() (smtplib.SMTP method), 911
help
  online, 1077
help (optparse.Option attribute), 478
help (pdb command), 1197
help() (built-in function), 11
help() (ntplib.NNTP method), 907
heror, 674
hex (uuid.UUID attribute), 919
hex() (built-in function), 11
hex() (float method), 30
hexadecimal
  literals, 26
hexbin() (in module binhex), 789
hexdigest() (hashlib.hash method), 384
hexdigest() (hmac.HMAC method), 385
hexdigits (in module string), 73
hexify() (in module binascii), 791
hexversion (in module sys), 1221
hidden() (curses.panel.Panel method), 544
hide() (curses.panel.Panel method), 544
hide() (tkinter.ttk.Notebook method), 1056
hide_cookie2 (http.cookiejar.CookiePolicy attribute), 939
hideturtle() (in module turtle), 1014
HierarchyRequestErr, 818
HIGHEST_PROTOCOL (in module pickle), 294
HKEY_CLASSES_ROOT (in module winreg), 1342
HKEY_CURRENT_CONFIG (in module winreg), 1342
HKEY_CURRENT_USER (in module winreg), 1342
HKEY_DYN_DATA (in module winreg), 1342
HKEY_LOCAL_MACHINE (in module winreg), 1342
HKEY_PERFORMANCE_DATA (in module winreg), 1342
HKEY_USERS (in module winreg), 1342
hline() (curses.window method), 533
HList (class in tkinter.tix), 1068
hls_to_rgb() (in module colorsys), 976
hmac (module), 385
HOME, 270
home() (in module turtle), 1005
HOMEDRIVE, 270
HOMEPATH, 270
hook_compressed() (in module fileinput), 274
hook_encoded() (in module fileinput), 274
identify_region() (tkinter.ttk.Treeview method), 1061
identify_row() (tkinter.ttk.Treeview method), 1061
idioms (2to3 fixer), 1171
IDLE, 1071, 1368
IDLESTARTUP, 1074
idlok() (curses.window method), 533
IEEE-754, 1268
if statement, 25
if_nameindex() (in module socket), 680
if_nametoindex() (in module socket), 680
ifloordiv() (in module operator), 266
iglob() (in module glob), 283
ignorableWhitespace() (xml.sax.handler.ContentHandler method), 830
ignore (pdb command), 1198
ignore_errors() (in module codecs), 123
IGNORE_EXCEPTION_DETAIL (in module doctest), 1085
ignore_patterns() (in module shutil), 286
IGNORECASE (in module re), 87
imath() (in module operator), 266
imag (numbers.Complex attribute), 203
imap() (multiprocessing.pool.Pool method), 615
IMAP4
  protocol, 897
IMAP4 (class in imaplib), 898
IMAP4.abort, 898
IMAP4.error, 898
IMAP4.readonly, 898
IMAP4_SSL
  protocol, 897
IMAP4_SSL (class in imaplib), 898
IMAP4_stream
  protocol, 897
IMAP4_stream (class in imaplib), 898
imap_unordered() (multiprocessing.pool.Pool method), 615
imaplib (module), 897
imghdr (module), 976
immedok() (curses.window method), 534
immutable, 1368
  sequence types, 34
imod() (in module operator), 266
imp
  module, 21
  imp (module), 1275
ImplImport (class in pkgutil), 1281
implementation (in module sys), 1221
ImplLoader (class in pkgutil), 1281
import
  statement, 21, 1275
import (2to3 fixer), 1171
Import module, 1072
import path, 1368
import_fresh_module() (in module test.support), 1179
IMPORT_FROM (opcode), 1322
import_module() (in module importlib), 1286
import_module() (in module test.support), 1178
IMPORT_NAME (opcode), 1322
IMPORT_STAR (opcode), 1321
importer, 1368
ImportError, 66
importing, 1368
importlib (module), 1286
importlib.abc (module), 1287
importlib.machinery (module), 1292
importlib.util (module), 1294
imports (2to3 fixer), 1171
imports2 (2to3 fixer), 1171
ImportWarning, 71
ImproperConnectionState, 886
in_transaction (sqlite3.Connection attribute), 315
inch() (curses.window method), 534
Incomplete, 791
IncompleteRead, 886
increment_lineno() (in module ast), 1304
IncrementalDecoder (class in codecs), 126
IncrementalEncoder (class in codecs), 126
IncrementalNewlineDecoder (class in io), 431
IncrementalParser (class in xml.sax.xmlreader), 833
indent (doctest.Example attribute), 1093
INDENT (in module token), 1308
indent() (in module textwrap), 108
indentation, 1073
IndentationError, 68
index() (array.array method), 189
index() (in module operator), 262
index() (sequence method), 32
index() (str method), 40
index() (tkinter.ttk.Notebook method), 1056
index() (tkinter.ttk.Treeview method), 1061
IndexError, 66
IndexOf() (in module operator), 263
IndexSizeErr, 818
inet_atom() (in module socket), 678
inet_ntoa() (in module socket), 678
inet_ntop() (in module socket), 679
Inexact (class in decimal), 229
infile (shlex.shlex attribute), 1038
Infinity, 10
info() (gettext.NullTranslations method), 986
info() (in module logging), 502
info() (logging.Logger method), 494
infolist() (zipfile.ZipFile method), 342
ini file, 360
init() (in module mimetypes), 785
init_color() (in module curses), 527
init_database() (in module msilib), 1331
init_pair() (in module curses), 527
initgroups() (in module os), 390
initial_indent (textwrap.TextWrapper attribute), 109
insch() (curses.window method), 534
install() (gettext.NullTranslations method), 986
install() (in module gettext), 985
install_opener() (in module urllib.request), 864
install_script() (venv.EnvBuilder method), 1183
installHandler() (in module unittest), 1120
instate() (tkinter.ttk.Widget method), 1054
instr() (curses.window method), 534
istream (shlex.shlex attribute), 1038
int
  built-in function, 27
int (uuid.UUID attribute), 919
int() (built-in function), 12
Int2AP() (in module imaplib), 898
int_info (in module sys), 1221
integer
  literals, 26
  object, 26
  types, operations on, 28
Integral (class in numbers), 204
Integrated Development Environment, 1071
Intel/DVI ADPCM, 965
interact (pdb command), 1199
interact() (code.InteractiveConsole method), 1272
interact() (in module code), 1271
interact() (telnetlib.Telnet method), 917
interactive, 1368
InteractiveConsole (class in code), 1271
InteractiveInterpreter (class in code), 1271
intern (2to3 fixer), 1171
intern() (in module sys), 1221
internal_attr (zipfile.ZipInfo attribute), 345
Internaldate2tuple() (in module imaplib), 898
internalSubset (xml.dom.DocumentType attribute), 815
Internet, 845
interpolation, string (%), 43
InterpolationDepthError, 376
InterpolationError, 376
InterpolationMissingOptionError, 376
InterpolationSyntaxError, 376
interpreted, 1368
interpreter prompts, 1223
interrupt() (sqlite3.Connection method), 317
interrupt_main() (in module _thread), 669
InterruptedError, 70
intersection() (set method), 55
intersection_update() (set method), 56
intro (cmd.Cmd attribute), 1033
InuseAttributeErr, 818
in() (in module operator), 262
InvalidAccessErr, 818
invalidate_caches() (importlib.abc.MetaPathFinder method), 1287
invalidate_caches() (importlib.abc.PathEntryFinder method), 1288
invalidate_caches() (importlib.machinery.FileFinder method), 1293
invalidate_caches() (importlib.machinery.PathFinder class method), 1293
invalidate_caches() (in module importlib), 1287
InvalidCharacterErr, 818
InvalidModificationErr, 818
InvalidOperation (class in decimal), 229
InvalidStateErr, 818
InvalidURL, 886
invert() (in module operator), 262
io (module), 421
io.StringIO object, 38
IOBase (class in io), 423
ioctl() (in module fcntl), 1355
ioctl() (socket.socket method), 681
IOError, 69
ior() (in module operator), 266
ip (ipaddress.IPv4Interface attribute), 962
ip (ipaddress.IPv6Interface attribute), 963
ip_address() (in module ipaddress), 954
ip_interface() (in module ipaddress), 954
ipaddress (module), 954
ipow() (in module operator), 267
ipv4_mapped (ipaddress.IPv6Address attribute), 957
IPv4Address (class in ipaddress), 955
IPv4Interface (class in ipaddress), 962
IPv4Network (class in ipaddress), 958
IPv6Address (class in ipaddress), 956
IPv6Interface (class in ipaddress), 963
IPv6Network (class in ipaddress), 960
irshift() (in module operator), 267
is operator, 26
is_not operator, 26
is() (in module operator), 262
is_alive() (multiprocessing.Process method), 598
is_alive() (threading.Thread method), 586
is_assigned() (symtable.Symbol method), 1307
is_blocked() (http.cookiejar.DefaultCookiePolicy method), 940
is_canonical() (decimal.Context method), 226
is_canonical() (decimal.Decimal method), 219
IS_CHARACTER_JUNK() (in module difflib), 102
is_check_supported() (in module lzma), 339
is_declared_global() (symtable.Symbol method), 1307
is_empty() (asynchat.fifo method), 709
is_enabled() (in module http.cookiejar), 1194
is_expired() (http.cookiejar.Cookie method), 942
is_finite() (decimal.Context method), 226
is_finite() (decimal.Decimal method), 219
is_free() (syntable.Symbol method), 1307
is_global() (syntable.Symbol method), 1307
is_hops_by_hop() (in module wsgiref.util), 855
is_imported() (syntable.Symbol method), 1307
is_infinite() (decimal.Context method), 226
is_infinite() (decimal.Decimal method), 220
is_integer() (float method), 29
is_jython (in module test.support), 1176
IS_LINE_JUNK() (in module difflib), 102
is_linetouched() (curses.window method), 534
is_link_local (ipaddress.IPv4Address attribute), 956
is_link_local (ipaddress.IPv4Network attribute), 959
is_link_local (ipaddress.IPv6Address attribute), 956
is_link_local (ipaddress.IPv6Network attribute), 961
is_local() (syntable.Symbol method), 1307
is_loopback (ipaddress.IPv4Address attribute), 956
is_loopback (ipaddress.IPv4Network attribute), 959
is_loopback (ipaddress.IPv6Address attribute), 956
is_loopback (ipaddress.IPv6Network attribute), 961
is_multicast (ipaddress.IPv4Address attribute), 955
is_multicast (ipaddress.IPv4Network attribute), 958
is_multicast (ipaddress.IPv6Address attribute), 956
is_multicast (ipaddress.IPv6Network attribute), 961
is_multipart() (email.message.Message method), 720
is_namespace() (syntable.Symbol method), 1307
is_n() (decimal.Context method), 226
is_n() (decimal.Decimal method), 220
is_nested() (syntable.SymbolTable method), 1306
is_normal() (decimal.Context method), 226
is_normal() (decimal.Decimal method), 220
is_not() (in module operator), 262
is_not_allowed() (http.cookiejar.DefaultCookiePolicy method), 940
is_optimized() (syntable.SymbolTable method), 1306
is_package() (importlib.abc.InspectLoader method), 1289
is_package() (importlib.abc.SourceLoader method), 1290
is_package() (importlib.machinery.ExtensionFileLoader method), 1294
is_package() (importlib.machinery.SourceFileLoader method), 1293
is_package() (importlib.machinery.SourcelessFileLoader method), 1294
is_package() (importlib.machinery.ZipImportPlugin method), 1280
is_paramter() (syntable.Symbol method), 1307
is_private (ipaddress.IPv4Address attribute), 955
is_private (ipaddress.IPv4Network attribute), 958
is_private (ipaddress.IPv6Address attribute), 956
is_private (ipaddress.IPv6Network attribute), 961
is_python_build() (in module sysconfig), 1230
is_qnan() (decimal.Context method), 226
is_qnan() (decimal.Decimal method), 220
is_referenced() (syntable.Symbol method), 1307
is_reserved (ipaddress.IPv6Address attribute), 956
is_reserved (ipaddress.IPv6Network attribute), 959
is_reserved (ipaddress.IPv6Network attribute), 956
is_reserved (ipaddress.IPv6Network attribute), 961
is_resource_enabled() (in module test.support), 1176
is_set() (threading.Event method), 591
is_signed() (decimal.Context method), 226
is_signed() (decimal.Decimal method), 220
is_site_local (ipaddress.IPv6Address attribute), 956
is_site_local (ipaddress.IPv6Network attribute), 961
is_snan() (decimal.Context method), 226
is_snan() (decimal.Decimal method), 220
is_subnormal() (decimal.Context method), 226
is_subnormal() (decimal.Decimal method), 220
is_term_resized() (in module curses), 347
is_term_resized() (in module curses), 527
is_tracked() (in module curses), 1254
is_unspecified (ipaddress.IPv4Address attribute), 956
is_unspecified (ipaddress.IPv4Network attribute), 958
is_unspecified (ipaddress.IPv6Address attribute), 961
is_unverified() (urllib.request.Request method), 868
is_wintouched() (curses.window method), 534
is_zero() (decimal.Context method), 226
is_zero() (decimal.Decimal method), 220
is_zipfile() (in module zipfile), 341
isabs() (in module os.path), 270
isabstract() (in module inspect), 1257
IsADirectoryError, 70
isalnum() (in module curses.ascii), 542
isalnum() (str method), 40
isalpha() (in module curses.ascii), 542
isalpha() (str method), 40
isascii() (in module curses.ascii), 542
isatty() (chunk.Chunk method), 975
isatty() (in module os), 394
isblk() (tarfile.TarInfo method), 351
isbuiltin() (in module inspect), 1257
ischr() (tarfile.TarInfo method), 351
isclass() (in module inspect), 1257
iscntrl() (in module curses.ascii), 542
iscode() (in module inspect), 1257
isctrl() (in module curses.ascii), 543
isDaemon() (threading.Thread method), 586
isdatadescriptor() (in module inspect), 1257
isdecimal() (str method), 40
isdev() (tarfile.TarInfo method), 351
isdir() (in module os.path), 271
isdir() (tarfile.TarInfo method), 351
isdisjoint() (set method), 55
isdown() (in module turtle), 1011
iselement() (in module xml.etree.ElementTree), 804
isenabled() (in module gc), 1253
isExposedFor() (logging.Logger method), 493
isfile() (in module tarfile), 347
ISEOF() (in module tokenize), 1308
isfinite() (in module cmath), 212
isfinite() (in module math), 207
isinstance (2to3 fixer), 1171
isinstance() (built-in function), 12
isidentifier() (str method), 40
isinf() (in module cmath), 212
isinf() (in module math), 207
instantiation (2to3 fixer), 493
isinstance() (built-in function), 12
iskeyword() (in module keyword), 12
isleap() (in module calendar), 164
islice() (in module itertools), 251
islk() (tarfile.TarInfo method), 351
islnk() (in module os.path), 271
islower() (in module os.path), 270
islower() (str method), 40
ismemberdescriptor() (in module inspect), 1258
ismeta() (in module curses.ascii), 542
ismethod() (in module inspect), 1257
ismethoddescriptor() (in module inspect), 1257
ismodule() (in module inspect), 1257
ismount() (in module os.path), 271
isnan() (in module cmath), 212
isnan() (in module math), 207
ISNONTERMINAL() (in module tokenize), 1308
isnumeric() (str method), 40
isocalendar() (datetime.date method), 143
isocalendar() (datetime.datetime method), 149
isformat() (datetime.date, 143
isformat() (datetime.datetime, 149
isformat() (datetime.time, 152
isolation_level (sqlite3.Connection attribute), 315
isoweekday() (datetime.date method), 149
isoweekday() (datetime.datetime method), 149
isprint() (in module curses.ascii), 542
isprintable() (str method), 40
ispunct() (in module curs.ascii), 542
issubclass() (built-in function), 12
issubset() (set method), 55
isspace() (in module os), 394
isspace() (str method), 40
issubset() (set method), 55
issym() (tarfile.TarInfo method), 351
isterminated() (threading.Thread method), 586
isTerminal() (in module tokenize), 1308
istitle() (str method), 40
isterminated() (threading.Thread method), 1011
isinstance() (built-in function), 12
isspace() (str method), 40
issubset() (set method), 55
isspace() (xml.etree.ElementTree, 804
isenabled() (in module gc), 1253
isExposedFor() (logging.Logger method), 493
isfile() (in module tarfile), 347
ISEOF() (in module tokenize), 1308
isfinite() (in module cmath), 212
isfinite() (in module math), 207
isinstance() (built-in function), 12
isspace() (str method), 40
issubset() (set method), 55
issym() (tarfile.TarInfo method), 351
ISTERMINAL() (in module tokenize), 1308
isSameNode() (xml.dom.Node method), 814
isspace() (module curses.ascii), 542
isspace() (str method), 40
issubclass() (built-in function), 12
issubset() (set method), 55
issuite() (in module parser), 1299
issuite() (parser.ST method), 1300
isfile() (in module os.path), 270
isfile() (tarfile.TarInfo method), 351
isfinite() (in module cmath), 212
isfinite() (in module math), 207
isfirstline() (in module fileinput), 273
isframe() (in module inspect), 1257
isfunction() (module inspect), 1257
isgenerator() (module inspect), 1257
isgeneratorfunction() (module inspect), 1257
isgetsetdescriptor() (module inspect), 1258
isgraph() (module curses.ascii), 542
isidentifier() (str method), 40
isinstance() (built-in function), 12
iskeyword() (in module keyword), 12
isleap() (in module calendar), 164
islice() (in module itertools), 251
islk() (tarfile.TarInfo method), 351
islnk() (in module os.path), 271
islower() (in module os.path), 270
islower() (str method), 40
ismemberdescriptor() (module inspect), 1258
ismeta() (module curses.ascii), 542
ismethod() (module inspect), 1257
ismethoddescriptor() (module inspect), 1257
ismodule() (module inspect), 1257
ismount() (module os.path), 271
isnan() (module cmath), 212
isnan() (module math), 207
ISNONTERMINAL() (module tokenize), 1308
isnumeric() (str method), 40
isocalendar() (datetime.date method), 143
isocalendar() (datetime.datetime method), 149
isformat() (datetime.date method), 143
isformat() (datetime.datetime method), 149
isformat() (datetime.time method), 152
isolation_level (sqlite3.Connection attribute), 315
isoweekday() (datetime.date method), 143
isoweekday() (datetime.datetime method), 149
isprint() (module curses.ascii), 542
isprintable() (str method), 40
ispunct() (module curses.ascii), 542
issubclass() (module pprint), 197
isreadable() (module pprint, 198
isreadable() (pprint.PrettyPrinter method), 198
isreadable() (pprint.PrettyPrinter method), 198
isreg() (tarfile.TarInfo method), 351
isReservedKey() (http.cookies.Morsel method), 934
isroutine() (module inspect), 1257
isSameNode() (xml.dom.Node method), 814
isspace() (module curses.ascii), 542
isspace() (str method), 40
issubclass() (built-in function), 12
issubset() (set method), 55
issuite() (module parser), 1299
issuite() (parser.ST method), 1300
issuperset() (set method), 55
issym() (tarfile.TarInfo method), 351
ISTERMINAL() (module tokenize), 1308
istitle() (str method), 40

The Python Library Reference, Release 3.3.3

istraceback() (in module inspect), 1257
isub() (in module operator), 267
isupper() (in module curses.ascii), 543
isupper() (str method), 41
isthread() (in module turtle), 1014
isxdigit() (in module curses.ascii), 543
item() (tkinter.ttk.Treeview method), 1062
item() (xml.dom.NamedNodeMap method), 817
item() (xml.dom.NodeList method), 814
itemgetter() (in module operator), 264
items() (configparser.ConfigParser method), 374
items() (dict method), 58
items() (email.message.Message method), 722
items() (mailbox.Mailbox method), 770
items() (types.MappingProxyType method), 195
itemsize (array.array attribute), 188
itemsize (memoryview attribute), 53
ItemsView (class in collections.abc), 180
iter() (built-in function), 12
iter() (xml.etree.ElementTree method), 807
iter() (xml.etree.ElementTree.Element method), 808
iter_child_nodes() (in module ast), 1304
iter_fields() (in module ast), 1304
iter_items() (in module pkgutil), 1282
iter_key() (module pkgutil), 1282
iterable, 1369
Iterable (class in collections.abc), 180
iterator, 1369
Iterator (class in collections.abc), 180
iterator protocol, 32
iterdecode() (in module codecs), 124
iterdump (sqlite3.Connection attribute), 319
iterencode() (in module codecs), 124
iterencode() (json.JSONEncoder method), 766
iterfind() (xml.etree.ElementTree method), 807
iterfind() (xml.etree.ElementTree.Element method), 808
iterfind() (xml.etree.ElementTree.Element method), 807
iterfind() (xml.etree.ElementTree.Element method), 808
iteritems() (mailbox.Mailbox method), 770
iterkeys() (mailbox.Mailbox method), 770
itermonthdates() (calendar.Calendar method), 163
itermonthdays() (calendar.Calendar method), 163
itermonthdays2() (calendar.Calendar method), 163
iterparse() (module xml.etree.ElementTree), 804
itertext() (xml.etree.ElementTree.Element method), 807
itertools (2to3 fixer), 1171
itertools (module), 245
itertools_imports (2to3 fixer), 1171
itervalues() (mailbox.Mailbox method), 770
iterweekdays() (calendar.Calendar method), 162
ITIMER_PROF (in module signal), 1323
ITIMER_REAL (in module signal), 1323
ITIMER_VIRTUAL (in module signal), 1323
itruediv() (in module operator), 267
ixor() (in module operator), 267
J
Jansen, Jack, 792
java_ver() (in module platform), 546
join() (in module os.path), 271
join() (multiprocessing.JoinableQueue method), 602
join() (multiprocessing.Pool.Pool method), 615
join() (multiprocessing.Process method), 598
join() (queue.Queue method), 663
join() (str method), 41
join() (threading.Thread method), 585
join_thread() (multiprocessing.Queue method), 602
JoinableQueue (class in multiprocessing), 602
js_output() (http.cookies.BaseCookie method), 933
js_output() (http.cookies.Morsel method), 934
json (module), 761
JSONDecoder (class in json), 764
JSONEncoder (class in json), 765
jump (pdb command), 1198
JUMP_ABSOLUTE (opcode), 1323
JUMP_FORWARD (opcode), 1323
JUMP_IF_FALSE_OR_POP (opcode), 1323
JUMP_IF_TRUE_OR_POP (opcode), 1323
K
kbhit() (in module msvcrt), 1337
KDEDIR, 846
kevent() (in module select), 664
key (http.cookies.Morsel attribute), 934
key function, 1369
KEY_ALL_ACCESS (in module winreg), 1343
KEY_CREATE_LINK (in module winreg), 1343
KEY_CREATE_SUB_KEY (in module winreg), 1343
KEY_ENUMERATE_SUB_KEYS (in module winreg), 1343
KEY_EXECUTE (in module winreg), 1343
KEY_NOTIFY (in module winreg), 1343
KEY_QUERY_VALUE (in module winreg), 1343
KEY_READ (in module winreg), 1343
KEY_SET_VALUE (in module winreg), 1343
KEY_WRITE (in module winreg), 1343
KeyboardInterrupt, 67
KeyError, 67
keyname() (in module curses), 528
keypad() (curses.window method), 534
keyrefs() (weakref.WeakKeyDictionary method), 191
keys() (dict method), 58
keys() (email.message.Message method), 722
keys() (mailbox.Mailbox method), 770
keys() (sqlite3.Row method), 322
keys() (types.MappingProxyType method), 195
keys() (xml.etree.ElementTree.Element method), 806
KeysView (class in collections.abc), 180
keyword (module), 1309
keyword argument, 1369

Index 1437
Index

keywords (functools.partial attribute), 261
kill() (in module os), 414
kill() (subprocess.Popen method), 655
killchar() (in module curses), 528
killpg() (in module os), 414
kind (inspect.Parameter attribute), 1260
knownfiles (in module mimetypes), 785
kqueue() (in module select), 664
Kuchling, Andrew, 386
kwarggs (inspect.BoundArguments attribute), 1261
kwlist (in module keyword), 1309

L

L (in module re), 87
LabelEntry (class in tkinter.tix), 1067
LabelFrame (class in tkinter.tix), 1067
lambda, 1369
LambdaType (in module types), 194
LANG, 983, 985, 991, 994
LANGUAGE, 983, 985
language
C, 26, 27
large files, 1347
LargeZipFile, 341
last() (ntplib.NNTP method), 907
last_accepted (multiprocessing.connection.Listener attribute), 617
last_traceback (in module sys), 1222
last_type (in module sys), 1222
last_value (in module sys), 1222
lastChild (xml.dom.Node attribute), 813
lastcmd (cmd.Cmd attribute), 1033
lastgroup (re.match attribute), 93
lastindex (re.match attribute), 93
lastResort (in module logging), 505
lastrowid (sqlite3.Cursor attribute), 322
layout() (tkinter.ttk.Style method), 1064
LBRACE (in module token), 1308
LIBRARY (in module dbm.ndbm), 310
library (ssl.SSLError attribute), 689
LibraryLoader (class in ctypes), 571
license (built-in variable), 23
LifoQueue (class in queue), 662
light-weight processes, 669
limit_denominator() (fractions.Fraction method), 238
lin2adpcm() (in module audioop), 966
lin2alaw() (in module audioop), 966
lin2lin() (in module audioop), 966
lin2ulaw() (in module audioop), 966
line() (msilib.Dialog method), 1335
line_buffered I/O, 16
line_buffering (io.TextIOWrapper attribute), 430
line_num (csv.csvreader attribute), 359
lincache (module), 284
lineno (ast.AST attribute), 1301
lineno (doctest.DocTest attribute), 1092
lineno (doctest.Example attribute), 1093
lineno (pyclbr.Class attribute), 1314
lineno (pyclbr.Function attribute), 1314
lineno (shlex.shlex attribute), 1038
lineno (xml.parsers.expat.ExpatError attribute), 841
lineno() (in module fileinput), 273
LINES, 526, 531
lines (os.terminal_size attribute), 398
linesep (email.policy.Policy attribute), 732
linesep (in module os), 421
lineterminator (csv.Dialect attribute), 358
link() (in module os), 402
linkname (tarfile.TarInfo attribute), 350
linux_distribution() (in module platform), 547
list, 1369
  object, 34, 35
type, operations on, 34
list (built-in class), 35
list (pdb command), 1199
list comprehension, 1369
list() (imaplib.IMAP4 method), 900
Index
LockType (in module _thread), 669
log() (in module cmath), 211
log() (in module logging), 502
log() (in module math), 207
log() (logging.Logger method), 494
log10() (decimal.Context method), 226
log10() (decimal.Decimal method), 220
log10() (in module cmath), 211
log10() (in module math), 208
log1p() (in module math), 208
log2() (in module math), 208
log_date_time_string()
(http.server.BaseHTTPRequestHandler method), 930
log_error() (http.server.BaseHTTPRequestHandler method), 930
log_exception() (wsgiref.handlers.BaseHandler method), 861
log_message() (http.server.BaseHTTPRequestHandler method), 930
log_request() (http.server.BaseHTTPRequestHandler method), 930
log_to_stderr() (in module multiprocessing), 619
logb() (decimal.Context method), 226
logb() (decimal.Decimal method), 220
Logger (class in logging), 492
LoggerAdapter (class in logging), 500
logging
Errors, 491
logging (module), 491
logging.config (module), 505
logging.handlers (module), 514
logical_and() (decimal.Context method), 226
logical_and() (decimal.Decimal method), 220
logical_invert() (decimal.Context method), 227
logical_invert() (decimal.Decimal method), 220
logical_or() (decimal.Context method), 227
logical_or() (decimal.Decimal method), 220
logical_xor() (decimal.Context method), 227
logical_xor() (decimal.Decimal method), 220
login() (ftpclient.FTP method), 893
login() (imaplib.IMAP4 method), 900
login() (ntplib.NTPMethod method), 905
login() (smtplib.SMTP method), 911
login Kramer MS IS (imaplib.IMAP4 method), 900
LOGNAME, 390, 524
lognormvariate() (in module random), 241
logout() (imaplib.IMAP4 method), 900
LogRecord (class in logging), 498
long (2to3 fixer), 1171
longMessage (unittest.TestCase attribute), 1112
longname() (in module curses), 528
lookup() (in module codecs), 122
lookup() (in module unicodedata), 110
lookup() (syntable.SymbolTable method), 1306
lookup() (tkinter.ttk.Style method), 1064
lookup_error() (in module codecs), 123
LookupError, 66
loop() (in module asyncore), 704
lower() (str method), 41
LPAR (in module token), 1308
lru_cache() (in module functools), 258
lseek() (in module os), 395
lshtift() (in module operator), 262
LSQB (in module token), 1308
lstat() (in module os), 402
lstrip() (str method), 41
lsb() (imaplib.IMAP4 method), 900
lt() (in module operator), 261
lt() (in module turtle), 1004
LWPCookieJar (class in http.cookiejar), 938
Lzma (module), 336
LZMACompressor (class in Lzma), 337
LZMAdecompressor (class in Lzma), 338
LZMAError, 336
LZMAFile (class in Lzma), 336
M
M (in module re), 88
mac_ver() (in module platform), 547
machine() (in module platform), 545
macpath (module), 291
macros (netrc.netrc attribute), 377
MagicMock (class in unittest.mock), 1142
Mailbox (class in mailbox), 769
mailbox (module), 768
cmailcap (module), 768
Maildir (class in mailbox), 771
MaildirMessage (class in mailbox), 776
mailfrom (smtpd.SMTPChannel attribute), 915
MailmanProxy (class in smtpd), 915
main() (in module py_compile), 1315
main() (in module site), 1267
main() (in module unittest), 1118
mainloop() (in module turtle), 1023
maintype (email.headerregistry.ContentTypeHeader attribute), 739
major (email.headerregistry.MIMEVersionHeader attribute), 739
make_archive() (in module shutil), 289
make_bad_fd() (in module test.support), 1178
MAKE_CLOSURE (opcode), 1324
make_cookies() (http.cookiejar.CookieJar method), 937
make_file() (difflib.HtmlDiff method), 99
MAKE_FUNCTION (opcode), 1324
make_header() (in module email.header), 745
make_msgid() (in module email.message), 751
make_parser() (in module xml.sax), 826
make_server() (in module wsgiref.simple_server), 857
make_table() (difflib.HtmlDiff method), 99
makedev() (in module os), 403
makedirs() (in module os), 403
makeelement() (xml.etree.ElementTree.Element method), 807
meta path finder, 1370
meta() (in module curses), 528
meta_path (in module sys), 1222
metaclass, 1370
metaclass (2to3 fixer), 1172
MetaPathFinder (class in importlib.abc), 1287
MetaVarTypeHelpFormatter (class in argparser), 443
Meter (class in tkinter.tix), 1067
min (in module operator), 61
min (urllib.request.Request attribute), 867
min() (built-in function), 13
min() (decimal.Context method), 227
min() (decimal.Decimal method), 220
MIN_EMIN (in module decimal), 228
MIN_ETINY (in module decimal), 228
min_mag() (decimal.Context method), 227
min_mag() (decimal.Decimal method), 221
MINEQUAL (in module token), 1308
minmax() (in module audioop), 966
minor (email.headerregistry.MIMEVersionHeader attribute), 739
minor() (in module os), 403
MINUS (in module token), 1308
minus() (decimal.Context method), 227
minute (datetime.datetime attribute), 146
minute (datetime.timedelta attribute), 139
minute (datetime.time attribute), 152
MINYEAR (in module datetime), 137
mirrored() (in module unicodedata), 111
misc_header (cmd.Cmd attribute), 1033
MissingSectionHeaderError, 376
MIXERDEV, 978
mkd() (ftplib.FTP method), 895
mkdir() (in module os), 402
mkdir() (in module tempfile), 281
mkfifo() (in module os), 403
mknod() (in module os), 403
mkstemp() (in module tempfile), 281
mktemp() (in module tempfile), 281
mktime() (in module time), 434
mktime_tz() (in module email.utils), 750
mmap (in module mmap), 715
mmap (module), 715
MMDF (class in mailbox), 775
MMDFMessage (class in mailbox), 781
Mock (class in unittest.mock), 1123
Mock (class in unittest.mock, Mock method), 1125
mock_calls (unittest.mock.Mock attribute), 1129
mock_open() (in module unittest.mock), 1148
mod() (in module operator), 262
mode (io.FileIO attribute), 427
mode (os.pathdev.oss_audio_device attribute), 980
mode (tarfile.TarInfo attribute), 350
mode() (in module turtle), 1024
modf() (in module math), 207
modified() (urllib.robotparser.RobotFileParser method), 885
Modify() (msilib.View method), 1332
modify() (select.devpoll method), 665
modify() (select.epoll method), 666
modify() (select.poll method), 666
module, 1370
locale, 991
array, 45
base64, 789
bdb, 1195
binhex, 789
cmd, 1195
copy, 304
crypt, 1348
dbm.gnu, 306
dbm.ndbm, 306
erro, 67
glob, 284
imp, 21
math, 27, 212
os, 1347
pickle, 196, 304, 305, 307
pty, 395
pwd, 270
pyexpat, 836
re, 38, 283
search path, 284, 1222, 1266
shelve, 307
signal, 670
sitecustomize, 1266
socket, 845
stat, 406
string, 995
struct, 684
sys, 16
types, 62
urllib.request, 885
usercustomize, 1266
uu, 789

module (pyclbr.Class attribute), 1314
module (pyclbr.Function attribute), 1314
module_for_loader() (in module importlib.util), 1294
module_repr() (importlib.abc.Loader method), 1288
ModuleFinder (class in modulefinder), 1283
modulefinder (module), 1283
modules (in module sys), 1222
modules (modulefinder.ModuleFinder attribute), 1283
ModuleType (in module types), 194
monotonic() (in module time), 434
month (datetime.date attribute), 142
month (datetime.datetime attribute), 146
month() (in module calendar), 165
month_abbr (in module calendar), 165
month_name (in module calendar), 165
monthcalendar() (in module calendar), 164
monthdatescalendar() (calendar.Calendar method), 163
monthdays2calendar() (calendar.Calendar method), 163
monthdayscalendar() (calendar.Calendar method), 163
monthrange() (in module calendar), 164
Morsel (class in http.cookies), 933
most_common() (collections.Counter method), 168
mouseinterval() (in module curses), 528
mousenask() (in module curses), 528
move() (curses.panel.Panel method), 544
move() (curses.window method), 534
move() (in module mmap), 717
move() (in module shutil), 287
move() (tkinter.ttk.Treeview method), 1062
move_to_end() (collections.OrderedDict method), 177
MozillaCookieJar (class in http.cookiejar), 938
MRO, 1370
mro() (class method), 63
msg (http.client.HTTPResponse attribute), 889
msg() (telnetlib.Telnet method), 917
msi, 1331
msilib (module), 1331
msvcr106 (module), 1336
mt_interact() (telnetlib.Telnet method), 917
mtime (tarfile.TarInfo attribute), 350
mtime() (urllib.robotparser.RobotFileParser method), 885
mul() (in module audioop), 966
mul() (in module operator), 262
MultiCall (class in xmlrpc.client), 948
MULTILINE (in module re), 88
MultipartConversionError, 748
multiply() (decimal.Context method), 227
multiprocessing (module), 594
multiprocessing.connection (module), 616
multiprocessing.dummy (module), 620
multiprocessing.Manager() (in module multiprocessing.sharedctypes), 608
multiprocessing.managers (module), 608
multiprocessing.pool (module), 614
multiprocessing.sharedctypes (module), 606
mutable, 1370

sequence types, 34
MutableMapping (class in collections.abc), 180
MutableSequence (class in collections.abc), 180
MutableSet (class in collections.abc), 180
mvderwin() (curses.window method), 534
mvwin() (curses.window method), 535
myrights() (imaplib.IMAP4 method), 900

N

N_TOKENS (in module token), 1308
n_waiting (threading.Barrier attribute), 593
name (doctest.DocTest attribute), 1092
name (email.headerregistry.BaseHeader attribute), 737
name (http.cookiejar.Cookie attribute), 941
name (importlib abc.FileLoader attribute), 1289
name (importlib.machinery.ExtensionFileLoader attribute), 1294
name (importlib.machinery.SourceFileLoader attribute), 1293
name (importlib.machinery.SourcelessFileLoader attribute), 1294
name (in module os), 387
NAME (in module token), 1308
name (inspect.Parameter attribute), 1260
name (io.FileIO attribute), 427
name (multiprocessing.Process attribute), 598
name (ossaudiodev.oss_audio_device attribute), 980
name (pyclbr.Class attribute), 1314
name (pyclbr.Function attribute), 1314
ordered_attributes (xml.parsers.expat.xmlparser attribute), 838
OrderedDict (class in collections), 177
origin_req_host (urllib.request.Request attribute), 866
origin_server (wsgiref.handlers.BaseHandler attribute), 861
os
   module, 1347
   os (module), 387
   os.path (module), 269
   os.environ (wsgiref.handlers.BaseHandler attribute), 860
OSError, 67
ossaudiodev (module), 977
OSSAudioError, 977
output (subprocess.CalledProcessError attribute), 650
output (subprocess.TimeoutExpired attribute), 650
output() (http.cookies.BaseCookie method), 933
output() (http.cookies.Morsel method), 934
output_charset (email.charset.Charset attribute), 746
output_charset() (gettext.NullTranslations method), 986
output_codec (email.charset.Charset attribute), 746
output_difference() (doctest.OutputChecker method), 1096
OutputChecker (class in doctest), 1095
OutputString() (http.cookies.Morsel method), 934
over() (ntplib.NNTP method), 906
Overflow (class in decimal), 230
OverflowError, 67
overlaps() (ipaddress.IPv4Network method), 959
overlaps() (ipaddress.IPv6Network method), 961
overlay() (curses.window method), 535
overwrite() (curses.window method), 535

P
P_ALL (in module os), 417
P_DETACH (in module os), 415
P_NOWAIT (in module os), 415
P_NOWAITO (in module os), 415
P_OVERLAY (in module os), 415
P_PGID (in module os), 417
P_PID (in module os), 417
P_WAIT (in module os), 415
pack() (in module struct), 117
pack() (mailbox.MH method), 774
pack() (struct.Struct method), 121
pack_array() (xdrlib.Packer method), 378
pack_bytes() (xdrlib.Packer method), 378
pack_double() (xdrlib.Packer method), 378
pack_farray() (xdrlib.Packer method), 378
pack_float() (xdrlib.Packer method), 378
pack_fopaque() (xdrlib.Packer method), 378
pack_fstring() (xdrlib.Packer method), 378
pack_into() (xdrlib.Packer method), 378
pack() (struct.Struct method), 121
pack_list() (xdrlib.Packer method), 378
pack_opaque() (xdrlib.Packer method), 378
pack() (struct.Struct method), 121
pack_opaque() (xdrlib.Packer method), 378
pack() (struct.Struct method), 121
pack() (struct.Struct method), 121
pack_string() (xdrlib.Packer method), 378
package, 1266, 1370
packed (ipaddress.IPv4Address attribute), 955
packed (ipaddress.IPv6Address attribute), 956
Packer (class in xdr), 377
packing  binary data, 117
packing (widgets), 1046
pair_content() (in module curses), 529
pair_number() (in module curses), 529
PanedWindow (class in tkinter.tix), 1069
parameter, 1371
Parameter (class in inspect), 1259
ParameterizedMIMEHeader (class in email.headerregistry), 739
parameters (inspect.Signature attribute), 1259
params (email.headerregistry.ParameterizedMIMEHeader attribute), 739
pardir (in module os), 420
paren (2to3 fixer), 1172
parent (urllib.request.BaseHandler attribute), 869
parent() (tkinter.ttk.Treeview method), 1062
parentNode (xml.dom.Node attribute), 813
parents (collections.ChainMap attribute), 166
parse() (in module random), 241
parse() (doctest.DocTestParser method), 1094
parse() (email.parser.BytesParser method), 727
parse() (email.parser.Parser method), 727
parse() (in module ast), 1304
parse() (in module cgi), 850
parse() (in module xml.dom.minidom), 820
parse() (in module xml.dom.pulldom), 825
parse() (in module xml.etree.ElementTree), 805
parse() (in module xml.sax), 826
parse() (string.Formatter method), 74
parse() (urllib.robotparser.RobotFileParser method), 884
parse() (xml.etree.ElementTree.ElementTree method), 808
Parse() (xml.parsers.expat.xmlparser method), 837
parse() (xml.sax.xmlreader.XMLReader method), 833
parse_and_bind() (in module readline), 113
parse_args() (argparse.ArgumentParser method), 455
PARSE_COINAMES (in module sqilt3), 313
parse_config_h() (in module sysconfig), 1240
PARSE_DECLTYPES (in module sqilt3), 313
parse_header() (in module sys), 1222
parse_known_args() (argparse.ArgumentParser method), 464
parse() (xml.etree.ElementTree.ElementTree method), 808
ParseFile() (xml.parsers.expat.xmlparser method), 837
ParseFlag() (in module imaplib), 898
Parser (class in email.parser), 727
parser (module), 1297
ParserCreate() (in module xml.parsers.expat), 837
ParserError, 1299
ParseResult (class in urllib.parse), 882
ParseResultBytes (class in urllib.parse), 882
parameters (inspect.Signature attribute), 1259
path (http.cookiejar.Cookie attribute), 942
path (http.server.BaseHTTPRequestHandler attribute), 928
path (importlib.abc.FileLoader attribute), 1289
path (importlib.machinery.ExtensionFileLoader attribute), 1294
path (importlib.machinery.FileFinder attribute), 1293
path (importlib.machinery.SourceFileLoader attribute), 1293
path (importlib.machinery.SourcelessFileLoader attribute), 1294
path (in module sys), 1222
path based finder, 1371
Path browser, 1071
path (in module sys), 1222
path based finder, 1371
path (in module sys), 1222
path_based_finder (in module sys), 1223
path_importer_cache (in module sys), 1223
path (importlib.abc.SourceLoader method), 1290
parsedate() (in module email.utils), 750
parsedate_tz() (in module email.utils), 750
ParseError (class in xml.etree.ElementTree), 810
ParseFile() (xml.parsers.expat.xmlparser method), 837
ParseFlag() (in module imaplib), 898
Parser (class in email.parser), 727
parser (module), 1297
ParserCreate() (in module xml.parsers.expat), 837
ParserError, 1299
ParseResult (class in urllib.parse), 882
ParseResultBytes (class in urllib.parse), 882
parses() (email.parser.Parser method), 727
parseString() (in module xml.dom.minidom), 820
parseString() (in module xml.dom.pulldom), 825
parseString() (in module xml.sax), 826
parsing  Python source code, 1297
URL, 878
ParsingError, 376
partial() (imaplib.IMAP4 method), 900
partial() (in module functools), 259
parties (threading.Barrier attribute), 593
partition() (str method), 41
pass() (poplib.POP3 method), 896
Paste, 1073
patch() (in module unittest.mock), 1134
patch.dict() (in module unittest.mock), 1136
patch.multiple() (in module unittest.mock), 1138
patch.object() (in module unittest.mock), 1136
patch.stopall() (in module unittest.mock), 1139
PATH, 412, 415, 421, 845, 851, 853
path  configuration file, 1266
module search, 284, 1222, 1266
operations, 269
path (http.cookiejar.Cookie attribute), 942
path (http.server.BaseHTTPRequestHandler attribute), 928
path (importlib.abc.FileLoader attribute), 1289
path (importlib.machinery.ExtensionFileLoader attribute), 1294
path (importlib.machinery.FileFinder attribute), 1293
path (importlib.machinery.SourceFileLoader attribute), 1293
path (importlib.machinery.SourcelessFileLoader attribute), 1294
path (in module sys), 1222
path based finder, 1371
Path browser, 1071
path entry, 1371
path entry finder, 1371
path entry hook, 1371
path_hook() (importlib.machinery.FileFinder class method), 1293
path_hooks (in module sys), 1222
path_importer_cache (in module sys), 1223
path_mtime() (importlib.abc.SourceLoader method), 1290
path_return_ok() (http.cookiejar.CookiePolicy
method), 939
path_stats() (importlib.abc.SourceLoader method),
1290
path_stats() (importlib.machinery.SourceFileLoader
method), 1293
pathconf() (in module os), 403
pathconf_names (in module os), 404
PathEntryFinder (class in importlib.abc), 1288
PathFinder (class in importlib.machinery), 1292
pathname2url() (in module urllib.request), 864
pathsep (in module os), 421
pattern (re.regex attribute), 91
pause() (in module signal), 712
PAX_FORMAT (in module tarfile), 347
pax_headers (tarfile.TarFile attribute), 350
pax_headers (tarfile.TarInfo attribute), 351
pdl() (in module turtle), 1010
Pdb (class in pdb), 1195, 1196
pdb (module), 1195
peek() (bz2.BZ2File method), 334
peek() (gzip.GzipFile method), 333
peek() (io.BufferedReader method), 428
peek() (lzma.LZMAFile method), 337
peer (smtpd.SMTPChannel attribute), 915
PEM_cert_to_DER_cert() (in module ssl), 692
pem() (in module turtle), 1010
pencolor() (in module turtle), 1011
PendingDeprecationWarning, 70
pendown() (in module turtle), 1010
pensize() (in module turtle), 1010
PERCENT (in module token), 1308
PERCENTEQUAL (in module token), 1308
phase() (in module cmath), 210
pi (in module cmath), 212
pi (in module math), 209
pi (module), 196, 304, 305, 307
pickle (module), 293
pickle() (in module copyreg), 304
PickleError, 295
Pickler (class in pickle), 295
pickletools (module), 1324
pickletools command line option
-a, --annotate, 1325
-l, --indentlevel=<num>, 1325
-m, --memo, 1325
-o, --output=<file>, 1325
-p, --preamble=<preamble>, 1325
pickling
objects, 293
PicklingError, 295
pid (multiprocessing.Process attribute), 599
pid (subprocess.Popen attribute), 655
PIPE (in module subprocess), 650
Pipe() (in module multiprocessing), 601
pipe() (in module os), 395
pipe2() (in module os), 395
PIPE_BUF (in module select), 664
pipelines (module), 1356
PKG_DIRECTORY (in module imp), 1278
pkgutil (module), 1280
platform (module), 1223
platform() (in module platform), 544
platform() (in module platform), 545
PlaySound() (in module winsound), 1345
plist
file, 380
plistlib (module), 380
plock() (in module os), 414
PLUS (in module token), 1308
plus() (decimal.Context method), 227
PLUS_EQUAL (in module token), 1308
pm() (in module pdb), 1196
POINTER() (in module ctypes), 576
counter() (in module ctes), 576
polar() (in module cmath), 211
Policy (class in email.policy), 732
poll() (in module select), 664
poll() (multiprocessing.Connection method), 604
poll() (select.devpoll method), 665
poll() (select.epoll method), 666
poll() (select.poll method), 666
poll() (subprocess.Popen method), 654
Pool (class in multiprocessing.pool), 614
pop() (array.array method), 189
pop() (asynchat.fifo method), 709
pop() (collections.deque method), 171
pop() (dict method), 58
pop() (mailbox.Mailbox method), 771
pop() (sequence method), 34
pop() (set method), 56
POP3
protocol, 895
POP3 (class in poplib), 896
POP3_SSL (class in poplib), 896
pop_all() (contextlib.ExitStack method), 1328
POP_BLOCK (opcode), 1321
putwin() (curses.window method), 535
pwd
module, 270
pwd (module), 1348
pwd() (ftplib.FTP method), 895
pwrite() (in module os), 396
py_compile (module), 1314
PY_COMPILED (in module imp), 1278
PY_FROZEN (in module imp), 1278
py_object (class in ctypes), 580
PY_SOURCE (in module imp), 1278
pyclbr (module), 1313
PyCompileError, 1314
PyDLL (class in ctypes), 570
pydoc (module), 1077
pyexpat
module, 836
PYFUNCTYPE() (in module ctypes), 573
PyLoader (class in importlib.abc), 1290
PyPycLoader (class in importlib.abc), 1291
Python 3000, 1371
Python Editor, 1071
Python Enhancement Proposals
PEP 0205, 192
PEP 0343, 1243
PEP 227, 1252
PEP 235, 1286
PEP 236, 7
PEP 237, 45
PEP 238, 1252, 1367
PEP 246, 324
PEP 249, 311, 313
PEP 255, 1252
PEP 263, 1286, 1310, 1311
PEP 273, 1279
PEP 278, 1373
PEP 282, 289, 505
PEP 292, 80
PEP 302, 21, 285, 1222, 1223, 1278, 1279, 1281–1289, 1295, 1367, 1369
PEP 305, 355
PEP 307, 294
PEP 3101, 74
PEP 3105, 1252
PEP 3107, 1367
PEP 3112, 1252
PEP 3115, 194
PEP 3116, 1373
PEP 3118, 49
PEP 3119, 181, 1243
PEP 3120, 1286
PEP 3141, 203, 1243
PEP 3147, 1277, 1285, 1286, 1291, 1315, 1316
PEP 3148, 647
PEP 3149, 1215
PEP 3151, 70, 663, 674, 1357
PEP 3155, 1372
PEP 324, 647
PEP 328, 21, 1252, 1286
PEP 3333, 854–858, 861
PEP 338, 1286
PEP 343, 1252, 1366
PEP 362, 1261, 1365, 1371
PEP 366, 1286
PEP 370, 1268
PEP 378, 77
PEP 383, 122, 125, 673
PEP 393, 130, 1222
PEP 397, 1182
PEP 405, 1180
PEP 411, 1371
PEP 420, 1367, 1370, 1371
PEP 8, 758
python_branch() (in module platform), 545
python_build() (in module platform), 545
python_compiler() (in module platform), 545
PYTHON_DOM, 811
python_implementation() (in module platform), 545
python_revision() (in module platform), 546
python_version() (in module platform), 546
python_version_tuple() (in module platform), 546
PYTHONDOCS, 1078
PYTHONDONTWRITEBYTECODE, 1217
PYTHONFAULTHANDLER, 1193
Pythonic, 1371
PYTHONIOENCODING, 1225
PYTHONNOUSERSITE, 1267
PYTHONPATH, 851, 1222
PYTHONSTARTUP, 115, 116, 1074
PYTHONUSERBASE, 1267
PyZipFile (class in zipfile), 344
Q
qiflush() (in module curses), 529
QName (class in xml.etree.ElementTree), 809
qsize() (multiprocessing.Queue method), 601
qsize() (queue.Queue method), 662
qualified name, 1372
quantize() (decimal.Context method), 228
quantize() (decimal.Decimal method), 221
QueryInfoKey() (in module winreg), 1340
QueryReflectionKey() (in module winreg), 1342
QueryValue() (in module winreg), 1340
QueryValueEx() (in module winreg), 1341
Queue (class in multiprocessing), 601
Queue (class in queue), 661
queue (module), 661
queue (sched.scheduler attribute), 661
Queue() (multiprocessing.managers.SyncManager method), 610
QueueHandler (class in logging.handlers), 523
QueueListener (class in logging.handlers), 523
quick_ratio() (difflib.SequenceMatcher method), 104
quit (built-in variable), 23
quit (pdb command), 1200
quit() (ftplib.FTP method), 895

quit() (ntplib.NNTP method), 905
quit() (poplib.POP3 method), 897
quit() (smtplib.SMTP method), 913
quopri (module), 791
quote() (in module email.utils), 749
quote() (in module shlex), 1036
quote() (in module urllib.parse), 882
QUOTE_ALL (in module csv), 357
quote_from_bytes() (in module urllib.parse), 883
QUOTE_MINIMAL (in module csv), 357
QUOTE_NONE (in module csv), 358
QUOTE_NONNUMERIC (in module csv), 358
quote_plus() (in module urllib.parse), 882
quoteattr() (in module xml.sax.saxutils), 832
quoted-printable
  encoding, 791
quotes (shlex.shlex attribute), 1038
quoting (csv.Dialect attribute), 358
R
R_OK (in module os), 400
radians() (in module math), 209
radians() (in module turtle), 1010
RadioButtonGroup (class in msilib), 1335
radiogroup() (msilib.Dialog method), 1335
radix() (decimal.Context method), 228
radix() (decimal.Decimal method), 222
RADIXCHAR (in module locale), 993
raise
  statement, 65
raise (2to3 fixer), 1172
raise_on_defect (email.policy.Policy attribute), 732
RAISE_VARARGS (opcode), 1324
RAND_add() (in module ssl), 691
RAND_bytes() (in module ssl), 691
RAND_egg() (in module ssl), 691
RAND_pseudo_bytes() (in module ssl), 691
RAND_status() (in module ssl), 691
randint() (in module random), 240
random (module), 240
random (module), 240
random (module), 240
random (module), 240
range
  object, 36
range (built-in class), 36
RARRROW (in module token), 1308
ratecv() (in module audioop), 967
ratio() (difflib.SequenceMatcher method), 104
Rational (class in numbers), 203
raw (io.BufferedIOBase attribute), 426
raw() (in module curses), 529
raw_decode() (json.JSONDecoder method), 765
raw_input (2to3 fixer), 1172
raw_input() (code.InteractiveConsole method), 1272
RawArray() (in module multiprocessing.sharedctypes), 606
RawConfigParser (class in configparser), 375
RawDescriptionHelpFormatter (class in argparser), 443
RawIOBase (class in io), 425
RawPen (class in turtle), 1026
RawTextHelpFormatter (class in argparser), 443
RawTurtle (class in turtle), 1026
RawValue() (in module multiprocessing.sharedctypes), 606
RBRACE (in module token), 1308
rcpttos (smtpd.SMTPChannel attribute), 915
re
  module, 38, 283
re (module), 82
re (re.match attribute), 93
read() (chunk.Chunk method), 975
read() (codecs.StreamReader method), 128
read() (configparser.ConfigParser method), 373
read() (http.client.HTTPResponse method), 889
read() (imaplib.IMAP4 method), 900
read() (in module mmap), 717
read() (in module os), 396
read() (io.BufferedIOBase method), 426
read() (io.BufferedReader method), 428
read() (io.RawIOBase method), 425
read() (io.TextIOBase method), 429
read() (mimetypes.MimeTypes method), 786
read() (ossaudiodev.oss_audio_device method), 978
read() (urllib.robotparser.RobotFileParser method), 884
read() (zipfile.ZipFile method), 343
read1() (io.BufferedIOBase method), 426
read1() (io.BufferedReader method), 428
read1() (io.BytesIO method), 427
read_all() (telnetlib.Telnet method), 916
read_byte() (in module mmap), 717
read_dict() (configparser.ConfigParser method), 373
read_eager() (telnetlib.Telnet method), 917
read_environ() (in module wsgiref.handlers), 862
read_file() (configparser.ConfigParser method), 373
read_history_file() (in module readline), 113
read_init_file() (in module readline), 113
read_lazy() (telnetlib.Telnet method), 917
read_mime_types() (in module mimetypes), 785
read_sb_data() (telnetlib.Telnet method), 917
read_some() (telnetlib.Telnet method), 917
read_string() (configparser.ConfigParser method), 373
read_token() (shlex.shlex method), 1037
read_until() (telnetlib.Telnet method), 916
read_very_eager() (telnetlib.Telnet method), 917
read_very_lazy() (telnetlib.Telnet method), 917
read_windows_registry() (mimetypes.MimeTypes method), 787
readable() (asyncore.dispatcher method), 705
readable() (io.IOBase method), 424
readall() (io.IOBase method), 425
reader() (in module csv), 355
ReadError, 347
readfp() (configparser.ConfigParser method), 374
readfp() (mimetypes.MimeTypes method), 787
readframes() (aifc.aifc method), 969
The Python Library Reference, Release 3.3.3

rpc.server.SimpleXMLRPCServer method), 951
register_multicall_functions() (xmlrpc.server.CGIXMLRPCRequestHandler method), 952
register_multicall_functions() (xmlrpc.server.SimpleXMLRPCServer method), 951
register_namespace() (in module xml.etree.ElementTree), 805
register_optionflag() (in module doctest), 1086
register_shape() (in module turtle), 1024
register_unpack_format() (in module shutil), 290
registerDOMImplementation() (in module xml.dom), 811
registerResult() (in module unittest), 1120
regular package, 1372
relative URL, 878
release() (_thread.lock method), 670
release() (in module platform), 546
release() (logging.Handler method), 495
release() (memoryview method), 50
release() (threading.Condition method), 589
release() (threading.Lock method), 587
release() (threading.RLock method), 588
release() (threading.Semaphore method), 590
release_lock() (in module imp), 1278
reload() (in module imp), 1276
relpath() (in module os.path), 271
remainder() (decimal.Context method), 228
remainder() (decimal.Context method), 228
remainder() (decimal.Decimal method), 222
remove() (array.array method), 189
remove() (collections.deque method), 171
remove() (in module os), 404
remove() (mailbox.Mailbox method), 769
remove() (mailbox.MH method), 774
remove() (sequence method), 34
remove() (set method), 56
remove() (xml.etree.ElementTree.Element method), 807
remove_flag() (mailbox.MaildirMessage method), 777
remove_flag() (mailbox.mboxMessage method), 779
remove_flag() (mailbox.MMDFMessage method), 782
remove_folder() (mailbox.Maildir method), 772
remove_folder() (mailbox.MH method), 774
remove_history_item() (in module readline), 114
replace() (in module codecs), 123
replace() (textwrap.TextWrapper attribute), 109
replaceChild() (xml.dom.Node method), 814
ReplacePackage() (in module modulefinder), 1283
report() (filecmp.dircmp method), 278
report() (modulefinder.ModuleFinder method), 1283
REPORT_CDIFF (in module doctest), 1086
report_failure() (doctest.DocTestRunner method), 1095
report_full_closure() (filecmp.dircmp method), 279
REPORT_NDIFF (in module doctest), 1086
REPORT_ONLY_FIRST_FAILURE (in module doctest), 1086
report_partial_closure() (filecmp.dircmp method), 279
REPORT_UDIFF (in module doctest), 1086
report_renames() (2to3 fixer), 1172
report_reorganize() (dbm.gnu.hash method), 310
replace() (in module itertools), 253
repeat() (in module timeit), 1208
report() (timeit.Timer method), 1208
repetition
replace() (curses.panel.Panel method), 544
replace() (datetime.date method), 142
replace() (datetime.datetime method), 147
replace() (datetime.time method), 152
replace() (in module os), 404
replace() (inspect.Parameter method), 1260
replace() (inspect.Signature method), 1259
replace() (str method), 41
replace() (in module codecs), 123
report() (filecmp.dircmp method), 278
report_unexpected_exception() (doctest.DocTestRunner method), 1095
REPORTING_FLAGS (in module doctest), 1086
repr (2to3 fixer), 1172
repr (str class in reprlib), 200
repr() (built-in function), 17
repr() (in module reprlib), 200
Index
same_quantum() (decimal.Context method), 228
same_quantum() (decimal.Decimal method), 222
samefile() (in module os.path), 271
sameopenfile() (in module os.path), 271
samestat() (in module os.path), 271
sample() (in module random), 240
save() (http.cookiejar.FileCookieJar method), 938
SaveKey() (in module winreg), 1341
savetty() (in module curses), 529
SAX2DOM (class in xml.dom.pulldom), 825
SAXException, 826
SAXNotRecognizedException, 826
SAXNotSupportedException, 827
SAXParseException, 826
scaleb() (decimal.Context method), 228
scaleb() (decimal.Decimal method), 222
scanf(), 94
sched (module), 660
SCHED_BATCH (in module os), 419
SCHED_FIFO (in module os), 419
sched_get_priority_max() (in module os), 419
sched_get_priority_min() (in module os), 419
sched_getparam() (in module os), 419
sched_getscheduler() (in module os), 419
SCHED_IDLE (in module os), 419
SCHED_OTHER (in module os), 418
sched_param (class in os), 419
sched_priority (os.sched_param attribute), 419
SCHED_RESET_ON_FORK (in module os), 419
SCHED_RR (in module os), 419
sched_rr_get_interval() (in module os), 419
sched_setaffinity() (in module os), 419
sched_setparam() (in module os), 419
SCHED_SPORADIC (in module os), 419
sched_yield() (in module os), 419
scheduler (class in sched), 660
Screen (class in turtle), 1026
screensize() (in module turtle), 1020
script_from_examples() (in module doctest), 1097
scroll() (curses.window method), 535
ScrolledCanvas (class in turtle), 1026
scroll() (curses.window method), 536
search
  path, module, 284, 1222, 1266
search() (imaplib.IMAP4 method), 901
search() (in module re), 88
search() (re.regex method), 90
second (datetime.datetime attribute), 146
second (datetime.time attribute), 152
SECTCRE (in module configparser), 370
sections() (configparser.ConfigParser method), 372
secure (http.cookiejar.CookieAttribute), 942
secure hash algorithm, SHA1, SHA224, SHA256, SHA384, SHA512, 383
Secure Sockets Layer, 688
security
  CGI, 851
see() (tkinter.ttk.Treeview method), 1062
seed() (in module random), 240
seek() (chunk.Chunk method), 975
seek() (in module mmap), 717
seek() (io.IOBase method), 424
seek() (io.TextIOBase method), 429
SEEK_CUR (in module os), 395
SEEK_END (in module os), 395
SEEK_SET (in module os), 395
seekable() (io.IOBase method), 425
seen_greeting (smtplib.SMTPChannel attribute), 915
Select (class in tkinter.tix), 1068
select (module), 663
select() (imaplib.IMAP4 method), 901
select() (in module select), 664
select() (tkinter.ttk.Notebook method), 1056
selected_npn_protocol() (ssl.SSLSocket method), 696
selection() (tkinter.ttk.Treeview method), 1062
selection_add() (tkinter.ttk.Treeview method), 1062
selection_remove() (tkinter.ttk.Treeview method), 1062
selection_set() (tkinter.ttk.Treeview method), 1062
selector (urllib.request.Request attribute), 866
Semaphore (class in multiprocessing), 605
Semaphore (class in threading), 590
Semaphore() (multiprocessing.managers.SyncManager method), 610
semaphores, binary, 669
SEMI (in module token), 1308
send() (asyncore.dispatcher method), 706
send() (http.client.HTTPConnection method), 889
send() (imaplib.IMAP4 method), 901
send() (logging.handlers.DatagramHandler method), 519
send() (logging.handlers.SocketHandler method), 518
send() (multiprocessing.Connection method), 603
send() (socket.socket method), 683
send_bytes() (multiprocessing.Connection method), 604
send() (smtplib.SMTP method), 912
send_error() (http.server.BaseHTTPRequestHandler method), 929
send() (socket.socket method), 683
send_flow_style() (formatter.writer method), 1330
send() (socket.socket method), 683
send() (http.server.BaseHTTPRequestHandler method), 930
send() (http.server.BaseHTTPRequestHandler method), 930
send() (http.server.BaseHTTPRequestHandler method), 930
send() (http.server.BaseHTTPRequestHandler method), 930
send() (subprocess.Popen method), 655
sendall() (socket.socket method), 683
sethostname() (in module socket), 679
SetInteger() (msilib.Record method), 1333
setItem() (in module operator), 263
setitimer() (in module signal), 713
setLevel() (logging.Handler method), 495
setLevel() (logging.Logger method), 492
setlocale() (in module locale), 991
setMark() (xml.sax.xmlreader.XMLReader method), 834
setLoggerClass() (in module logging), 504
setMaxConns() (urllib.request.CacheFTPHandler method), 872
setmode() (in module msvcrt), 1336
setName() (threading.Thread method), 586
setnchannels() (aifc.aifc method), 969
setnchannels() (sunau.AU_write method), 972
setnchannels() (wave.Wave_write method), 974
setnframes() (aifc.aifc method), 969
setnframes() (sunau.AU_write method), 972
setnframes() (wave.Wave_write method), 974
SetParamEntityParsing() (xml.parsers.expat.xmlparser method), 838
setProperty() (xml.sax.xmlreader.XMLReader method), 834
setpgid() (in module os), 391
setpgrp() (in module os), 391
setpos() (aifc.aifc method), 969
setpos() (sunau.AU_read method), 971
setpos() (wave.Wave_read method), 973
setProperty() (in module os), 391
setProfile() (logging.handlers.MemoryHandler method), 522
setpassword() (zipfile.ZipFile method), 343
setpriority() (in module os), 391
setprofile() (in module sys), 1224
setprofile() (in module threading), 583
setProfile() (xml.parsers.expat.xmlparser method), 838
setpriority() (in module os), 391
setposition() (in module turtle), 1025
setrlimit() (in module resource), 1358
setsampwidth() (aifc.aifc method), 969
setsampwidth() (sunau.AU_write method), 972
setsampwidth() (wave.Wave_write method), 974
setStderr() (in module socket), 679
setStdin() (in module socket), 679
setStdout() (in module socket), 679
setTrace() (in module sys), 1224
setTrace() (in module threading), 583
settdump() (in module sys), 1224
setup() (in module turtle), 1005
setup() (socketserver.RequestHandler method), 924
setUp() (unittest.TestCase method), 1106
setUp() (socketserver.RequestHandler method), 924
setup_python() (venv.EnvBuilder method), 1183
setup() (socketserver.RequestHandler method), 924
setuac() (in module winreg), 1341
setUser() (xml.sax.XMLVersion method), 834
setworldcoordinates() (in module turtle), 1021
setText() (in module turtle), 1005
setText() (in module turtle), 1005
setxattr() (in module os), 411
setX() (in module turtle), 1005
setX() (in module turtle), 1005
setX() (in module turtle), 1005
setX() (in module turtle), 1005
setX() (in module turtle), 1005
setX() (in module turtle), 1005
Shelf (class in shelve), 306
shelve module, 307
shelve (module), 305
shift() (decimal.Context method), 228
shift() (decimal.Decimal method), 222
shift_path_info() (in module wsgiref.util), 854
shifting operations, 28
shlex (class in shlex), 1037
shlex (module), 1036
shortDescription() (unittest.TestCase method), 1112
shouldFlush() (logging.handlers.BufferingHandler method), 522
shouldFlush() (logging.handlers.MemoryHandler method), 522
shouldStop (unittest.TestResult attribute), 1116
show() (curses.panel.Panel method), 544
show_code() (in module dis), 1317
showsyntaxerror() (code.InteractiveInterpreter method), 1272
showtraceback() (code.InteractiveInterpreter method), 1272
showturtle() (in module turtle), 1014
showwarning() (in module warnings), 1235
shutdown() (concurrent.futures.Executor method), 643
shutdown() (imaplib.IMAP4 method), 901
shutdown() (in module logging), 504
shutdown() (multiprocessing.managers.BaseManager method), 609
shutdown() (socket.socket method), 684
shutdown() (socketserver.BaseServer method), 923
shutil (module), 285
side_effect (unittest.mock.Mock attribute), 1127
SIG_BBLOCK (in module signal), 712
SIG_DFL (in module signal), 711
SIG_IGN (in module signal), 711
SIG_SETMASK (in module signal), 712
SIG_UNBLOCK (in module signal), 712
siginterrupt() (in module signal), 713
signal module, 670
signal (module), 710
signal() (in module signal), 713
Signature (class in inspect), 1259
signature() (in module inspect), 1258
sigpending() (in module signal), 714
sigtermwaint() (in module signal), 714
sigwait() (in module signal), 714
sigwaitinfo() (in module signal), 714
Simple Mail Transfer Protocol, 909
SimpleCookie (class in http.cookies), 933
simplefilter() (in module warnings), 1235
SimpleHandler (class in wsgiref.handlers), 859
SimpleHTTPRequestHandler (class in http.server), 930
SimpleNamespace (class in types), 195
SimpleQueue (class in multiprocessing), 602
SimpleXMLRPCRequestHandler (class in xmlrpc.server), 950
SimpleXMLRPCServer (class in xmlrpc.server), 950
sin() (in module cmath), 211
sin() (in module math), 208
SimpleAddressHeader (class in email.headerregistry), 739
sinh() (in module cmath), 212
sinh() (in module math), 209
site (module), 1265
site command line option
--user-base, 1267
--user-site, 1267
site-packages
directory, 1266
site-python
directory, 1266
sitecustomize module, 1266
sixtofour (ipaddress.IPv6Address attribute), 957
size (struct.Struct attribute), 121
size (tarfile.TarInfo attribute), 350
size() (ftplib.FTP method), 895
size() (in module mmap), 717
Sized (class in collections.abc), 180
sizeof() (in module ctypes), 576
SKIP (in module doctest), 1085
skipt() (chunk.Chunk method), 975
skipt() (in module unittest), 1106
skipUnless (in module unittest), 1106
skipped (unittest.TestResult attribute), 1116
skippedEntity() (xml.sax.handler.ContentHandler method), 831
SkipTest, 1106
skipTest() (unittest.TestCase method), 1107
skipUnless() (in module unittest), 1106
SLASH (in module token), 1308
SLASHEQUAL (in module token), 1308
slave() (ntplib.NTPP method), 908
sleep() (in module time), 434
slice, 1372
assignment, 34
built-in function, 1324
operation, 32
slice() (built-in function), 18
SMTP protocol, 909
SMTP (class in smtplib), 909
SMTP (in module email.policy), 736
smtp_server (smtplib.SMTPChannel attribute), 915
SMTP_SSL (class in smtplib), 909
smtp_state (smtplib.SMTPChannel attribute), 915
SMTPAuthenticationError, 910
SMTPChannel (class in smtpd), 915
SMTPConnectError, 910
smtpd (module), 914
Index 1465
STORE_ACTIONS (optparse.Option attribute), 488
STORE_ATTR (opcode), 1322
STORE_DEREF (opcode), 1323
STORE_FAST (opcode), 1323
STORE_GLOBAL (opcode), 1322
STORE_LOCALS (opcode), 1322
STORE_MAP (opcode), 1323
STORE_NAME (opcode), 1322
STORE_SUBSCR (opcode), 1320
storlines() (ftplib.FTP method), 894
str (built-in class), 38
   (see also string), 37
str() (in module locale), 995
strcoll() (in module locale), 994
StreamError, 347
StreamHandler (class in logging), 514
StreamReader (class in codecs), 128
StreamReaderWriter (class in codecs), 129
StreamRecoder (class in codecs), 129
streams, 121
   stackable, 121
StreamWriter (class in codecs), 127
sterror() (in module os), 392
strftime() (datetime.date method), 143
strftime() (datetime.datetime method), 149
strftime() (datetime.time method), 153
strftime() (in module time), 435
strict (csv.Dialect attribute), 359
strict (module), 995
strict (object), 37
strict (in module email.policy), 736
strict_domain (http.cookiejar.DefaultCookiePolicy attribute), 940
strict_errors() (in module codecs), 123
strict_ns_domain (http.cookiejar.DefaultCookiePolicy attribute), 941
strict_ns_set_initial_dollar (http.cookiejar.DefaultCookiePolicy attribute), 941
strict_ns_set_path (http.cookiejar.DefaultCookiePolicy attribute), 941
strict_ns_unverifiable (http.cookiejar.DefaultCookiePolicy attribute), 941
strict_rfc2965_unverifiable (http.cookiejar.DefaultCookiePolicy attribute), 940
strides (memoryview attribute), 54
string
   format() (built-in function), 11
   formatting, 43
   interpolation, 43
   methods, 38
   module, 995
   object, 37
   str (built-in class), 38
   str() (built-in function), 19
   text sequence type, 37
STRING (in module string), 1308
string (module), 73
string (re.match attribute), 93
string_at() (in module ctypes), 576
StringIO (class in io), 430
stringprep (module), 112
strip() (str method), 42
strip_dirs() (pstats.Stats method), 1203
stripspaces (curses.textpad.Textbox attribute), 541
strptime() (datetime.datetime class method), 145
strptime() (in module time), 436
struct
   module, 684
   Struct (in module struct), 121
   struct (module), 117
   struct sequence, 1372
   struct_time (class in time), 436
   Structure (class in ctypes), 580
   structures
      C, 117
strxfrm() (in module locale), 994
STType (in module parser), 1300
Style (class in tkinter.ttk), 1063
sub() (in module operator), 263
sub() (in module re), 89
sub() (re.regex method), 91
subdirs (filecmp.dircmp attribute), 279
SubElement() (in module xml.etree.ElementTree), 805
submit() (concurrent.futures.Executor method), 643
subn() (in module re), 90
subn() (re.regex method), 91
subnets() (ipaddress.IPv4Network method), 959
subnets() (ipaddress.IPv6Network method), 961
Subnormal (class in decimal), 230
suboffsets (memoryview attribute), 54
subpad() (curses.window attribute), 54
subprocess (module), 647
SubprocessError, 650
subscribe() (imaplib.IMAP4 method), 902
subscript
   assignment, 34
   operation, 32
suffix_map (in module mimetypes), 785
suffix_map (mimetypes.MimeTypes attribute), 786
suite() (in module unittest.TestLoader), 1115
sum() (built-in function), 19
summarize() (doctest.DocTestRunner method), 1095
summarize_address_range() (in module ipaddress), 963
sunau (module), 970
super (pyclbr.Class attribute), 1314
super() (built-in function), 19
supernet() (ipaddress.IPv4Network method), 960
supernet() (ipaddress.IPv6Network method), 961
supports_bytes_environ (in module os), 392
supports_dir_fd (in module os), 407
supports_effective_ids (in module os), 407
supports_fd (in module os), 407
supports_followシンLINKS (in module os), 407
supports_unicode_filenanes (in module os.path), 272
SW_HIDE (in module subprocess), 656
swapcase() (str method), 43
sym_name (in module symbol), 1307
Symbol (class in symtable), 1307
symbol (module), 1307
SymbolTable (class in symtable), 1307
system() (in module os), 416
system() (in module platform), 546
system_alias() (in module platform), 546
SystemError, 68
SystemExit, 68
SystemRandom (class in random), 242
SystemRoot, 653
T
T_FMT (in module locale), 993
T_FMT_APM (in module locale), 993
tab() (tkinter.ttk.Notebook method), 1056
TabError, 68
tabnanny (module), 1313
tabs() (tkinter.ttk.Notebook method), 1056
tabsize (textwrap.TextWrapper attribute), 109
tabular
data, 355
tag (xml.etree.ElementTree.Element attribute), 806
tag_bind() (tkinter.ttk.Treeview method), 1062
tag_setconfig() (tkinter.ttk.Treeview method), 1063
tag_has() (tkinter.ttk.Treeview method), 1063
tagName (xml.dom.Element attribute), 816
tail (xml.etree.ElementTree.Element attribute), 806
takewhile() (in module itertoolis), 253
tan() (in module math), 211
tan() (in module math), 208
tanh() (in module math), 212
tanh() (in module math), 209
TarError, 347
TarFile (class in tarfile), 347, 348
tarfile (module), 346
target (xml.dom.ProcessingInstruction attribute), 818
TarInfo (class in tarfile), 350
task_done() (multiprocessing.JoinableQueue method), 602
task_done() (queue.Queue method), 663
thbreak (pdb command), 1197
tcdrain() (in module termios), 1352
tcflow() (in module termios), 1352
tcflush() (in module termios), 1352
tcgetattr() (in module termios), 1352
tcgetpgrp() (in module os), 397
Tel() (in module tkinter), 1042
tcsendbreak() (in module termios), 1352
tcsetattr() (in module termios), 1352
tcsetpgrp() (in module os), 397
tearDown() (unittest.TestCase method), 1107
tearDownClass() (unittest.TestCase method), 1107
tee() (in module itertoolis), 253
tell() (aifc.aifc method), 969
tell() (chunk.Chunk method), 975
tell() (in module mmap), 717
tell() (io.IOBase method), 42S
tell() (io.TextIOBase method), 429
tell() (sunau.AU_read method), 971
tell() (sunau.AU_write method), 972
tell() (wave.Wave_read method), 973
tell() (wave.Wave_write method), 974
Telnet (class in telnetlib), 916
telnetlib (module), 916
TEMP, 282
temp_cwd() (in module test.support), 1178
temp_dir() (in module test.support), 1178
temp_umask() (in module test.support), 1178
tempdir (in module tempfile), 282
tempfile (module), 280
Template (class in pipes), 1356
Template (class in string), 81
template (string.Template attribute), 81
temporary
  file, 280
  file name, 280
TemporaryDirectory() (in module tempfile), 280
TemporaryFile() (in module tempfile), 280
teredo (ipaddress.IPv6Address attribute), 957
TERM, 529, 530
termattr() (in module curses), 530
term_size (class in os), 398
terminate() (multiprocessing.pool.Pool method), 615
terminate() (multiprocessing.Process method), 599
terminate() (subprocess.Popen method), 655
termios (module), 1352
termmame() (in module curses), 530
test (doctest.DocTestFailure attribute), 1098
test (doctest.UnexpectedException attribute), 1098
test (module), 1173
test() (in module cgi), 850
test.support (module), 1175
TestCase (class in unittest), 1106
TestFailed, 1176
testfile() (in module doctest), 1088
TESTFN (in module test.support), 1176
TestLoader (class in unittest), 1114
testMethodPrefix (unittest.TestLoader attribute), 1115
testmod() (in module doctest), 1089
TestResult (class in unittest), 1116
tests (in module imghdr), 977
testsuite() (in module doctest), 1097
testsRun (unittest.TestResult attribute), 1116
TestSuite (class in unittest), 1113
testzip() (zipfile.ZipFile method), 343
text (in module msilib), 1336
text (xml.etree.ElementTree.Element attribute), 806
text mode, 16
text() (msilib.Dialog method), 1335
text._factory (sqlite3.Connection attribute), 318
Textbox (class in curses.textpad), 540
TextCalendar (class in calendar), 163
textdomain() (in module gettext), 983
textinput() (in module turtle), 1023
textIOBase (class in io), 429
textIOWrapper (class in io), 429
textTestResult (class in unittest), 1117
textTestRunner (class in unittest), 1117
textwrap (module), 107
TextWrapper (class in textwrap), 108
theme_create() (tkinter.ttk.Style method), 1065
theme_names() (tkinter.ttk.Style method), 1066
theme_settings() (tkinter.ttk.Style method), 1065
theme_use() (tkinter.ttk.Style method), 1066
THOUSEP (in module locale), 993
Thread (class in threading), 585
thread() (imaplib.IMAP4 method), 902
thread_info (in module sys), 1226
threading (module), 583
ThreadPoolExecutor (class in concurrent.futures), 644
threads
  POSIX, 669
throw (2to3 fixer), 1173
tigetflag() (in module curses), 530
tigetnum() (in module curses), 530
tigetstr() (in module curses), 530
TILDE (in module token), 1308
tilt() (in module turtle), 1015
tiltangle() (in module turtle), 1016
time (class in datetime), 151
time (module), 431
time() (datetime.datetime method), 147
time() (module time), 437
Time2Internaldate() (in module imaplib), 898
timedelta (class in datetime), 138
TimedRotatingFileHandler (class in logging.handlers), 517
timegm() (module calendar), 165
timeit (module), 1207
timeit command line option
  -c, –clock, 1209
  -h, –help, 1209
  -n N, –number=N, 1209
  -p, –process, 1209
  -r N, –repeat=N, 1209
  -s S, –setup=S, 1209
  -t, –time, 1209
  -v, –verbose, 1209
timeit() (in module timeit), 1208
timeit() (timeit.Timer method), 1208
timeout, 675
timeout (socketserver.BaseServer attribute), 923
timeout (subprocess.TimeoutExpired attribute), 650
timeout() (curses.window method), 536
TIMEOUT_MAX (in module _thread), 669
TIMEOUT_MAX (in module threading), 584
TimeoutError, 70, 600
TimeoutExpired, 650
Timer (class in threading), 592
Timer (class in timeit), 1208
times() (in module os), 416
timestamp() (datetime.datetime method), 148
timetuple() (datetime.datetime method), 142
timetuple() (datetime.datetime method), 148
timetz() (datetime.datetime method), 147
timezone (class in datetime), 138, 159
timezone (module time), 437
title() (in module turtle), 1026
title() (str method), 43
tix, 1066
tix_additmapdir() (tkinter.tix.tixCommand method), 1070
tix_cget() (tkinter.tix.tixCommand method), 1070
tix_configure() (tkinter.tix.tixCommand method), 1070
tix_filedialog() (tkinter.tix.tixCommand method), 1070
tix_getbitmap() (tkinter.tix.tixCommand method), 1070
tix_getimaget() (tkinter.tix.tixCommand method), 1070

1468 Index
trunc() (in module math), 27, 207
truncat() (in module os), 408
truncat() (io.IOBase method), 425
truth
  value, 25
truth() (in module operator), 262
try
  statement, 65
ttk, 1051
tty
  I/O control, 1352
tty (module), 1353
tylabel() (in module os), 397
tuple
  object, 34, 36
  tuple (built-in class), 36
tuple2st() (in module parser), 1298
tuple2params (2to3 fixer), 1173
turnoff_sigfpe() (in module fpectl), 1268
turnon_sigfpe() (in module fpectl), 1268
Turtle (class in turtle), 1026
turtle (module), 999
turtles() (in module turtle), 1025
TurtleScreen (class in turtle), 1026
turtlesize() (in module turtle), 1015
type, 1373
  Boolean, 6
  built-in function, 62
  object, 20
  operations on dictionary, 56
  operations on list, 34
type (optparse.Option attribute), 477
type (socket.socket attribute), 684
type (tarfile.TarInfo attribute), 350
type (urlib.request.Request attribute), 866
type() (built-in function), 19
TYPE_CHECKER (optparse.Option attribute), 487
typeahead() (in module curses), 530
typecode (array.array attribute), 188
typecodes (in module array), 188
TYPED_ACTIONS (optparse.Option attribute), 488
typed_subpart_iterator() (in module email.iterators), 751
TypeError, 68
types
  built-in, 25
  immutable sequence, 34
  module, 62
  mutable sequence, 34
  operations on integer, 28
  operations on mapping, 56
  operations on numeric, 27
  operations on sequence, 32, 34
types (2to3 fixer), 1173
types (module), 193
TYPES (optparse.Option attribute), 487
types_map (in module mimetypes), 786
types_map (mimetypes.MimeType attribute), 786
UNARY_INVERT (opcode), 1319
UNARY_NEGATIVE (opcode), 1319
UNARY_NOT (opcode), 1319
UNARY_POSITIVE (opcode), 1319
UnboundLocalError, 68
unbuffered I/O, 16
UNC paths
  and os.makedirs(), 403
unconsumed_tail (zlib.Decompress attribute), 331
unctrl() (in module curses), 530
unctrl() (in module curses.ascii), 543
Underflow (class in decimal), 230
undisplay (pdb command), 1199
undo() (in module turtle), 1007
undobufferentries() (in module turtle), 1019
undoc_header (cmd.Cmd attribute), 1033
unescape() (in module xml.sax.saxutils), 832
UnexpectedException, 1098
unexpectedSuccesses (unittest.TestResult attribute), 1116
unget_wch() (in module curses), 530
ungetch() (in module curses), 530
ungetch() (in module msvr), 1337
ungetmouse() (in module curses), 530
ungetwch() (in module msvr), 1337
unhexify() (in module binascii), 791
update() (mailbox.Mailbox method), 771
update() (mailbox.Maildir method), 772
update() (set method), 56
update() (trace.CoverageResults method), 1213
update_visible() (mailbox.BabylMessage method), 781
update_wrapper() (in module functools), 260
upper() (str method), 43
urandom() (in module os), 421
URL, 847, 884, 928
parsing, 878
relative, 878
url (xmlrpc.client.ProtocolError attribute), 947
url2pathname() (in module urllib.request), 864
urlcleanup() (in module urllib.request), 876
urldefrag() (in module urllib.parse), 881
urlencode() (in module urllib.parse), 883
URLError, 884
urlopen() (in module urllib.request), 863
URLopener (class in urllib.request), 876
urlparse() (in module urllib.parse), 878
urlretrieve() (in module urllib.request), 875
urlsafe_b64decode() (in module base64), 787
urlsafe_b64encode() (in module base64), 787
urlsplit() (in module urllib.parse), 880
urlunparse() (in module urllib.parse), 880
urn (uuid.UUID attribute), 919
USER, 524
effective id, 389
id, 390
id, setting, 392
user() (poplib.POP3 method), 896
USER_BASE (in module site), 1267
user_call() (bdb.Bdb method), 1191
user_exception() (bdb.Bdb method), 1191
user_line() (bdb.Bdb method), 1191
user_return() (bdb.Bdb method), 1191
USER_SITE (in module site), 1267
usercustomize
module, 1266
UserDict (class in collections), 178
UserList (class in collections), 179
USERNAME, 390, 525
username (email.headerregistry.Address attribute), 741
USERPROFILE, 270
userptr() (curses.panel.Panel method), 544
UserWarning, 70
USTAR_FORMAT (in module tarfile), 347
UTC, 432
utc (datetime.timezone attribute), 160
utcfromtimestamp() (datetime.datetime class method), 145
utcnow() (datetime.datetime class method), 145
utcfromtimestamp() (datetime.datetime method), 148
utcfromtimestamp() (datetime.time method), 153
utcfromtimestamp() (datetime.timezone method), 159
utcfromtimestamp() (datetime.timezone method), 154
utctimetuple() (datetime.datetime method), 148
untime() (in module os), 408
uu
module, 789
uu (module), 792
UUID (class in uuid), 918
uuid (module), 918
uuid1, 919
uuid1() (in module uuid), 919
uuid3, 920
uuid3() (in module uuid), 919
uuid4, 920
uuid4() (in module uuid), 920
uuid5, 920
uuid5() (in module uuid), 920
UuidCreate() (in module msilib), 1331
V
v4_int_to_packed() (module ipaddress), 963
v6_int_to_packed() (module ipaddress), 963
validator() (in module wsgiref.validate), 858
value
truth, 25
value (ctypes._SimpleCData attribute), 578
value (http.cookiejar.Cookie attribute), 941
value (http.cookies.Morsel attribute), 934
value (xml.dom.Attr attribute), 817
Value() (in module multiprocessing), 606
Value() (in module multiprocessing.sharedctypes), 607
Value() (multiprocessing.managers.SyncManager method), 610
value_decode() (http.cookies.BaseCookie method), 933
value_encode() (http.cookies.BaseCookie method), 933
ValueError, 69
valuerefs() (weakref.WeakValueDictionary method), 191
values
Boolean, 62
values() (dict method), 58
values() (email.message.Message method), 722
values() (mailbox.Mailbox method), 770
values() (types.MappingProxyType method), 195
ValuesView (class in collections.abc), 180
variant (uuid.UUID attribute), 919
vars() (built-in function), 20
VBAR (in module token), 1308
vbar (tkinter.scrolledtext.ScrolledText attribute), 1071
VBAREQUAL (in module token), 1308
Vec2D (class in turtle), 1027
venv (module), 1180
VERBOSE (in module re), 88
verbose (in module tabnanny), 1313
verbose (in module test.support), 1176
verify() (smtplib.SMTP method), 911
verify_mode (ssl.SSLContext attribute), 698
verify_request() (socketserver.BaseServer method), 924
version (email.headerregistry.MIMEVersionHeader attribute), 739
version (http.client.HTTPResponse attribute), 889
version (http.cookiejar.Cookie attribute), 941
version (in module module.curses), 537
version (in module module.marshal), 308
version (in module module.sqlite3), 313
version (in module module.sys), 1226
version (ipaddress.IPv4Address attribute), 955
version (ipaddress.IPv4Network attribute), 958
version (ipaddress.IPv6Address attribute), 956
version (ipaddress.IPv6Network attribute), 961
version (urlib.request.URLopener attribute), 876
version (uuid.UUID attribute), 919
version() (in module platform), 546
version_info (in module module.sqlite3), 313
version_info (in module module.sys), 1226
version_string() (http.server.BaseHTTPRequestHandler method), 930
vformat() (string.Formatter method), 74
view, 1373
virtual
      Environments, 1180
virtual machine, 1373
visit() (ast.NodeVisitor method), 1305
vline() (curses.window method), 536
VMSError, 69
voidcmd() (ftplib.FTP method), 893
volume (zipfile.ZipInfo attribute), 345
vonnizesvatiolate() (in module random), 241
W
W_OK (in module os), 400
wait() (in module concurrent.futures), 647
wait() (in module multiprocessing.connection), 617
wait() (in module os), 416
wait() (multiprocessing.poolAsyncResult method), 615
wait() (subprocess.Popen method), 654
wait() (threading.Barrier method), 592
wait() (threading.Condition method), 589
wait() (threading.Event method), 591
wait3() (in module os), 417
wait4() (in module os), 417
wait_for() (threading.Condition method), 589
waitid() (in module os), 416
waitpid() (in module os), 417
walk() (email.message.Message method), 725
walk() (in module ast), 1304
walk() (in module os), 409
walk_packages() (in module pkgutil), 1282
want (doctest.Example attribute), 1093
warn() (in module warnings), 1234
warn_explicit() (in module warnings), 1234
Warning, 70
warning() (in module logging), 502
warning() (logging.Logger method), 494
warning() (xml.sax.handler.ErrorHandler method), 831
warnings, 1231
warnings (module), 1231
WarningsRecorder (class in test.support), 1180
warnoptions (in module sys), 1227
wasSuccessful() (unittest.TestResult method), 1116
WatchedFileHandler (class in logging.handlers), 516
wave (module), 972
WCONTINUED (in module os), 418
WCOREDUMP() (in module os), 418
WeakKeyDictionary (class in weakref), 191
weakref (module), 190
WeakSet (class in weakref), 191
WeakValueDictionary (class in weakref), 191
webbrowser (module), 845
weekday() (datetime.date method), 142
weekday() (datetime.datetime method), 149
weekday() (in module calendar), 164
weekheader() (in module calendar), 164
weibullviate() (in module random), 242
WEXITED (in module os), 417
WEXITSTATUS() (in module os), 418
while (http.server.BaseHTTPRequestHandler attribute), 929
what() (in module imghdr), 976
what() (in module sndhdr), 977
whatdir() (in module sndhdr), 977
whatis (pdb command), 1199
where (pdb command), 1197
which() (in module shutil), 288
whichdb() (in module dbm), 308
whereable (thread), 25
while statement, 25
whitespace (in module string), 74
whitespace (shlex.shlex attribute), 1038
whitespace_split (shlex.shlex attribute), 1038
Widget (class in tkinter.ttk), 1053
width (textwrap.TextWrapper attribute), 109
width() (in module turtle), 1010
WIFCONTINUED() (in module os), 418
WIFEXITED() (in module os), 418
WIFSIGNALED() (in module os), 418
WIFSTOPPED() (in module os), 418
win32_ver() (in module platform), 546
WinDLL (class in ctypes), 570
window manager (widgets), 1048
window() (curses.panel.Panel method), 544
window_height() (in module turtle), 1025
window_width() (in module turtle), 1025
Windows ini file, 360
WindowsError, 69
WindowsRegistryFinder (class in importlib.machinery), 1292
WinError() (in module ctypes), 576
WINFUNCTYPE() (in module ctypes), 573
winreg (module), 1337
WinSock, 664
winsound (module), 1345
winver (in module sys), 1227
WITH_CLEANUP (opcode), 1321
with_hostmask (ipaddress.IPv4Interface attribute), 963
with_hostmask (ipaddress.IPv4Network attribute), 959
with_hostmask (ipaddress.IPv6Interface attribute), 963
with_hostmask (ipaddress.IPv6Network attribute), 961
with_netmask (ipaddress.IPv4Interface attribute), 963
with_netmask (ipaddress.IPv4Network attribute), 959
with_netmask (ipaddress.IPv6Interface attribute), 963
with_netmask (ipaddress.IPv6Network attribute), 961
with_prefixlen (ipaddress.IPv4Interface attribute), 963
with_prefixlen (ipaddress.IPv4Network attribute), 959
with_prefixlen (ipaddress.IPv6Interface attribute), 963
with_prefixlen (ipaddress.IPv6Network attribute), 961
with_traceback() (BaseException method), 66
WOHANG (in module os), 417
NOWAIT (in module os), 417
wordchars (shlex.shlex attribute), 1038
World Wide Web, 845, 878, 884
wrap() (in module textwrap), 107
wrap() (textwrap.TextWrapper method), 110
wrap_socket() (in module ssl), 690
wrap_socket() (ssl.SSLContext method), 697
wrapper() (in module curses), 531
wraps() (in module functools), 260
wrappable (asyncio.dispatcher method), 706
writable () (io.IOBase method), 425
write () (code.InteractiveInterpreter method), 1272
write () (codecs.StreamWriter method), 127
write () (configparser.ConfigParser method), 374
write () (email.generator.BytesGenerator method), 730
write () (email.generator.Generator method), 729
write () (in module mmap), 718
write () (in module os), 397
write () (in module turtle), 1013
write () (io.BufferedReader method), 426
write () (io.BufferedWriter method), 428
write () (io.RawIOBase method), 425
write () (io.TextIOBase method), 429
write () (ossaudiodev.oss_audio_device method), 978
write () (xml.etree.ElementTree.ElementTree method), 808
write () (zipfile.ZipFile method), 343
write_byte() (in module mmap), 718
write_bytecode() (importlib.abc.PyPycLoader method), 1292
write_docstringdict() (in module turtle), 1028
write_history_file() (in module readline), 114
write_results() (trace.CoverageResults method), 1213
writeall() (ossaudiodev.oss_audio_device method), 978
writeframes() (aifc.aifc method), 970
writeframes() (sunau_.AU_write method), 972
writeframes() (wave.Wave_write method), 974
writeframesraw() (aifc.aifc method), 970
writeframesraw() (sunau_.AU_write method), 972
writeframesraw() (wave.Wave_write method), 974
writeheader() (csv.DictWriter method), 359
writelines() (codecs.StreamWriter method), 128
writelines() (io.IOBase method), 425
writePlist() (in module plistlib), 380
writePlistToBytes() (in module plistlib), 380
writepy() (zipfile.PyZipFile method), 344
writer (formatter.formatter attribute), 1327
writer() (in module csv), 356
writerow() (csv.csvwriter method), 359
writerows() (csv.csvwriter method), 359
writev() (in module os), 397
writexml() (xml.dom.minidom.Node method), 821
WrongDocumentErr, 819
ws_comma (2to3 fixer), 1173
wsgi_file_wrapper (wsgiref.handlers.BaseHandler attribute), 861
wsgi_multiprocess (wsgiref.handlers.BaseHandler attribute), 860
wsgi_multithread (wsgiref.handlers.BaseHandler attribute), 860
wsgi_run_once (wsgiref.handlers.BaseHandler attribute), 860
wsgiref (module), 854
wsgiref.handlers (module), 859
wsgiref.headers (module), 856
wsgiref.simple_server (module), 857
wsgiref.util (module), 854
wsgiref.validate (module), 858
WSGIRequestHandler (class in wsgiref.simple_server), 857
WSGIServer (class in wsgiref.simple_server), 857
wShowWindow (subprocess.STARTUPINFO attribute), 656
WSTOPPED (in module os), 417
WSTOPSIG() (in module os), 418
wstring_at() (in module ctypes), 577
WTERMSIG() (in module os), 418
WUNTRACED (in module os), 418
WWW, 845, 878, 884
server, 847, 928

X
X (in module re), 88
X509 certificate, 698
X_OK (in module os), 400
xatom() (imaplib.IMAP4 method), 902
XATTR_CREATE (in module os), 411
XATTR_REPLACE (in module os), 411
XATTR_SIZE_MAX (in module os), 411
xcor() (in module turtle), 1008
XDR, 377
xdrlib (module), 377
xhdr() (nntplib.NNTP method), 908
XHTML, 793
XHTML_NAMESPACE (in module xml.dom), 812
xml (module), 798
xml.etree.ElementTree (module), 805
xml.dom (module), 810
xml.dom.minidom (module), 820
xml.dom.pulldom (module), 824
xml.etree.ElementTree (module), 799
xml.parsers.expat (module), 836
xml.parsers.expat.errors (module), 843
xml.parsers.expat.model (module), 842
xml.sax (module), 826
xml.sax.handler (module), 827
xml.sax.saxutils (module), 832
xml.sax.xmlreader (module), 833
XML_ERROR_ABTERTED (in module
xml.parsers.expat.errors), 844
XML_ERROR_ASYNC_ENTITY (in module
xml.parsers.expat.errors), 843
XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF
(xml.parsers.expat.errors), 843
XML_ERROR_BAD_CHAR_REF (in module
xml.parsers.expat.errors), 843
XML_ERROR_BINARY_ENTITY_REF (in module
xml.parsers.expat.errors), 843
XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING
(xml.parsers.expat.errors), 844
XML_ERROR_DUPPLICATE_ATTRIBUTE (in module
xml.parsers.expat.errors), 843
XML_ERROR_ENTITY_DECARED_IN_PE (in module
xml.parsers.expat.errors), 844
XML_ERROR_EXTERNAL_ENTITY_HANDLING (in module
xml.parsers.expat.errors), 844
XML_ERROR_FEATURE_REQUIRES_XML_DTD (in module
xml.parsers.expat.errors), 844
XML_ERROR_FINISHED (in module
xml.parsers.expat.errors), 844
XML_ERROR_GONE (in module
xml.parsers.expat.errors), 844
XML_ERROR_INCOMPLETE_PE (in module
xml.parsers.expat.errors), 844
XML_ERROR_INCORRECT_ENCODING (in module
xml.parsers.expat.errors), 843
XML_ERROR_INVALID_TOKEN (in module
xml.parsers.expat.errors), 843
XML_ERROR_JUNK_AFTER_DOC_ELEMENT (in module
xml.parsers.expat.errors), 843
XML_ERROR_MISPLACED_XML_PI (in module
xml.parsers.expat.errors), 843
XML_ERROR_MISSING_PI (in module
xml.parsers.expat.errors), 843
XML_ERROR_NO_ELEMENTS (in module
xml.parsers.expat.errors), 843
XML_ERROR_NO_MEMORY (in module
xml.parsers.expat.errors), 843
XML_ERROR_NOT_STANDALONE (in module
xml.parsers.expat.errors), 844
XML_ERROR_NOT_SUSPENDED (in module
xml.parsers.expat.errors), 844
XML_ERROR_PARAM_ENTITY_REF (in module
xml.parsers.expat.errors), 843
XML_ERROR_PARTIAL_CHARS_RECOGNIZED (in module
xml.parsers.expat.errors), 844
XML_ERROR_PUBLICID (in module
xml.parsers.expat.errors), 844
XML_ERROR_RECURSIVE_ENTITY_REF (in module
xml.parsers.expat.errors), 843
XML_ERROR_SUSPEND_PE (in module
xml.parsers.expat.errors), 844
XML_ERROR_SUSPENDED (in module
xml.parsers.expat.errors), 844
XML_ERROR_SYNTAX (in module
xml.parsers.expat.errors), 843
XML_ERROR_TEXT_DECL (in module
xml.parsers.expat.errors), 844
XML_ERROR_UNBOUND_PREFIX (in module
xml.parsers.expat.errors), 844
XML_ERROR_UNCLOSED_CDATA_SECTION (in module
xml.parsers.expat.errors), 844
XML_ERROR_UNCLOSED_TOKEN (in module
xml.parsers.expat.errors), 843
XML_ERROR_UNDECLARING_PREFIX (in module
xml.parsers.expat.errors), 844
XML_ERROR_UNDEFINED_ENTITY (in module
xml.parsers.expat.errors), 843
XML_ERROR_UNEXPECTED_STATE (in module
xml.parsers.expat.errors), 844
XML_ERROR_UNKNOWN_ENCODING (in module
xml.parsers.expat.errors), 844
XML_ERROR_XML_DECL (in module
xml.parsers.expat.errors), 844
XML_NAMESPACE (in module xml.dom), 811
xmlcharrefreplace_errors() (in module codecs), 123
XmlDeclHandler() (xml.parsers.expat.xmlparser
method), 839
XMLFilterBase (class in xml.sax.saxutils), 832
XMLGenerator (class in xml.sax.saxutils), 832
XMLID() (in module xml.etree.ElementTree), 805
XMLNS_NAMESPACE (in module xml.dom), 812
XMLParser (class in xml.etree.ElementTree), 809
XMLParserType (in module xml.parsers.expat), 837
XMLReader (class in xml.sax.xmlreader), 833
xmlrpc.client (module), 943
xmlrpc.server (module), 950
xor() (in module operator), 263
xover() (nttplib.NNTP method), 908
xpath() (nttplib.NNTP method), 908
xrange (2to3 fixer), 1173
xreadlines (2to3 fixer), 1173

Index 1475
xview() (tkinter.ttk.Treeview method), 1063

Y

Y2K, 432
ycor() (in module turtle), 1009
year (datetime.date attribute), 142
year (datetime.datetime attribute), 146
Year 2000, 432
Year 2038, 432
yeardatescalendar() (calendar.Calendar method), 163
yeardays2calendar() (calendar.Calendar method), 163
yeardayscalendar() (calendar.Calendar method), 163
YESEXPR (in module locale), 993
YIELD_FROM (opcode), 1321
YIELD_VALUE (opcode), 1321
yiq_to_rgb() (in module colorsys), 976
yview() (tkinter.ttk.Treeview method), 1063

Z

Zen of Python, 1373
ZeroDivisionError, 69
zfill() (str method), 43
zip (2to3 fixer), 1173
zip() (built-in function), 20
ZIP_BZIP2 (in module zipfile), 341
ZIP_DEFLATED (in module zipfile), 341
zip_longest() (in module itertools), 254
ZIP_LZMA (in module zipfile), 341
ZIP_STORED (in module zipfile), 341
ZipFile (class in zipfile), 342
zipfile (module), 340
zipimport (module), 1279
zipimporter (class in zipimport), 1279
ZipImportError, 1279
ZipInfo (class in zipfile), 341
zlib (module), 329
ZLIB_RUNTIME_VERSION (in module zlib), 332
ZLIB_VERSION (in module zlib), 331