## CONTENTS

1 Introduction 3
   1.1 Best case: trivial installation .................................. 3
   1.2 The new standard: Distutils .................................. 3

2 Standard Build and Install 5
   2.1 Platform variations ........................................... 5
   2.2 Splitting the job up ........................................... 5
   2.3 How building works ........................................... 6
   2.4 How installation works ........................................ 6

3 Alternate Installation 9
   3.1 Alternate installation: the user scheme ......................... 9
   3.2 Alternate installation: the home scheme ........................ 10
   3.3 Alternate installation: Unix (the prefix scheme) ............... 10
   3.4 Alternate installation: Windows (the prefix scheme) .......... 11

4 Custom Installation 13
   4.1 Modifying Python’s Search Path ................................ 15

5 Distutils Configuration Files 17
   5.1 Location and names of config files ........................... 17
   5.2 Syntax of config files ......................................... 18

6 Building Extensions: Tips and Tricks 19
   6.1 Tweaking compiler/linker flags ................................ 19
   6.2 Using non-Microsoft compilers on Windows ...................... 20

A Glossary 23

B About these documents 33
   B.1 Contributors to the Python Documentation ...................... 33

C History and License 35
   C.1 History of the software ....................................... 35
   C.2 Terms and conditions for accessing or otherwise using Python .... 35
   C.3 Licenses and Acknowledgements for Incorporated Software ...... 38

D Copyright 49

Index 51
Author  Greg Ward

Abstract
This document describes the Python Distribution Utilities ("Distutils") from the end-user’s point-of-view, describing how to extend the capabilities of a standard Python installation by building and installing third-party Python modules and extensions.
Although Python’s extensive standard library covers many programming needs, there often comes a time when you need to add some new functionality to your Python installation in the form of third-party modules. This might be necessary to support your own programming, or to support an application that you want to use and that happens to be written in Python.

In the past, there has been little support for adding third-party modules to an existing Python installation. With the introduction of the Python Distribution Utilities (Distutils for short) in Python 2.0, this changed.

This document is aimed primarily at the people who need to install third-party Python modules: end-users and system administrators who just need to get some Python application running, and existing Python programmers who want to add some new goodies to their toolbox. You don’t need to know Python to read this document; there will be some brief forays into using Python’s interactive mode to explore your installation, but that’s it. If you’re looking for information on how to distribute your own Python modules so that others may use them, see the *distutils-index* manual.

### 1.1 Best case: trivial installation

In the best case, someone will have prepared a special version of the module distribution you want to install that is targeted specifically at your platform and is installed just like any other software on your platform. For example, the module developer might make an executable installer available for Windows users, an RPM package for users of RPM-based Linux systems (Red Hat, SuSE, Mandrake, and many others), a Debian package for users of Debian-based Linux systems, and so forth.

In that case, you would download the installer appropriate to your platform and do the obvious thing with it: run it if it’s an executable installer, `rpm --install` it if it’s an RPM, etc. You don’t need to run Python or a setup script, you don’t need to compile anything—you might not even need to read any instructions (although it’s always a good idea to do so anyway).

Of course, things will not always be that easy. You might be interested in a module distribution that doesn’t have an easy-to-use installer for your platform. In that case, you’ll have to start with the source distribution released by the module’s author/maintainer. Installing from a source distribution is not too hard, as long as the modules are packaged in the standard way. The bulk of this document is about building and installing modules from standard source distributions.

### 1.2 The new standard: Distutils

If you download a module source distribution, you can tell pretty quickly if it was packaged and distributed in the standard way, i.e. using the Distutils. First, the distribution’s name and version number will be featured prominently in the name of the downloaded archive, e.g. `foo-1.0.tar.gz` or `widget-0.9.7.zip`. Next, the archive will unpack into a similarly-named directory: `foo-1.0` or `widget-0.9.7`. Additionally, the distribution will contain a setup script `setup.py`, and a file named `README.txt` or possibly just `README`, which should explain that building and installing the module distribution is a simple matter of running one command from a terminal:
python setup.py install

For Windows, this command should be run from a command prompt window (Start → Accessories):

setup.py install

If all these things are true, then you already know how to build and install the modules you’ve just downloaded: Run the command above. Unless you need to install things in a non-standard way or customize the build process, you don’t really need this manual. Or rather, the above command is everything you need to get out of this manual.
STANDARD BUILD AND INSTALL

As described in section *The new standard: Distutils*, building and installing a module distribution using the Distutils is usually one simple command to run from a terminal:

```
python setup.py install
```

### 2.1 Platform variations

You should always run the setup command from the distribution root directory, i.e. the top-level subdirectory that the module source distribution unpacks into. For example, if you’ve just downloaded a module source distribution `foo-1.0.tar.gz` onto a Unix system, the normal thing to do is:

```
gunzip -c foo-1.0.tar.gz | tar xf -  # unpacks into directory foo-1.0
cd foo-1.0
python setup.py install
```

On Windows, you’d probably download `foo-1.0.zip`. If you downloaded the archive file to `C:\Temp`, then it would unpack into `C:\Temp\foo-1.0`; you can use either a archive manipulator with a graphical user interface (such as WinZip) or a command-line tool (such as `unzip` or `pkunzip`) to unpack the archive. Then, open a command prompt window and run:

```
cd c:\Temp\foo-1.0
python setup.py install
```

### 2.2 Splitting the job up

Running `setup.py install` builds and installs all modules in one run. If you prefer to work incrementally—especially useful if you want to customize the build process, or if things are going wrong—you can use the setup script to do one thing at a time. This is particularly helpful when the build and install will be done by different users—for example, you might want to build a module distribution and hand it off to a system administrator for installation (or do it yourself, with super-user privileges).

For example, you can build everything in one step, and then install everything in a second step, by invoking the setup script twice:

```
python setup.py build
python setup.py install
```

If you do this, you will notice that running the `install` command first runs the `build` command, which—in this case—quickly notices that it has nothing to do, since everything in the `build` directory is up-to-date.

You may not need this ability to break things down often if all you do is install modules downloaded off the ‘net, but it’s very handy for more advanced tasks. If you get into distributing your own Python modules and extensions, you’ll run lots of individual Distutils commands on their own.
2.3 How building works

As implied above, the build command is responsible for putting the files to install into a build directory. By default, this is build under the distribution root; if you’re excessively concerned with speed, or want to keep the source tree pristine, you can change the build directory with the --build-base option. For example:

```bash
python setup.py build --build-base=/path/to/pybuild/foo-1.0
```

(Or you could do this permanently with a directive in your system or personal Distutils configuration file; see section Distutils Configuration Files.) Normally, this isn’t necessary.

The default layout for the build tree is as follows:

```
--- build/ --- lib/
or
--- build/ --- lib.<plat>/
  temp.<plat>/
```

where <plat> expands to a brief description of the current OS/hardware platform and Python version. The first form, with just a lib directory, is used for “pure module distributions”—that is, module distributions that include only pure Python modules. If a module distribution contains any extensions (modules written in C/C++), then the second form, with two <plat> directories, is used. In that case, the temp.<plat> directory holds temporary files generated by the compile/link process that don’t actually get installed. In either case, the lib (or lib.<plat>) directory contains all Python modules (pure Python and extensions) that will be installed.

In the future, more directories will be added to handle Python scripts, documentation, binary executables, and whatever else is needed to handle the job of installing Python modules and applications.

2.4 How installation works

After the build command runs (whether you run it explicitly, or the install command does it for you), the work of the install command is relatively simple: all it has to do is copy everything under build/lib (or build/lib.<plat>) to your chosen installation directory.

If you don’t choose an installation directory—i.e., if you just run setup.py install—then the install command installs to the standard location for third-party Python modules. This location varies by platform and by how you built/installed Python itself. On Unix (and Mac OS X, which is also Unix-based), it also depends on whether the module distribution being installed is pure Python or contains extensions (“non-pure”):

<table>
<thead>
<tr>
<th>Platform</th>
<th>Standard installation location</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unix (pure)</td>
<td>prefix/lib/pythonX.Y/site-packages</td>
<td>/usr/local/lib/pythonX.Y/site-packages</td>
</tr>
<tr>
<td>Unix (non-pure)</td>
<td>exec-prefix/lib/pythonX.Y/site-packages</td>
<td>/usr/local/lib/pythonX.Y/site-packages</td>
</tr>
<tr>
<td>Windows</td>
<td>prefix\Lib\site-packages</td>
<td>C:\PythonXY\Lib\site-packages</td>
</tr>
</tbody>
</table>

Notes:

1. Most Linux distributions include Python as a standard part of the system, so prefix and exec-prefix are usually both /usr on Linux. If you build Python yourself on Linux (or any Unix-like system), the default prefix and exec-prefix are /usr/local.

2. The default installation directory on Windows was C:\Program Files\Python under Python 1.6a1, 1.5.2, and earlier.

prefix and exec-prefix stand for the directories that Python is installed to, and where it finds its libraries at run-time. They are always the same under Windows, and very often the same under Unix and Mac OS X. You can find out what your Python installation uses for prefix and exec-prefix by running Python in interactive mode and typing a few simple commands. Under Unix, just type python at the shell prompt. Under Windows, choose Start → Programs → Python X.Y → Python (command line). Once the interpreter is started, you type Python code at the prompt. For example, on my Linux system, I type the three Python statements shown below, and get the output as shown, to find out my prefix and exec-prefix:
A few other placeholders are used in this document: X.Y stands for the version of Python, for example 3.2; `abiflags` will be replaced by the value of `sys.abiflags` or the empty string for platforms which don’t define ABI flags; `distname` will be replaced by the name of the module distribution being installed. Dots and capitalization are important in the paths; for example, a value that uses `python3.2` on UNIX will typically use `Python32` on Windows.

If you don’t want to install modules to the standard location, or if you don’t have permission to write there, then you need to read about alternate installations in section `Alternate Installation`. If you want to customize your installation directories more heavily, see section `Custom Installation` on custom installations.
Often, it is necessary or desirable to install modules to a location other than the standard location for third-party Python modules. For example, on a Unix system you might not have permission to write to the standard third-party module directory. Or you might wish to try out a module before making it a standard part of your local Python installation. This is especially true when upgrading a distribution already present: you want to make sure your existing base of scripts still works with the new version before actually upgrading.

The Distutils `install` command is designed to make installing module distributions to an alternate location simple and painless. The basic idea is that you supply a base directory for the installation, and the `install` command picks a set of directories (called an `installation scheme`) under this base directory in which to install files. The details differ across platforms, so read whichever of the following sections applies to you.

Note that the various alternate installation schemes are mutually exclusive: you can pass `--user`, or `--home`, or `--prefix` and `--exec-prefix`, or `--install-base` and `--install-platbase`, but you can’t mix from these groups.

### 3.1 Alternate installation: the user scheme

This scheme is designed to be the most convenient solution for users that don’t have write permission to the global site-packages directory or don’t want to install into it. It is enabled with a simple option:

```
python setup.py install --user
```

Files will be installed into subdirectories of `site.USER_BASE` (written as `userbase` hereafter). This scheme installs pure Python modules and extension modules in the same location (also known as `site.USER_SITE`). Here are the values for UNIX, including Mac OS X:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Installation directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>modules</td>
<td><code>userbase/lib/pythonX.Y/site-packages</code></td>
</tr>
<tr>
<td>scripts</td>
<td><code>userbase/bin</code></td>
</tr>
<tr>
<td>data</td>
<td><code>userbase</code></td>
</tr>
<tr>
<td>C headers</td>
<td><code>userbase/include/pythonX.Yabiflags/distname</code></td>
</tr>
</tbody>
</table>

And here are the values used on Windows:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Installation directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>modules</td>
<td><code>userbase\PythonXY\site-packages</code></td>
</tr>
<tr>
<td>scripts</td>
<td><code>userbase\Scripts</code></td>
</tr>
<tr>
<td>data</td>
<td><code>userbase</code></td>
</tr>
<tr>
<td>C headers</td>
<td><code>userbase\PythonXY\Include\distname</code></td>
</tr>
</tbody>
</table>

The advantage of using this scheme compared to the other ones described below is that the user site-packages directory is under normal conditions always included in `sys.path` (see `site` for more information), which means that there is no additional step to perform after running the `setup.py` script to finalize the installation.

The `build_ext` command also has a `--user` option to add `userbase/include` to the compiler search path for header files and `userbase\lib` to the compiler search path for libraries as well as to the runtime search path for shared C libraries (rpath).
3.2 Alternate installation: the home scheme

The idea behind the “home scheme” is that you build and maintain a personal stash of Python modules. This scheme’s name is derived from the idea of a “home” directory on Unix, since it’s not unusual for a Unix user to make their home directory have a layout similar to /usr/ or /usr/local/. This scheme can be used by anyone, regardless of the operating system they are installing for.

Installing a new module distribution is as simple as

```
python setup.py install --home=<dir>
```

where you can supply any directory you like for the `--home` option. On Unix, lazy typists can just type a tilde (~); the `install` command will expand this to your home directory:

```
python setup.py install --home=~
```

To make Python find the distributions installed with this scheme, you may have to modify Python’s search path or edit `sitecustomize` (see `site`) to call `site.addsitedir()` or edit `sys.path`.

The `--home` option defines the installation base directory. Files are installed to the following directories under the installation base as follows:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Installation directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>modules</td>
<td>home/lib/python</td>
</tr>
<tr>
<td>scripts</td>
<td>home/bin</td>
</tr>
<tr>
<td>data</td>
<td>home</td>
</tr>
<tr>
<td>C headers</td>
<td>home/include/python/distname</td>
</tr>
</tbody>
</table>

(Mentally replace slashes with backslashes if you’re on Windows.)

3.3 Alternate installation: Unix (the prefix scheme)

The “prefix scheme” is useful when you wish to use one Python installation to perform the build/install (i.e., to run the setup script), but install modules into the third-party module directory of a different Python installation (or something that looks like a different Python installation). If this sounds a trifle unusual, it is—that’s why the user and home schemes come before. However, there are at least two known cases where the prefix scheme will be useful.

First, consider that many Linux distributions put Python in /usr, rather than the more traditional /usr/local. This is entirely appropriate, since in those cases Python is part of “the system” rather than a local add-on. However, if you are installing Python modules from source, you probably want them to go in /usr/local/lib/python2.X rather than /usr/lib/python2.X. This can be done with

```
/usr/bin/python setup.py install --prefix=/usr/local
```

Another possibility is a network filesystem where the name used to write to a remote directory is different from the name used to read it: for example, the Python interpreter accessed as /usr/local/bin/python might search for modules in /usr/local/lib/python2.X, but those modules would have to be installed to, say, /mnt/@server/export/lib/python2.X. This could be done with

```
/usr/local/bin/python setup.py install --prefix=/mnt/@server/export
```

In either case, the `--prefix` option defines the installation base, and the `--exec-prefix` option defines the platform-specific installation base, which is used for platform-specific files. (Currently, this just means non-pure module distributions, but could be expanded to C libraries, binary executables, etc.) If `--exec-prefix` is not supplied, it defaults to `--prefix`. Files are installed as follows:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Installation directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python modules</td>
<td>prefix/lib/pythonX.Y/site-packages</td>
</tr>
<tr>
<td>extension modules</td>
<td>exec-prefix/lib/pythonX.Y/site-packages</td>
</tr>
<tr>
<td>scripts</td>
<td>prefix/bin</td>
</tr>
<tr>
<td>data</td>
<td>prefix</td>
</tr>
<tr>
<td>C headers</td>
<td>prefix/include/pythonX.Yabiflags/distname</td>
</tr>
</tbody>
</table>
There is no requirement that --prefix or --exec-prefix actually point to an alternate Python installation; if the directories listed above do not already exist, they are created at installation time.

Incidentally, the real reason the prefix scheme is important is simply that a standard Unix installation uses the prefix scheme, but with --prefix and --exec-prefix supplied by Python itself as sys.prefix and sys.exec_prefix. Thus, you might think you’ll never use the prefix scheme, but every time you run python setup.py install without any other options, you’re using it.

Note that installing extensions to an alternate Python installation has no effect on how those extensions are built: in particular, the Python header files (Python.h and friends) installed with the Python interpreter used to run the setup script will be used in compiling extensions. It is your responsibility to ensure that the interpreter used to run extensions installed in this way is compatible with the interpreter used to build them. The best way to do this is to ensure that the two interpreters are the same version of Python (possibly different builds, or possibly copies of the same build). (Of course, if your --prefix and --exec-prefix don’t even point to an alternate Python installation, this is immaterial.)

### 3.4 Alternate installation: Windows (the prefix scheme)

Windows has no concept of a user’s home directory, and since the standard Python installation under Windows is simpler than under Unix, the --prefix option has traditionally been used to install additional packages in separate locations on Windows.

```bash
python setup.py install --prefix="\Temp\Python"
```

to install modules to the \Temp\Python directory on the current drive.

The installation base is defined by the --prefix option; the --exec-prefix option is not supported under Windows, which means that pure Python modules and extension modules are installed into the same location. Files are installed as follows:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Installation directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>modules</td>
<td>prefix\Lib\site-packages</td>
</tr>
<tr>
<td>scripts</td>
<td>prefix\Scripts</td>
</tr>
<tr>
<td>data</td>
<td>prefix</td>
</tr>
<tr>
<td>C headers</td>
<td>prefix\Include\distname</td>
</tr>
</tbody>
</table>
CHAPTER FOUR

CUSTOM INSTALLATION

Sometimes, the alternate installation schemes described in section Alternate Installation just don’t do what you want. You might want to tweak just one or two directories while keeping everything under the same base directory, or you might want to completely redefine the installation scheme. In either case, you’re creating a custom installation scheme.

To create a custom installation scheme, you start with one of the alternate schemes and override some of the installation directories used for the various types of files, using these options:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Override option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python modules</td>
<td>--install-purelib</td>
</tr>
<tr>
<td>extension modules</td>
<td>--install-platlib</td>
</tr>
<tr>
<td>all modules</td>
<td>--install-lib</td>
</tr>
<tr>
<td>scripts</td>
<td>--install-scripts</td>
</tr>
<tr>
<td>data</td>
<td>--install-data</td>
</tr>
<tr>
<td>C headers</td>
<td>--install-headers</td>
</tr>
</tbody>
</table>

These override options can be relative, absolute, or explicitly defined in terms of one of the installation base directories. (There are two installation base directories, and they are normally the same—they only differ when you use the Unix “prefix scheme” and supply different --prefix and --exec-prefix options; using --install-lib will override values computed or given for --install-purelib and --install-platlib, and is recommended for schemes that don’t make a difference between Python and extension modules.)

For example, say you’re installing a module distribution to your home directory under Unix—but you want scripts to go in ~/scripts rather than ~/bin. As you might expect, you can override this directory with the --install-scripts option; in this case, it makes most sense to supply a relative path, which will be interpreted relative to the installation base directory (your home directory, in this case):

```bash
python setup.py install --home=~ --install-scripts=scripts
```

Another Unix example: suppose your Python installation was built and installed with a prefix of /usr/local/python, so under a standard installation scripts will wind up in /usr/local/python/bin. If you want them in /usr/local/bin instead, you would supply this absolute directory for the --install-scripts option:

```bash
python setup.py install --install-scripts=/usr/local/bin
```

(This performs an installation using the “prefix scheme,” where the prefix is whatever your Python interpreter was installed with—/usr/local/python in this case.)

If you maintain Python on Windows, you might want third-party modules to live in a subdirectory of prefix, rather than right in prefix itself. This is almost as easy as customizing the script installation directory—you just have to remember that there are two types of modules to worry about, Python and extension modules, which can conveniently be both controlled by one option:

```bash
python setup.py install --install-lib=Site
```

The specified installation directory is relative to prefix. Of course, you also have to ensure that this directory is in Python’s module search path, such as by putting a .pth file in a site directory (see site). See section Modifying Python’s Search Path to find out how to modify Python’s search path.
If you want to define an entire installation scheme, you just have to supply all of the installation directory options. The recommended way to do this is to supply relative paths; for example, if you want to maintain all Python module-related files under `python` in your home directory, and you want a separate directory for each platform that you use your home directory from, you might define the following installation scheme:

```bash
python setup.py install --home=~ \
  --install-purelib=python/lib \
  --install-platlib=python/lib.$PLAT \
  --install-scripts=python/scripts \
  --install-data=python/data
```

or, equivalently,

```bash
python setup.py install --home=/python \
  --install-purelib=lib \
  --install-platlib='lib.$PLAT' \
  --install-scripts=scripts \
  --install-data=data
```

$PLAT is not (necessarily) an environment variable—it will be expanded by the Distutils as it parses your command line options, just as it does when parsing your configuration file(s).

Obviously, specifying the entire installation scheme every time you install a new module distribution would be very tedious. Thus, you can put these options into your Distutils config file (see section Distutils Configuration Files):

```ini
[install]
install-base=$HOME
install-purelib=python/lib
install-platlib=python/lib.$PLAT
install-scripts=python/scripts
install-data=python/data
```

or, equivalently,

```ini
[install]
install-base=$HOME/python
install-purelib=lib
install-platlib=lib.$PLAT
install-scripts=scripts
install-data=data
```

Note that these two are not equivalent if you supply a different installation base directory when you run the setup script. For example,

```bash
python setup.py install --install-base=/tmp
```

would install pure modules to `/tmp/python/lib` in the first case, and to `/tmp/lib` in the second case. (For the second case, you probably want to supply an installation base of `/tmp/python`.)

You probably noticed the use of `$HOME` and `$PLAT` in the sample configuration file input. These are Distutils configuration variables, which bear a strong resemblance to environment variables. In fact, you can use environment variables in config files on platforms that have such a notion but the Distutils additionally define a few extra variables that may not be in your environment, such as `$PLAT`. (And of course, on systems that don’t have environment variables, such as Mac OS 9, the configuration variables supplied by the Distutils are the only ones you can use.) See section Distutils Configuration Files for details.

**Note:** When a virtual environment is activated, any options that change the installation path will be ignored from all distutils configuration files to prevent inadvertently installing projects outside of the virtual environment.
4.1 Modifying Python’s Search Path

When the Python interpreter executes an `import` statement, it searches for both Python code and extension modules along a search path. A default value for the path is configured into the Python binary when the interpreter is built. You can determine the path by importing the `sys` module and printing the value of `sys.path`.

```
$ python
[GCC 2.96 20000731 (Red Hat Linux 7.3 2.96-112)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
```

The null string in `sys.path` represents the current working directory.

The expected convention for locally installed packages is to put them in the `../site-packages/` directory, but you may want to install Python modules into some arbitrary directory. For example, your site may have a convention of keeping all software related to the web server under `/www`. Add-on Python modules might then belong in `/www/python`, and in order to import them, this directory must be added to `sys.path`. There are several different ways to add the directory.

The most convenient way is to add a path configuration file to a directory that’s already on Python’s path, usually to the `../site-packages/` directory. Path configuration files have an extension of `.pth`, and each line must contain a single path that will be appended to `sys.path`. (Because the new paths are appended to `sys.path`, modules in the added directories will not override standard modules. This means you can’t use this mechanism for installing fixed versions of standard modules.)

Paths can be absolute or relative, in which case they’re relative to the directory containing the `.pth` file. See the documentation of the `site` module for more information.

A slightly less convenient way is to edit the `site.py` file in Python’s standard library, and modify `sys.path`. `site.py` is automatically imported when the Python interpreter is executed, unless the `-S` switch is supplied to suppress this behaviour. So you could simply edit `site.py` and add two lines to it:

```
import sys
sys.path.append('/www/python/')
```

However, if you reinstall the same major version of Python (perhaps when upgrading from 2.2 to 2.2.2, for example) `site.py` will be overwritten by the stock version. You’d have to remember that it was modified and save a copy before doing the installation.

There are two environment variables that can modify `sys.path`.

`PYTHONHOME` sets an alternate value for the prefix of the Python installation. For example, if `PYTHONHOME` is set to `/www/python`, the search path will be set to `[", '/www/python/lib/pythonX.Y/’, '/www/python/lib/pythonX.Y/plat-linux2’, ...]`.

The `PYTHONPATH` variable can be set to a list of paths that will be added to the beginning of `sys.path`. For example, if `PYTHONPATH` is set to `/www/python:/opt/py`, the search path will begin with `[’/www/python’, ’/opt/py’]`. (Note that directories must exist in order to be added to `sys.path`; the `site` module removes paths that don’t exist.)

Finally, `sys.path` is just a regular Python list, so any Python application can modify it by adding or removing entries.
As mentioned above, you can use Distutils configuration files to record personal or site preferences for any Distutils options. That is, any option to any command can be stored in one of two or three (depending on your platform) configuration files, which will be consulted before the command-line is parsed. This means that configuration files will override default values, and the command-line will in turn override configuration files. Furthermore, if multiple configuration files apply, values from “earlier” files are overridden by “later” files.

5.1 Location and names of config files

The names and locations of the configuration files vary slightly across platforms. On Unix and Mac OS X, the three configuration files (in the order they are processed) are:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Location and filename</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td>prefix/lib/pythonver/distutils/distutils.cfg</td>
<td>(1)</td>
</tr>
<tr>
<td>personal</td>
<td>$HOME/.pydistutils.cfg</td>
<td>(2)</td>
</tr>
<tr>
<td>local</td>
<td>setup.cfg</td>
<td>(3)</td>
</tr>
</tbody>
</table>

And on Windows, the configuration files are:

<table>
<thead>
<tr>
<th>Type of file</th>
<th>Location and filename</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td>prefix\Lib\distutils\distutils.cfg</td>
<td>(4)</td>
</tr>
<tr>
<td>personal</td>
<td>%HOME%\pydistutils.cfg</td>
<td>(5)</td>
</tr>
<tr>
<td>local</td>
<td>setup.cfg</td>
<td>(3)</td>
</tr>
</tbody>
</table>

On all platforms, the “personal” file can be temporarily disabled by passing the \texttt{--no-user-cfg} option.

Notes:

1. Strictly speaking, the system-wide configuration file lives in the directory where the Distutils are installed; under Python 1.6 and later on Unix, this is as shown. For Python 1.5.2, the Distutils will normally be installed to \texttt{prefix/lib/python1.5/site-packages/distutils}, so the system configuration file should be put there under Python 1.5.2.

2. On Unix, if the \texttt{HOME} environment variable is not defined, the user’s home directory will be determined with the \texttt{getpwuid()} function from the standard \texttt{pwd} module. This is done by the \texttt{os.path.expanduser()} function used by Distutils.

3. I.e., in the current directory (usually the location of the setup script).

4. (See also note (1).) Under Python 1.6 and later, Python’s default “installation prefix” is \texttt{C:\Python}, so the system configuration file is normally \texttt{C:\Python\Lib\distutils\distutils.cfg}. Under Python 1.5.2, the default prefix was \texttt{C:\Program Files\Python}, and the Distutils were not part of the standard library—so the system configuration file would be \texttt{C:\Program Files\Python\distutils\distutils.cfg} in a standard Python 1.5.2 installation under Windows.

5. On Windows, if the \texttt{HOME} environment variable is not defined,
USERPROFILE then HOMEDRIVE and HOMEPATH will be tried. This is done by the os.path.expanduser() function used by Distutils.

5.2 Syntax of config files

The Distutils configuration files all have the same syntax. The config files are grouped into sections. There is one section for each Distutils command, plus a global section for global options that affect every command. Each section consists of one option per line, specified as option=value.

For example, the following is a complete config file that just forces all commands to run quietly by default:

```
[global]
verbose=0
```

If this is installed as the system config file, it will affect all processing of any Python module distribution by any user on the current system. If it is installed as your personal config file (on systems that support them), it will affect only module distributions processed by you. And if it is used as the setup.cfg for a particular module distribution, it affects only that distribution.

You could override the default “build base” directory and make the build* commands always forcibly rebuild all files with the following:

```
[build]
build-base=blib
force=1
```

which corresponds to the command-line arguments

```
python setup.py build --build-base=blib --force
```

except that including the build command on the command-line means that command will be run. Including a particular command in config files has no such implication; it only means that if the command is run, the options in the config file will apply. (Or if other commands that derive values from it are run, they will use the values in the config file.)

You can find out the complete list of options for any command using the --help option, e.g.:

```
python setup.py build --help
```

and you can find out the complete list of global options by using --help without a command:

```
python setup.py --help
```

See also the “Reference” section of the “Distributing Python Modules” manual.
Whenever possible, the Distutils try to use the configuration information made available by the Python interpreter used to run the setup.py script. For example, the same compiler and linker flags used to compile Python will also be used for compiling extensions. Usually this will work well, but in complicated situations this might be inappropriate. This section discusses how to override the usual Distutils behaviour.

### 6.1 Tweaking compiler/linker flags

Compiling a Python extension written in C or C++ will sometimes require specifying custom flags for the compiler and linker in order to use a particular library or produce a special kind of object code. This is especially true if the extension hasn’t been tested on your platform, or if you’re trying to cross-compile Python.

In the most general case, the extension author might have foreseen that compiling the extensions would be complicated, and provided a Setup file for you to edit. This will likely only be done if the module distribution contains many separate extension modules, or if they often require elaborate sets of compiler flags in order to work.

A Setup file, if present, is parsed in order to get a list of extensions to build. Each line in a Setup describes a single module. Lines have the following structure:

```
module ... [sourcefile ...] [cpparg ...] [library ...]
```

Let’s examine each of the fields in turn.

- **module** is the name of the extension module to be built, and should be a valid Python identifier. You can’t just change this in order to rename a module (edits to the source code would also be needed), so this should be left alone.

- **sourcefile** is anything that’s likely to be a source code file, at least judging by the filename. Filenames ending in `.c` are assumed to be written in C, filenames ending in `.C`, `.cc`, and `.cpp` are assumed to be C++, and filenames ending in `.m` or `.mm` are assumed to be in Objective C.

- **cpparg** is an argument for the C preprocessor, and is anything starting with `-I`, `-D`, `-U` or `-C`.

- **library** is anything ending in `.a` or beginning with `-l` or `-L`.

If a particular platform requires a special library on your platform, you can add it by editing the Setup file and running python setup.py build. For example, if the module defined by the line

```
foo foomodule.c
```

must be linked with the math library `libm.a` on your platform, simply add `-lm` to the line:

```
foo foomodule.c -lm
```

Arbitrary switches intended for the compiler or the linker can be supplied with the `-Xcompiler arg` and `-Xlinker arg` options:

```
foo foomodule.c -Xcompiler -o32 -Xlinker -shared -lm
```
The next option after \texttt{-Xcompiler} and \texttt{-Xlinker} will be appended to the proper command line, so in the above example the compiler will be passed the \texttt{-o32} option, and the linker will be passed \texttt{-shared}. If a compiler option requires an argument, you’ll have to supply multiple \texttt{-Xcompiler} options; for example, to pass \texttt{-x c++} the Setup file would have to contain \texttt{-Xcompiler -x -Xcompiler c++}.

Compiler flags can also be supplied through setting the \texttt{CFLAGS} environment variable. If set, the contents of \texttt{CFLAGS} will be added to the compiler flags specified in the Setup file.

### 6.2 Using non-Microsoft compilers on Windows

#### 6.2.1 Borland/CodeGear C++

This subsection describes the necessary steps to use Distutils with the Borland C++ compiler version 5.5. First you have to know that Borland’s object file format (OMF) is different from the format used by the Python version you can download from the Python or ActiveState Web site. (Python is built with Microsoft Visual C++, which uses COFF as the object file format.) For this reason you have to convert Python’s library \texttt{python25.lib} into the Borland format. You can do this as follows:

\begin{verbatim}
coff2omf python25.lib python25_bcpp.lib
\end{verbatim}

The \texttt{coff2omf} program comes with the Borland compiler. The file \texttt{python25.lib} is in the \texttt{Libs} directory of your Python installation. If your extension uses other libraries (zlib, ...) you have to convert them too.

The converted files have to reside in the same directories as the normal libraries.

How does Distutils manage to use these libraries with their changed names? If the extension needs a library (eg. \texttt{foo}) Distutils checks first if it finds a library with suffix \texttt{_bcpp} (eg. \texttt{foo\_bcpp.lib}) and then uses this library. In the case it doesn’t find such a special library it uses the default name (\texttt{foo.lib})\footnote{This also means you could replace all existing COFF-libraries with OMF-libraries of the same name.}

To let Distutils compile your extension with Borland C++ you now have to type:

\begin{verbatim}
python setup.py build --compiler=bcpp
\end{verbatim}

If you want to use the Borland C++ compiler as the default, you could specify this in your personal or system-wide configuration file for Distutils (see section \textit{Distutils Configuration Files}.)

**See Also:**

- C++Builder Compiler Information about the free C++ compiler from Borland, including links to the download pages.

- Creating Python Extensions Using Borland’s Free Compiler Document describing how to use Borland’s free command-line C++ compiler to build Python.

#### 6.2.2 GNU C / Cygwin / MinGW

This section describes the necessary steps to use Distutils with the GNU C/C++ compilers in their Cygwin and MinGW distributions.\footnote{Check \url{http://sources.redhat.com/cygwin} and \url{http://www.mingw.org/} for more information} For a Python interpreter that was built with Cygwin, everything should work without any of these following steps.

Not all extensions can be built with MinGW or Cygwin, but many can. Extensions most likely to not work are those that use C++ or depend on Microsoft Visual C extensions.

To let Distutils compile your extension with Cygwin you have to type:

\begin{verbatim}
python setup.py build --compiler=cygwin
\end{verbatim}

and for Cygwin in no-cygwin mode\footnote{Then you have no POSIX emulation available, but you also don’t need \texttt{cygwin1.dll}.} or for MinGW type:

\begin{verbatim}
python setup.py build --compiler=mingw32
\end{verbatim}
If you want to use any of these options/compilers as default, you should consider writing it in your personal or system-wide configuration file for Distutils (see section Distutils Configuration Files.)

**Older Versions of Python and MinGW**

The following instructions only apply if you’re using a version of Python inferior to 2.4.1 with a MinGW inferior to 3.0.0 (with binutils-2.13.90-20030111-1).

These compilers require some special libraries. This task is more complex than for Borland’s C++, because there is no program to convert the library. First you have to create a list of symbols which the Python DLL exports. (You can find a good program for this task at http://sourceforge.net/projects/mingw/files/MinGW/Extension/pexports/).

```bash
pexports python25.dll >python25.def
```

The location of an installed `python25.dll` will depend on the installation options and the version and language of Windows. In a “just for me” installation, it will appear in the root of the installation directory. In a shared installation, it will be located in the system directory.

Then you can create from these information an import library for gcc.

```bash
/cygwin/bin/dlltool --dllname python25.dll --def python25.def --output-lib libpython25.a
```

The resulting library has to be placed in the same directory as `python25.lib`. (Should be the `libs` directory under your Python installation directory.)

If your extension uses other libraries (zlib,...) you might have to convert them too. The converted files have to reside in the same directories as the normal libraries do.

**See Also:**

Building Python modules on MS Windows platform with MinGW  Information about building the required libraries for the MinGW environment.
>>> 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-reference.

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()`:
  ```python
  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:
  ```python
  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__()` and `__exit__()` 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__()` and `__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:

```python
>>> 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:

```
>>> 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 (including 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). Hashable 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 dictionaries) 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 typeiter.

**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 include 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 between “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:

  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:

  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:

  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
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:

```ambiguity
>>> 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`:

```ambiguity
>>> 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 preDeclaring 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 doubles 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 dict-views.

**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.

<table>
<thead>
<tr>
<th>Release</th>
<th>Derived from</th>
<th>Year</th>
<th>Owner</th>
<th>GPL compatible?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9.0 thru 1.2</td>
<td>n/a</td>
<td>1991-1995</td>
<td>CWI</td>
<td>yes</td>
</tr>
<tr>
<td>1.3 thru 1.5.2</td>
<td>1.2</td>
<td>1995-1999</td>
<td>CNRI</td>
<td>yes</td>
</tr>
<tr>
<td>1.6</td>
<td>1.5.2</td>
<td>2000</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.0</td>
<td>1.6</td>
<td>2000</td>
<td>BeOpen.com</td>
<td>no</td>
</tr>
<tr>
<td>1.6.1</td>
<td>1.6</td>
<td>2001</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>no</td>
</tr>
<tr>
<td>2.0.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.1</td>
<td>2.1+2.0.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.2</td>
<td>2.1.1</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.3</td>
<td>2.1.2</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2 and above</td>
<td>2.1.1</td>
<td>2001-now</td>
<td>PSF</td>
<td>yes</td>
</tr>
</tbody>
</table>

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:

```
/ \  
|  Copyright (c) 1996. 
|  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.3.4 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:

The XML-RPC client interface is

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 MERCHANT-
ABILITY 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.                                   *
 *                                                                               *
*********************************************************************************/
C.3.12 OpenSSL

The modules `hashlib`, `posix`, `ssl`, `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
EXPRESSED 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 rouines 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.
INDEX

Symbols
..., 23
__future__, 25
__slots__, 30
>>, 23
2to3, 23

A
abstract base class, 23
argument, 23
attribute, 23

B
BDFL, 23
bytecode, 24
bytes-like object, 23

c
CFLAGS, 20
class, 24
coevolution, 24
complex number, 24
context manager, 24
CPython, 24

D
decorator, 24
descriptor, 24
dictionary, 24
docstring, 24
duck-typing, 25

E
EAFP, 25
environment variable
  CFLAGS, 20
  HOME, 17
  HOMEDRIVE, 18
  HOMEPATH, 18
  PYTHONHOME, 15
  PYTHONPATH, 15
  USERPROFILE, 17

expression, 25
extension module, 25

F
file object, 25
file-like object, 25
finder, 25
floor division, 25
function, 25
function annotation, 25

G
garbage collection, 25
generator, 26
generator expression, 26
GIL, 26
global interpreter lock, 26

H
hashable, 26
HOME, 17
HOMEDRIVE, 18
HOMEPATH, 18

I
IDLE, 26
immutable, 26
import path, 26
importer, 26
importing, 26
interactive, 26
interpreted, 26
iterable, 27
iterator, 27

K
key function, 27
keyword argument, 27

L
lambda, 27
LBYL, 27
list, 27
list comprehension, 27
loader, 27

M
mapping, 27
meta path finder, 28
metaclass, 28
method, 28
method resolution order, 28
module, 28
MRO, 28
mutable, 28

N
named tuple, 28
namespace, 28
namespace package, 28
nested scope, 28
new-style class, 28

O
object, 28

P
package, 28
parameter, 29
path based finder, 29
path entry, 29
path entry finder, 29
path entry hook, 29
portion, 29
positional argument, 29
provisional argument, 29
provisional package, 29
Python 3000, 29
Python Enhancement Proposals
    PEP 238, 25
    PEP 278, 31
    PEP 302, 25, 27
    PEP 3107, 25
    PEP 3116, 31
    PEP 3155, 30
    PEP 343, 24
    PEP 362, 23, 29
    PEP 411, 29
    PEP 420, 25, 28, 29
PYTHONHOME, 15
Pythonic, 29
PYTHONPATH, 15

Q
qualified name, 30

R
reference count, 30
regular package, 30

S
sequence, 30
slice, 30
special method, 30
statement, 30
struct sequence, 30

T
triple-quoted string, 31
type, 31

U
universal newlines, 31
USERPROFILE, 17

V
view, 31
virtual machine, 31

Z
Zen of Python, 31