

Python Standard Library

3.10.0a5 reference

February 13, 2021

Justin Mitchell

While [The Python Language Reference](#) describes the exact syntax and semantics of the Python language, this library reference manual describes the standard library that is distributed with Python. It also describes some of the optional components that are commonly included in Python distributions.

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

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

In addition to the standard library, there is a growing collection of several thousand components (from individual programs and modules to packages and entire application development frameworks), available from the [Python Package Index](#).

- [Introduction](#)
 - [Notes on availability](#)
- [Built-in Functions](#)
- [Built-in Constants](#)
 - [Constants added by the `site` module](#)
- [Built-in Types](#)
 - [Truth Value Testing](#)
 - [Boolean Operations](#) — `and`, `or`, `not`
 - [Comparisons](#)
 - [Numeric Types](#) — `int`, `float`, `complex`
 - [Iterator Types](#)
 - [Sequence Types](#) — `list`, `tuple`, `range`
 - [Text Sequence Type](#) — `str`
 - [Binary Sequence Types](#) — `bytes`, `bytearray`, `memoryview`
 - [Set Types](#) — `set`, `frozenset`
 - [Mapping Types](#) — `dict`
 - [Context Manager Types](#)
 - [Type Annotation Types](#) — `Generic Alias`, `Union`
 - [Other Built-in Types](#)
 - [Special Attributes](#)
- [Built-in Exceptions](#)
 - [Base classes](#)
 - [Concrete exceptions](#)
 - [Warnings](#)
 - [Exception hierarchy](#)
- [Text Processing Services](#)
 - [string](#) — Common string operations
 - [re](#) — Regular expression operations
 - [difflib](#) — Helpers for computing deltas
 - [textwrap](#) — Text wrapping and filling
 - [unicodedata](#) — Unicode Database

- `stringprep` — Internet String Preparation
 - `readline` — GNU readline interface
 - `rlcompleter` — Completion function for GNU readline
- **Binary Data Services**
 - `struct` — Interpret bytes as packed binary data
 - `codecs` — Codec registry and base classes
- **Data Types**
 - `datetime` — Basic date and time types
 - `zoneinfo` — IANA time zone support
 - `calendar` — General calendar-related functions
 - `collections` — Container datatypes
 - `collections.abc` — Abstract Base Classes for Containers
 - `heapq` — Heap queue algorithm
 - `bisect` — Array bisection algorithm
 - `array` — Efficient arrays of numeric values
 - `weakref` — Weak references
 - `types` — Dynamic type creation and names for built-in types
 - `copy` — Shallow and deep copy operations
 - `pprint` — Data pretty printer
 - `reprlib` — Alternate `repr()` implementation
 - `enum` — Support for enumerations
 - `graphlib` — Functionality to operate with graph-like structures
- **Numeric and Mathematical Modules**
 - `numbers` — Numeric abstract base classes
 - `math` — Mathematical functions
 - `cmath` — Mathematical functions for complex numbers
 - `decimal` — Decimal fixed point and floating point arithmetic
 - `fractions` — Rational numbers
 - `random` — Generate pseudo-random numbers
 - `statistics` — Mathematical statistics functions
- **Functional Programming Modules**
 - `itertools` — Functions creating iterators for efficient looping
 - `functools` — Higher-order functions and operations on callable objects
 - `operator` — Standard operators as functions

- **File and Directory Access**
 - `pathlib` — Object-oriented filesystem paths
 - `os.path` — Common pathname manipulations
 - `fileinput` — Iterate over lines from multiple input streams
 - `stat` — Interpreting `stat()` results
 - `filecmp` — File and Directory Comparisons
 - `tempfile` — Generate temporary files and directories
 - `glob` — Unix style pathname pattern expansion
 - `fnmatch` — Unix filename pattern matching
 - `linecache` — Random access to text lines
 - `shutil` — High-level file operations
- **Data Persistence**
 - `pickle` — Python object serialization
 - `copyreg` — Register `pickle` support functions
 - `shelve` — Python object persistence
 - `marshal` — Internal Python object serialization
 - `dbm` — Interfaces to Unix “databases”
 - `sqlite3` — DB-API 2.0 interface for SQLite databases
- **Data Compression and Archiving**
 - `zlib` — Compression compatible with **gzip**
 - `gzip` — Support for **gzip** files
 - `bz2` — Support for **bzip2** compression
 - `lzma` — Compression using the LZMA algorithm
 - `zipfile` — Work with ZIP archives
 - `tarfile` — Read and write tar archive files
- **File Formats**
 - `csv` — CSV File Reading and Writing
 - `configparser` — Configuration file parser
 - `netrc` — `netrc` file processing
 - `xdrlib` — Encode and decode XDR data
 - `plistlib` — Generate and parse Apple `.plist` files
- **Cryptographic Services**
 - `hashlib` — Secure hashes and message digests
 - `hmac` — Keyed-Hashing for Message Authentication

- `secrets` — Generate secure random numbers for managing secrets
- **Generic Operating System Services**
 - `os` — Miscellaneous operating system interfaces
 - `io` — Core tools for working with streams
 - `time` — Time access and conversions
 - `argparse` — Parser for command-line options, arguments and sub-commands
 - `getopt` — C-style parser for command line options
 - `logging` — Logging facility for Python
 - `logging.config` — Logging configuration
 - `logging.handlers` — Logging handlers
 - `getpass` — Portable password input
 - `curses` — Terminal handling for character-cell displays
 - `curses.textpad` — Text input widget for curses programs
 - `curses.ascii` — Utilities for ASCII characters
 - `curses.panel` — A panel stack extension for curses
 - `platform` — Access to underlying platform's identifying data
 - `errno` — Standard errno system symbols
 - `ctypes` — A foreign function library for Python
- **Concurrent Execution**
 - `threading` — Thread-based parallelism
 - `multiprocessing` — Process-based parallelism
 - `multiprocessing.shared_memory` — Provides shared memory for direct access across processes
 - **The concurrent package**
 - `concurrent.futures` — Launching parallel tasks
 - `subprocess` — Subprocess management
 - `sched` — Event scheduler
 - `queue` — A synchronized queue class
 - `_thread` — Low-level threading API
- **`contextvars` — Context Variables**
 - **Context Variables**
 - **Manual Context Management**
 - **asyncio support**

- **Networking and Interprocess Communication**
 - `asyncio` — Asynchronous I/O
 - `socket` — Low-level networking interface
 - `ssl` — TLS/SSL wrapper for socket objects
 - `select` — Waiting for I/O completion
 - `selectors` — High-level I/O multiplexing
 - `asyncore` — Asynchronous socket handler
 - `asynchat` — Asynchronous socket command/response handler
 - `signal` — Set handlers for asynchronous events
 - `mmap` — Memory-mapped file support
- **Internet Data Handling**
 - `email` — An email and MIME handling package
 - `json` — JSON encoder and decoder
 - `mailcap` — Mailcap file handling
 - `mailbox` — Manipulate mailboxes in various formats
 - `mimetypes` — Map filenames to MIME types
 - `base64` — Base16, Base32, Base64, Base85 Data Encodings
 - `binhex` — Encode and decode binhex4 files
 - `binascii` — Convert between binary and ASCII
 - `quopri` — Encode and decode MIME quoted-printable data
 - `uu` — Encode and decode uuencode files
- **Structured Markup Processing Tools**
 - `html` — HyperText Markup Language support
 - `html.parser` — Simple HTML and XHTML parser
 - `html.entities` — Definitions of HTML general entities
 - **XML Processing Modules**
 - `xml.etree.ElementTree` — The ElementTree XML API
 - `xml.dom` — The Document Object Model API
 - `xml.dom.minidom` — Minimal DOM implementation
 - `xml.dom.pulldom` — Support for building partial DOM trees
 - `xml.sax` — Support for SAX2 parsers
 - `xml.sax.handler` — Base classes for SAX handlers
 - `xml.sax.saxutils` — SAX Utilities
 - `xml.sax.xmlreader` — Interface for XML parsers

- `xml.parsers.expat` — Fast XML parsing using Expat
- **Internet Protocols and Support**
 - `webbrowser` — Convenient Web-browser controller
 - `cgi` — Common Gateway Interface support
 - `cgitb` — Traceback manager for CGI scripts
 - `wsgiref` — WSGI Utilities and Reference Implementation
 - `urllib` — URL handling modules
 - `urllib.request` — Extensible library for opening URLs
 - `urllib.response` — Response classes used by urllib
 - `urllib.parse` — Parse URLs into components
 - `urllib.error` — Exception classes raised by urllib.request
 - `urllib.robotparser` — Parser for robots.txt
 - `http` — HTTP modules
 - `http.client` — HTTP protocol client
 - `ftplib` — FTP protocol client
 - `poplib` — POP3 protocol client
 - `imaplib` — IMAP4 protocol client
 - `nntplib` — NNTP protocol client
 - `smtplib` — SMTP protocol client
 - `smtpd` — SMTP Server
 - `telnetlib` — Telnet client
 - `uuid` — UUID objects according to **RFC 4122**
 - `socketserver` — A framework for network servers
 - `http.server` — HTTP servers
 - `http.cookies` — HTTP state management
 - `http.cookiejar` — Cookie handling for HTTP clients
 - `xmlrpc` — XMLRPC server and client modules
 - `xmlrpc.client` — XML-RPC client access
 - `xmlrpc.server` — Basic XML-RPC servers
 - `ipaddress` — IPv4/IPv6 manipulation library
- **Multimedia Services**
 - `audioop` — Manipulate raw audio data
 - `aifc` — Read and write AIFF and AIFC files
 - `sunau` — Read and write Sun AU files

- `wave` — Read and write WAV files
 - `chunk` — Read IFF chunked data
 - `colorsys` — Conversions between color systems
 - `imghdr` — Determine the type of an image
 - `sndhdr` — Determine type of sound file
 - `ossaudiodev` — Access to OSS-compatible audio devices
- **Internationalization**
 - `gettext` — Multilingual internationalization services
 - `locale` — Internationalization services
- **Program Frameworks**
 - `turtle` — Turtle graphics
 - `cmd` — Support for line-oriented command interpreters
 - `shlex` — Simple lexical analysis
- **Graphical User Interfaces with Tk**
 - `tkinter` — Python interface to Tcl/Tk
 - `tkinter.colorchooser` — Color choosing dialog
 - `tkinter.font` — Tkinter font wrapper
 - **Tkinter Dialogs**
 - `tkinter.messagebox` — Tkinter message prompts
 - `tkinter.scrolledtext` — Scrolled Text Widget
 - `tkinter.dnd` — Drag and drop support
 - `tkinter.ttk` — Tk themed widgets
 - `tkinter.tix` — Extension widgets for Tk
 - **IDLE**
 - **Other Graphical User Interface Packages**
- **Development Tools**
 - `typing` — Support for type hints
 - `pydoc` — Documentation generator and online help system
 - **Python Development Mode**
 - **Effects of the Python Development Mode**
 - **ResourceWarning Example**
 - **Bad file descriptor error example**
 - `doctest` — Test interactive Python examples
 - `unittest` — Unit testing framework

- `unittest.mock` — mock object library
- `unittest.mock` — getting started
- [2to3](#) - Automated Python 2 to 3 code translation
- `test` — Regression tests package for Python
- `test.support` — Utilities for the Python test suite
- `test.support.socket_helper` — Utilities for socket tests
- `test.support.script_helper` — Utilities for the Python execution tests
- `test.support.bytecode_helper` — Support tools for testing correct bytecode generation
- `test.support.threading_helper` — Utilities for threading tests
- `test.support.os_helper` — Utilities for os tests
- `test.support.import_helper` — Utilities for import tests
- `test.support.warnings_helper` — Utilities for warnings tests
- **Debugging and Profiling**
 - [Audit events table](#)
 - `bdb` — Debugger framework
 - `faulthandler` — Dump the Python traceback
 - `pdb` — The Python Debugger
 - [The Python Profilers](#)
 - `timeit` — Measure execution time of small code snippets
 - `trace` — Trace or track Python statement execution
 - `tracemalloc` — Trace memory allocations
- **Software Packaging and Distribution**
 - `distutils` — Building and installing Python modules
 - `ensurepip` — Bootstrapping the `pip` installer
 - `venv` — Creation of virtual environments
 - `zipapp` — Manage executable Python zip archives
- **Python Runtime Services**
 - `sys` — System-specific parameters and functions
 - `sysconfig` — Provide access to Python's configuration information
 - `builtins` — Built-in objects
 - `__main__` — Top-level script environment
 - `warnings` — Warning control
 - `dataclasses` — Data Classes

- `contextlib` — Utilities for `with`-statement contexts
- `abc` — Abstract Base Classes
- `atexit` — Exit handlers
- `traceback` — Print or retrieve a stack traceback
- `__future__` — Future statement definitions
- `gc` — Garbage Collector interface
- `inspect` — Inspect live objects
- `site` — Site-specific configuration hook
- Custom Python Interpreters
 - `code` — Interpreter base classes
 - `codeop` — Compile Python code
- Importing Modules
 - `zipimport` — Import modules from Zip archives
 - `pkgutil` — Package extension utility
 - `modulefinder` — Find modules used by a script
 - `runpy` — Locating and executing Python modules
 - `importlib` — The implementation of `import`
 - Using `importlib.metadata`
- Python Language Services
 - `ast` — Abstract Syntax Trees
 - `symtable` — Access to the compiler's symbol tables
 - `token` — Constants used with Python parse trees
 - `keyword` — Testing for Python keywords
 - `tokenize` — Tokenizer for Python source
 - `tabnanny` — Detection of ambiguous indentation
 - `pyclbr` — Python module browser support
 - `py_compile` — Compile Python source files
 - `compileall` — Byte-compile Python libraries
 - `dis` — Disassembler for Python bytecode
 - `pickletools` — Tools for pickle developers
- MS Windows Specific Services
 - `msilib` — Read and write Microsoft Installer files
 - `msvcrt` — Useful routines from the MS VC++ runtime
 - `winreg` — Windows registry access

- `winsound` — Sound-playing interface for Windows
- **Unix Specific Services**
 - `posix` — The most common POSIX system calls
 - `pwd` — The password database
 - `spwd` — The shadow password database
 - `grp` — The group database
 - `crypt` — Function to check Unix passwords
 - `termios` — POSIX style tty control
 - `tty` — Terminal control functions
 - `pty` — Pseudo-terminal utilities
 - `fcntl` — The `fcntl` and `ioctl` system calls
 - `pipes` — Interface to shell pipelines
 - `resource` — Resource usage information
 - [`nis` — Interface to Sun's NIS \(Yellow Pages\)](#)
 - `syslog` — Unix syslog library routines
- **Superseded Modules**
 - `optparse` — Parser for command line options
 - `imp` — Access the import internals
- **Undocumented Modules**
 - Platform specific modules

The Python Language Reference

This reference manual describes the syntax and “core semantics” of the language. It is terse, but attempts to be exact and complete. The semantics of non-essential built-in object types and of the built-in functions and modules are described in [The Python Standard Library](#). For an informal introduction to the language, see [The Python Tutorial](#). For C or C++ programmers, two additional manuals exist: [Extending and Embedding the Python Interpreter](#) describes the high-level picture of how to write a Python extension module, and the [Python/C API Reference Manual](#) describes the interfaces available to C/C++ programmers in detail.

- 1. Introduction
 - 1.1. Alternate Implementations
 - 1.2. Notation
- 2. Lexical analysis
 - 2.1. Line structure
 - 2.2. Other tokens
 - 2.3. Identifiers and keywords
 - 2.4. Literals
 - 2.5. Operators
 - 2.6. Delimiters
- 3. Data model
 - 3.1. Objects, values and types
 - 3.2. The standard type hierarchy
 - 3.3. Special method names
 - 3.4. Coroutines
- 4. Execution model
 - 4.1. Structure of a program

- 4.2. Naming and binding
 - 4.3. Exceptions
- 5. The import system
 - 5.1. `importlib`
 - 5.2. Packages
 - 5.3. Searching
 - 5.4. Loading
 - 5.5. The Path Based Finder
 - 5.6. Replacing the standard import system
 - 5.7. Package Relative Imports
 - 5.8. Special considerations for `__main__`
 - 5.9. Open issues
 - 5.10. References
- 6. Expressions
 - 6.1. Arithmetic conversions
 - 6.2. Atoms
 - 6.3. Primaries
 - 6.4. Await expression
 - 6.5. The power operator
 - 6.6. Unary arithmetic and bitwise operations
 - 6.7. Binary arithmetic operations
 - 6.8. Shifting operations
 - 6.9. Binary bitwise operations
 - 6.10. Comparisons
 - 6.11. Boolean operations
 - 6.12. Assignment expressions
 - 6.13. Conditional expressions
 - 6.14. Lambdas
 - 6.15. Expression lists
 - 6.16. Evaluation order
 - 6.17. Operator precedence
- 7. Simple statements
 - 7.1. Expression statements
 - 7.2. Assignment statements
 - 7.3. The `assert` statement
 - 7.4. The `pass` statement

- 7.5. The `del` statement
 - 7.6. The `return` statement
 - 7.7. The `yield` statement
 - 7.8. The `raise` statement
 - 7.9. The `break` statement
 - 7.10. The `continue` statement
 - 7.11. The `import` statement
 - 7.12. The `global` statement
 - 7.13. The `nonlocal` statement
- 8. Compound statements
 - 8.1. The `if` statement
 - 8.2. The `while` statement
 - 8.3. The `for` statement
 - 8.4. The `try` statement
 - 8.5. The `with` statement
 - 8.6. Function definitions
 - 8.7. Class definitions
 - 8.8. Coroutines
- 9. Top-level components
 - 9.1. Complete Python programs
 - 9.2. File input
 - 9.3. Interactive input
 - 9.4. Expression input
- 10. Full Grammar specification

Python Setup and Usage

This part of the documentation is devoted to general information on the setup of the Python environment on different platforms, the invocation of the interpreter and things that make working with Python easier.

- 1. [Command line and environment](#)
 - 1.1. [Command line](#)
 - 1.1.1. [Interface options](#)
 - 1.1.2. [Generic options](#)
 - 1.1.3. [Miscellaneous options](#)
 - 1.1.4. [Options you shouldn't use](#)
 - 1.2. [Environment variables](#)
 - 1.2.1. [Debug-mode variables](#)
- 2. [Using Python on Unix platforms](#)
 - 2.1. [Getting and installing the latest version of Python](#)
 - 2.1.1. [On Linux](#)
 - 2.1.2. [On FreeBSD and OpenBSD](#)
 - 2.1.3. [On OpenSolaris](#)
 - 2.2. [Building Python](#)
 - 2.3. [Python-related paths and files](#)
 - 2.4. [Miscellaneous](#)
- 3. [Using Python on Windows](#)
 - 3.1. [The full installer](#)
 - 3.1.1. [Installation steps](#)
 - 3.1.2. [Removing the MAX_PATH Limitation](#)
 - 3.1.3. [Installing Without UI](#)
 - 3.1.4. [Installing Without Downloading](#)
 - 3.1.5. [Modifying an install](#)

- 3.2. The Microsoft Store package
 - 3.2.1. Known Issues
- 3.3. The nuget.org packages
- 3.4. The embeddable package
 - 3.4.1. Python Application
 - 3.4.2. Embedding Python
- 3.5. Alternative bundles
- 3.6. Configuring Python
 - 3.6.1. Excursus: Setting environment variables
 - 3.6.2. Finding the Python executable
- 3.7. UTF-8 mode
- 3.8. Python Launcher for Windows
 - 3.8.1. Getting started
 - 3.8.1.1. From the command-line
 - 3.8.1.2. Virtual environments
 - 3.8.1.3. From a script
 - 3.8.1.4. From file associations
 - 3.8.2. Shebang Lines
 - 3.8.3. Arguments in shebang lines
 - 3.8.4. Customization
 - 3.8.4.1. Customization via INI files
 - 3.8.4.2. Customizing default Python versions
 - 3.8.5. Diagnostics
- 3.9. Finding modules
- 3.10. Additional modules
 - 3.10.1. PyWin32
 - 3.10.2. cx_Freeze
 - 3.10.3. WConio
- 3.11. Compiling Python on Windows
- 3.12. Other Platforms
- 4. Using Python on a Macintosh
 - 4.1. Getting and Installing MacPython
 - 4.1.1. How to run a Python script
 - 4.1.2. Running scripts with a GUI
 - 4.1.3. Configuration
 - 4.2. The IDE

- [4.3. Installing Additional Python Packages](#)
 - [4.4. GUI Programming on the Mac](#)
 - [4.5. Distributing Python Applications on the Mac](#)
 - [4.6. Other Resources](#)
- [5. Editors and IDEs](#)

Python HOWTOs

Python HOWTOs are documents that cover a single, specific topic, and attempt to cover it fairly completely. Modelled on the Linux Documentation Project's HOWTO collection, this collection is an effort to foster documentation that's more detailed than the Python Library Reference.

Currently, the HOWTOs are:

- [Porting Python 2 Code to Python 3](#)
- [Porting Extension Modules to Python 3](#)
- [Curses Programming with Python](#)
- [Descriptor HowTo Guide](#)
- [Functional Programming HOWTO](#)
- [Logging HOWTO](#)
- [Logging Cookbook](#)
- [Regular Expression HOWTO](#)
- [Socket Programming HOWTO](#)
- [Sorting HOW TO](#)
- [Unicode HOWTO](#)
- [HOWTO Fetch Internet Resources Using The urllib Package](#)
- [Argparse Tutorial](#)
- [An introduction to the ipaddress module](#)
- [Argument Clinic How-To](#)
- [Instrumenting CPython with DTrace and SystemTap](#)

Installing Python Modules

As a popular open source development project, Python has an active supporting community of contributors and users that also make their software available for other Python developers to use under open source license terms.

This allows Python users to share and collaborate effectively, benefiting from the solutions others have already created to common (and sometimes even rare!) problems, as well as potentially contributing their own solutions to the common pool.

This guide covers the installation part of the process. For a guide to creating and sharing your own Python projects, refer to the [distribution guide](#).

Note For corporate and other institutional users, be aware that many organisations have their own policies around using and contributing to open source software. Please take such policies into account when making use of the distribution and installation tools provided with Python.

Key terms

- `pip` is the preferred installer program. Starting with Python 3.4, it is included by default with the Python binary installers.
- A *virtual environment* is a semi-isolated Python environment that allows packages to be installed for use by a particular application, rather than being installed system wide.
- `venv` is the standard tool for creating virtual environments, and has been part of Python since Python 3.3. Starting with Python 3.4, it defaults to installing `pip` into all created virtual environments.

- `virtualenv` is a third party alternative (and predecessor) to `venv`. It allows virtual environments to be used on versions of Python prior to 3.4, which either don't provide `venv` at all, or aren't able to automatically install `pip` into created environments.
- The [Python Packaging Index](#) is a public repository of open source licensed packages made available for use by other Python users.
- the [Python Packaging Authority](#) is the group of developers and documentation authors responsible for the maintenance and evolution of the standard packaging tools and the associated metadata and file format standards. They maintain a variety of tools, documentation, and issue trackers on both [GitHub](#) and [Bitbucket](#).
- `distutils` is the original build and distribution system first added to the Python standard library in 1998. While direct use of `distutils` is being phased out, it still laid the foundation for the current packaging and distribution infrastructure, and it not only remains part of the standard library, but its name lives on in other ways (such as the name of the mailing list used to coordinate Python packaging standards development).

Changed in version 3.5: The use of `venv` is now recommended for creating virtual environments.

See also [Python Packaging User Guide: Creating and using virtual environments](#)

Basic usage

The standard packaging tools are all designed to be used from the command line.

The following command will install the latest version of a module and its dependencies from the Python Packaging Index:

```
python -m pip install SomePackage
```

Note For POSIX users (including Mac OS X and Linux users), the examples in this guide assume the use of a [virtual environment](#).

For Windows users, the examples in this guide assume that the option to adjust the system PATH environment variable was selected when installing Python.

It's also possible to specify an exact or minimum version directly on the command line. When using comparator operators such as `>`, `<` or some other special character which get interpreted by shell, the package name and the version should be enclosed within double quotes:

```
python -m pip install SomePackage==1.0.4    # specific version
```

```
python -m pip install "SomePackage>=1.0.4"  # minimum version
```

Normally, if a suitable module is already installed, attempting to install it again will have no effect. Upgrading existing modules must be requested explicitly:

```
python -m pip install --upgrade SomePackage
```

More information and resources regarding `pip` and its capabilities can be found in the [Python Packaging User Guide](#).

Creation of virtual environments is done through the `venv` module. Installing packages into an active virtual environment uses the commands shown above.

See also [Python Packaging User Guide: Installing Python Distribution Packages](#)

How do I ...?

These are quick answers or links for some common tasks.

... install `pip` in versions of Python prior to Python 3.4?

Python only started bundling `pip` with Python 3.4. For earlier versions, `pip` needs to be “bootstrapped” as described in the Python Packaging User Guide.

See also [Python Packaging User Guide: Requirements for Installing Packages](#)

... install packages just for the current user?

Passing the `--user` option to `python -m pip install` will install a package just for the current user, rather than for all users of the system.

... install scientific Python packages?

A number of scientific Python packages have complex binary dependencies, and aren't currently easy to install using `pip` directly. At this point in time, it will often be easier for users to install these packages by [other means](#) rather than attempting to install them with `pip`.

See also [Python Packaging User Guide: Installing Scientific Packages](#)

... work with multiple versions of Python installed in parallel?

On Linux, Mac OS X, and other POSIX systems, use the versioned Python commands in combination with the `-m` switch to run the appropriate copy of `pip`:

```
python2    -m pip install SomePackage # default Python 2
python2.7  -m pip install SomePackage # specifically Python 2.7
python3    -m pip install SomePackage # default Python 3
python3.4  -m pip install SomePackage # specifically Python 3.4
```

Appropriately versioned `pip` commands may also be available.

On Windows, use the `py` Python launcher in combination with the `-m` switch:

```
py -2      -m pip install SomePackage # default Python 2
py -2.7    -m pip install SomePackage # specifically Python 2.7
```

```
py -3 -m pip install SomePackage # default Python 3
```

```
py -3.4 -m pip install SomePackage # specifically Python 3.4
```

Common installation issues

Installing into the system Python on Linux

On Linux systems, a Python installation will typically be included as part of the distribution. Installing into this Python installation requires root access to the system, and may interfere with the operation of the system package manager and other components of the system if a component is unexpectedly upgraded using `pip`.

On such systems, it is often better to use a virtual environment or a per-user installation when installing packages with `pip`.

Pip not installed

It is possible that `pip` does not get installed by default. One potential fix is:

```
python -m ensurepip --default-pip
```

There are also additional resources for [installing pip](#).

Installing binary extensions

Python has typically relied heavily on source based distribution, with end users being expected to compile extension modules from source as part of the installation process.

With the introduction of support for the binary `wheel` format, and the ability to publish wheels for at least Windows and Mac OS X through the Python Packaging Index, this

problem is expected to diminish over time, as users are more regularly able to install pre-built extensions rather than needing to build them themselves.

Some of the solutions for installing [scientific software](#) that are not yet available as pre-built `wheel` files may also help with obtaining other binary extensions without needing to build them locally.

See also [Python Packaging User Guide: Binary Extensions](#)

Distributing Python Modules

Email

distutils-sig@python.org

As a popular open source development project, Python has an active supporting community of contributors and users that also make their software available for other Python developers to use under open source license terms.

This allows Python users to share and collaborate effectively, benefiting from the solutions others have already created to common (and sometimes even rare!) problems, as well as potentially contributing their own solutions to the common pool.

This guide covers the distribution part of the process. For a guide to installing other Python projects, refer to the [installation guide](#).

Note For corporate and other institutional users, be aware that many organisations have their own policies around using and contributing to open source software. Please take such policies into account when making use of the distribution and installation tools provided with Python.

Key terms

- the [Python Packaging Index](#) is a public repository of open source licensed packages made available for use by other Python users
- the [Python Packaging Authority](#) are the group of developers and documentation authors responsible for the maintenance and evolution of the standard packaging tools and the associated metadata and file format standards. They maintain a variety of tools, documentation and issue trackers on both [GitHub](#) and [Bitbucket](#).
- `distutils` is the original build and distribution system first added to the Python standard library in 1998. While direct use of `distutils` is being phased out, it still laid the foundation for the current packaging and distribution infrastructure, and it not only remains part of the standard library, but its name lives on in other ways (such as the name of the mailing list used to coordinate Python packaging standards development).
- `setuptools` is a (largely) drop-in replacement for `distutils` first published in 2004. Its most notable addition over the unmodified `distutils` tools was the ability to declare dependencies on other packages. It is currently recommended as a more regularly updated alternative to `distutils` that offers consistent support for more recent packaging standards across a wide range of Python versions.
- `wheel` (in this context) is a project that adds the `bdist_wheel` command to `distutils/setuptools`. This produces a cross platform binary packaging format (called “wheels” or “wheel files” and defined in [PEP 427](#)) that allows Python libraries, even those including binary extensions, to be installed on a system without needing to be built locally.

Open source licensing and collaboration

In most parts of the world, software is automatically covered by copyright. This means that other developers require explicit permission to copy, use, modify and redistribute the software.

Open source licensing is a way of explicitly granting such permission in a relatively consistent way, allowing developers to share and collaborate efficiently by making common solutions to various problems freely available. This leaves many developers free to spend more time focusing on the problems that are relatively unique to their specific situation.

The distribution tools provided with Python are designed to make it reasonably straightforward for developers to make their own contributions back to that common pool of software if they choose to do so.

The same distribution tools can also be used to distribute software within an organisation, regardless of whether that software is published as open source software or not.

Installing the tools

The standard library does not include build tools that support modern Python packaging standards, as the core development team has found that it is important to have standard tools that work consistently, even on older versions of Python.

The currently recommended build and distribution tools can be installed by invoking the `pip` module at the command line:

```
python -m pip install setuptools wheel twine
```

Note For POSIX users (including Mac OS X and Linux users), these instructions assume the use of a [virtual environment](#).

For Windows users, these instructions assume that the option to adjust the system PATH environment variable was selected when installing Python.

The Python Packaging User Guide includes more details on the [currently recommended tools](#).

Reading the Python Packaging User Guide

The Python Packaging User Guide covers the various key steps and elements involved in creating and publishing a project:

- [Project structure](#)
- [Building and packaging the project](#)
- [Uploading the project to the Python Packaging Index](#)
- [The .pypirc file](#)

How do I...?

These are quick answers or links for some common tasks.

... choose a name for my project?

This isn't an easy topic, but here are a few tips:

- check the Python Packaging Index to see if the name is already in use
- check popular hosting sites like GitHub, Bitbucket, etc to see if there is already a project with that name
- check what comes up in a web search for the name you're considering
- avoid particularly common words, especially ones with multiple meanings, as they can make it difficult for users to find your software when searching for it

... create and distribute binary extensions?

This is actually quite a complex topic, with a variety of alternatives available depending on exactly what you're aiming to achieve. See the [Python Packaging User Guide](#) for more information and recommendations.

See also [Python Packaging User Guide: Binary Extensions](#)

Extending and Embedding the Python Interpreter

This document describes how to write modules in C or C++ to extend the Python interpreter with new modules. Those modules can not only define new functions but also new object types and their methods. The document also describes how to embed the Python interpreter in another application, for use as an extension language. Finally, it shows how to compile and link extension modules so that they can be loaded dynamically (at run time) into the interpreter, if the underlying operating system supports this feature.

This document assumes basic knowledge about Python. For an informal introduction to the language, see [The Python Tutorial](#). [The Python Language Reference](#) gives a more formal definition of the language. [The Python Standard Library](#) documents the existing object types, functions and modules (both built-in and written in Python) that give the language its wide application range.

For a detailed description of the whole Python/C API, see the separate [Python/C API Reference Manual](#).

Recommended third party tools

This guide only covers the basic tools for creating extensions provided as part of this version of CPython. Third party tools like [Cython](#), [ffi](#), [SWIG](#) and [Numba](#) offer both simpler and more sophisticated approaches to creating C and C++ extensions for Python.

See also

[Python Packaging User Guide: Binary Extensions](#)

The Python Packaging User Guide not only covers several available tools that simplify the creation of binary extensions, but also discusses the various reasons why creating an extension module may be desirable in the first place.

Creating extensions without third party tools

This section of the guide covers creating C and C++ extensions without assistance from third party tools. It is intended primarily for creators of those tools, rather than being a recommended way to create your own C extensions.

- [1. Extending Python with C or C++](#)
 - [1.1. A Simple Example](#)
 - [1.2. Intermezzo: Errors and Exceptions](#)
 - [1.3. Back to the Example](#)
 - [1.4. The Module's Method Table and Initialization Function](#)
 - [1.5. Compilation and Linkage](#)
 - [1.6. Calling Python Functions from C](#)
 - [1.7. Extracting Parameters in Extension Functions](#)
 - [1.8. Keyword Parameters for Extension Functions](#)
 - [1.9. Building Arbitrary Values](#)
 - [1.10. Reference Counts](#)
 - [1.11. Writing Extensions in C++](#)
 - [1.12. Providing a C API for an Extension Module](#)
- [2. Defining Extension Types: Tutorial](#)
 - [2.1. The Basics](#)
 - [2.2. Adding data and methods to the Basic example](#)
 - [2.3. Providing finer control over data attributes](#)

- 2.4. Supporting cyclic garbage collection
 - 2.5. Subclassing other types
- 3. Defining Extension Types: Assorted Topics
 - 3.1. Finalization and De-allocation
 - 3.2. Object Presentation
 - 3.3. Attribute Management
 - 3.4. Object Comparison
 - 3.5. Abstract Protocol Support
 - 3.6. Weak Reference Support
 - 3.7. More Suggestions
- 4. Building C and C++ Extensions
 - 4.1. Building C and C++ Extensions with distutils
 - 4.2. Distributing your extension modules
- 5. Building C and C++ Extensions on Windows
 - 5.1. A Cookbook Approach
 - 5.2. Differences Between Unix and Windows
 - 5.3. Using DLLs in Practice

Embedding the CPython runtime in a larger application

Sometimes, rather than creating an extension that runs inside the Python interpreter as the main application, it is desirable to instead embed the CPython runtime inside a larger application. This section covers some of the details involved in doing that successfully.

- 1. Embedding Python in Another Application
 - 1.1. Very High Level Embedding
 - 1.2. Beyond Very High Level Embedding: An overview
 - 1.3. Pure Embedding
 - 1.4. Extending Embedded Python
 - 1.5. Embedding Python in C++
 - 1.6. Compiling and Linking under Unix-like systems

Python/C API Reference Manual

This manual documents the API used by C and C++ programmers who want to write extension modules or embed Python. It is a companion to [Extending and Embedding the Python Interpreter](#), which describes the general principles of extension writing but does not document the API functions in detail.

- [Introduction](#)
 - [Coding standards](#)
 - [Include Files](#)
 - [Useful macros](#)
 - [Objects, Types and Reference Counts](#)
 - [Exceptions](#)
 - [Embedding Python](#)
 - [Debugging Builds](#)
- [Stable Application Binary Interface](#)
- [The Very High Level Layer](#)
- [Reference Counting](#)
- [Exception Handling](#)
 - [Printing and clearing](#)
 - [Raising exceptions](#)
 - [Issuing warnings](#)
 - [Querying the error indicator](#)
 - [Signal Handling](#)
 - [Exception Classes](#)
 - [Exception Objects](#)
 - [Unicode Exception Objects](#)
 - [Recursion Control](#)
 - [Standard Exceptions](#)
 - [Standard Warning Categories](#)
- [Utilities](#)

- Operating System Utilities
 - System Functions
 - Process Control
 - Importing Modules
 - Data marshalling support
 - Parsing arguments and building values
 - String conversion and formatting
 - Reflection
 - Codec registry and support functions
- Abstract Objects Layer
 - Object Protocol
 - Call Protocol
 - Number Protocol
 - Sequence Protocol
 - Mapping Protocol
 - Iterator Protocol
 - Buffer Protocol
- Concrete Objects Layer
 - Fundamental Objects
 - Numeric Objects
 - Sequence Objects
 - Container Objects
 - Function Objects
 - Other Objects
- Initialization, Finalization, and Threads
 - Before Python Initialization
 - Global configuration variables
 - Initializing and finalizing the interpreter
 - Process-wide parameters
 - Thread State and the Global Interpreter Lock
 - Sub-interpreter support
 - Asynchronous Notifications
 - Profiling and Tracing
 - Advanced Debugger Support
 - Thread Local Storage Support
- Python Initialization Configuration

- Example
- PyWideStringList
- PyStatus
- PyPreConfig
- Preinitialize Python with PyPreConfig
- PyConfig
- Initialization with PyConfig
- Isolated Configuration
- Python Configuration
- Python Path Configuration
- Py_RunMain()
- Py_GetArgcArgv()
- Multi-Phase Initialization Private Provisional API
- Memory Management
 - Overview
 - Allocator Domains
 - Raw Memory Interface
 - Memory Interface
 - Object allocators
 - Default Memory Allocators
 - Customize Memory Allocators
 - The pymalloc allocator
 - tracemalloc C API
 - Examples
- Object Implementation Support
 - Allocating Objects on the Heap
 - Common Object Structures
 - Type Objects
 - Number Object Structures
 - Mapping Object Structures
 - Sequence Object Structures
 - Buffer Object Structures
 - [Async Object Structures](#)
 - Slot Type typedefs
 - Examples
 - Supporting Cyclic Garbage Collection

- [API and ABI Versioning](#)