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.

- 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

- 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

    - `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.

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.

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:

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, cffi, 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.

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.

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.

- API and ABI Versioning