# Python Packaging User Guide

***Release***

**Python Packaging Authority**

**Dec 13, 2017**

# Contents

CHAPTER 1

# Tutorials

**Tutorials** are opinionated step-by-step guides to help you get familiar with packaging concepts. For more detailed information on specific packaging topics, see *Guides*.

## 1.1 Installing Packages

This section covers the basics of how to install Python *packages*.

It's important to note that the term "package" in this context is being used as a synonym for a *distribution* (i.e. a bundle of software to be installed), not to refer to the kind of *package* that you import in your Python source code (i.e. a container of modules). It is common in the Python community to refer to a *distribution* using the term "package". Using the term "distribution" is often not preferred, because it can easily be confused with a Linux distribution, or another larger software distribution like Python itself.

**Contents**

### 1.1.1 Requirements for Installing Packages

This section describes the steps to follow before installing other Python packages.

#### Ensure you can run Python from the command line

Before you go any further, make sure you have Python and that the expected version is available from your command line. You can check this by running:

```
python --version
```

You should get some output like `Python 3.6.3`. If you do not have Python, please install the latest 3.x version from python.org or refer to the Installing Python section of the Hitchhiker's Guide to Python.

---

**Note:** If you're a newcomer and you get an error like this:

```
>>> python --version
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'python' is not defined
```

It's because this command and other suggested commands in this tutorial are intended to be run in a *shell* (also called a *terminal* or *console*). See the Python for Beginners getting started tutorial for an introduction to using your operating system's shell and interacting with Python.

If you're using an enhanced shell like IPython or the Jupyter notebook, you can run system commands like those in this tutorial by prefacing them with a `!` character:

```
.. code-block:: python
```

    In [1]: !python –version Python 3.6.3

---

**Note:** Due to the way most Linux distributions are handling the Python 3 migration, Linux users using the system Python without creating a virtual environment first should replace the `python` command in this tutorial with `python3` and the `pip` command with `pip3 --user`. Do *not* run any of the commands in this tutorial with `sudo`:

if you get a permissions error, come back to the section on creating virtual environments, set one up, and then continue with the tutorial as written.

### Ensure you can run pip from the command line

Additionally, you'll need to make sure you have *pip* available. You can check this by running:

```
pip --version
```

If you installed Python from source, with an installer from python.org, or via Homebrew you should already have pip. If you're on Linux and installed using your OS package manager, you may have to install pip separately, see *Installing pip/setuptools/wheel with Linux Package Managers*.

If `pip` isn't already installed, then first try to bootstrap it from the standard library:

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

If that still doesn't allow you to run `pip`:

- Securely Download get-pip.py[1]
- Run `python get-pip.py`.[2] This will install or upgrade pip. Additionally, it will install *setuptools* and *wheel* if they're not installed already.

> **Warning:** Be cautious if you're using a Python install that's managed by your operating system or another package manager. get-pip.py does not coordinate with those tools, and may leave your system in an inconsistent state. You can use `python get-pip.py --prefix=/usr/local/` to install in `/usr/local` which is designed for locally-installed software.

### Ensure pip, setuptools, and wheel are up to date

While `pip` alone is sufficient to install from pre-built binary archives, up to date copies of the `setuptools` and `wheel` projects are useful to ensure you can also install from source archives:

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

### Optionally, create a virtual environment

See *section below* for details, but here's the basic venv[3] command to use on a typical Linux system:

```
python3 -m venv tutorial_env
source tutorial_env/bin/activate
```

This will create a new virtual environment in the `tutorial_env` subdirectory, and configure the current shell to use it as the default `python` environment.

[1] "Secure" in this context means using a modern browser or a tool like *curl* that verifies SSL certificates when downloading from https URLs.

[2] Depending on your platform, this may require root or Administrator access. *pip* is currently considering changing this by making user installs the default behavior.

[3] Beginning with Python 3.4, `venv` (a stdlib alternative to *virtualenv*) will create virtualenv environments with `pip` pre-installed, thereby making it an equal alternative to *virtualenv*.

### 1.1.2 Creating Virtual Environments

Python "Virtual Environments" allow Python *packages* to be installed in an isolated location for a particular application, rather than being installed globally.

Imagine you have an application that needs version 1 of LibFoo, but another application requires version 2. How can you use both these applications? If you install everything into /usr/lib/python2.7/site-packages (or whatever your platform's standard location is), it's easy to end up in a situation where you unintentionally upgrade an application that shouldn't be upgraded.

Or more generally, what if you want to install an application and leave it be? If an application works, any change in its libraries or the versions of those libraries can break the application.

Also, what if you can't install *packages* into the global site-packages directory? For instance, on a shared host.

In all these cases, virtual environments can help you. They have their own installation directories and they don't share libraries with other virtual environments.

Currently, there are two common tools for creating Python virtual environments:

- venv is available by default in Python 3.3 and later, and installs *pip* and *setuptools* into created virtual environments in Python 3.4 and later.
- *virtualenv* needs to be installed separately, but supports Python 2.6+ and Python 3.3+, and *pip*, *setuptools* and *wheel* are always installed into created virtual environments by default (regardless of Python version).

The basic usage is like so:

Using *virtualenv*:

```
virtualenv <DIR>
source <DIR>/bin/activate
```

Using venv:

```
python3 -m venv <DIR>
source <DIR>/bin/activate
```

For more information, see the virtualenv docs or the venv docs.

Managing multiple virtual environments directly can become tedious, so the *dependency management tutorial* introduces a higher level tool, *Pipenv*, that automatically manages a separate virtual environment for each project and application that you work on.

### 1.1.3 Use pip for Installing

*pip* is the recommended installer. Below, we'll cover the most common usage scenarios. For more detail, see the pip docs, which includes a complete Reference Guide.

### 1.1.4 Installing from PyPI

The most common usage of *pip* is to install from the *Python Package Index* using a *requirement specifier*. Generally speaking, a requirement specifier is composed of a project name followed by an optional *version specifier*. **PEP 440** contains a **full specification** of the currently supported specifiers. Below are some examples.

To install the latest version of "SomeProject":

```
pip install 'SomeProject'
```

To install a specific version:

```
pip install 'SomeProject==1.4'
```

To install greater than or equal to one version and less than another:

```
pip install 'SomeProject>=1,<2'
```

To install a version that's **"compatible"** with a certain version:[4]

```
pip install 'SomeProject~=1.4.2'
```

In this case, this means to install any version "==1.4.*" version that's also ">=1.4.2".

### 1.1.5 Source Distributions vs Wheels

*pip* can install from either *Source Distributions (sdist)* or *Wheels*, but if both are present on PyPI, pip will prefer a compatible *wheel*.

*Wheels* are a pre-built *distribution* format that provides faster installation compared to *Source Distributions (sdist)*, especially when a project contains compiled extensions.

If *pip* does not find a wheel to install, it will locally build a wheel and cache it for future installs, instead of rebuilding the source distribution in the future.

### 1.1.6 Upgrading packages

Upgrade an already installed *SomeProject* to the latest from PyPI.

```
pip install --upgrade SomeProject
```

### 1.1.7 Installing to the User Site

To install *packages* that are isolated to the current user, use the `--user` flag:

```
pip install --user SomeProject
```

For more information see the User Installs section from the pip docs.

Note that the `--user` flag has no effect when inside a virtual environment - all installation commands will affect the virtual environment.

### 1.1.8 Requirements files

Install a list of requirements specified in a Requirements File.

```
pip install -r requirements.txt
```

[4] The compatible release specifier was accepted in **PEP 440** and support was released in *setuptools* v8.0 and *pip* v6.0

### 1.1.9 Installing from VCS

Install a project from VCS in "editable" mode. For a full breakdown of the syntax, see pip's section on VCS Support.

```
pip install -e git+https://git.repo/some_pkg.git#egg=SomeProject           # from git
pip install -e hg+https://hg.repo/some_pkg#egg=SomeProject                 # from␣
↪mercurial
pip install -e svn+svn://svn.repo/some_pkg/trunk/#egg=SomeProject          # from svn
pip install -e git+https://git.repo/some_pkg.git@feature#egg=SomeProject   # from a␣
↪branch
```

### 1.1.10 Installing from other Indexes

Install from an alternate index

```
pip install --index-url http://my.package.repo/simple/ SomeProject
```

Search an additional index during install, in addition to *PyPI*

```
pip install --extra-index-url http://my.package.repo/simple SomeProject
```

### 1.1.11 Installing from a local src tree

Installing from local src in Development Mode, i.e. in such a way that the project appears to be installed, but yet is still editable from the src tree.

```
pip install -e <path>
```

You can also install normally from src

```
pip install <path>
```

### 1.1.12 Installing from local archives

Install a particular source archive file.

```
pip install ./downloads/SomeProject-1.0.4.tar.gz
```

Install from a local directory containing archives (and don't check *PyPI*)

```
pip install --no-index --find-links=file:///local/dir/ SomeProject
pip install --no-index --find-links=/local/dir/ SomeProject
pip install --no-index --find-links=relative/dir/ SomeProject
```

### 1.1.13 Installing from other sources

To install from other data sources (for example Amazon S3 storage) you can create a helper application that presents the data in a **PEP 503** compliant index format, and use the `--extra-index-url` flag to direct pip to use that index.

```
./s3helper --port=7777
pip install --extra-index-url http://localhost:7777 SomeProject
```

### 1.1.14 Installing Prereleases

Find pre-release and development versions, in addition to stable versions. By default, pip only finds stable versions.

```
pip install --pre SomeProject
```

### 1.1.15 Installing Setuptools "Extras"

Install setuptools extras.

```
$ pip install SomePackage[PDF]
$ pip install SomePackage[PDF]==3.0
$ pip install -e .[PDF]==3.0  # editable project in current directory
```

---

## 1.2 Managing Application Dependencies

The *package installation tutorial* covered the basics of getting set up to install and update Python packages.

However, running these commands interactively can get tedious even for your own personal projects, and things get even more difficult when trying to set up development environments automatically for projects with multiple contributors.

This tutorial walks you through the use of *Pipenv* to manage dependencies for an application. It will show you how to install and use the necessary tools and make strong recommendations on best practices.

Keep in mind that Python is used for a great many different purposes, and precisely how you want to manage your dependencies may change based on how you decide to publish your software. The guidance presented here is most directly applicable to the development and deployment of network services (including web applications), but is also very well suited to managing development and testing environments for any kind of project.

---

**Note:** This guide is written for Python 3, however, these instructions should also work on Python 2.7.

---

### 1.2.1 Installing Pipenv

*Pipenv* is a dependency manager for Python projects. If you're familiar with Node.js' npm or Ruby's bundler, it is similar in spirit to those tools. While *pip* alone is often sufficient for personal use, Pipenv is recommended for collaborative projects as it's a higher-level tool that simplifies dependency management for common use cases.

Use `pip` to install Pipenv:

```
pip install --user pipenv
```

---

**Note:** This does a user installation to prevent breaking any system-wide packages. If `pipenv` isn't available in your shell after installation, you'll need to add the user base's binary directory to your `PATH`.

On Linux and macOS you can find the user base binary directory by running `python -m site --user-base` and adding `bin` to the end. For example, this will typically print `~/.local` (with `~` expanded to the absolute path to your home directory) so you'll need to add `~/.local/bin` to your `PATH`. You can set your `PATH` permanently by modifying ~/.profile.

On Windows you can find the user base binary directory by running `py -m site --user-site` and replacing `site-packages` with `Scripts`. For example, this could return `C:\Users\Username\AppData\Roaming\Python36\site-packages` so you would need to set your `PATH` to include `C:\Users\Username\AppData\Roaming\Python36\Scripts`. You can set your user `PATH` permanently in the Control Panel. You may need to log out for the `PATH` changes to take effect.

### 1.2.2 Installing packages for your project

Pipenv manages dependencies on a per-project basis. To install packages, change into your project's directory (or just an empty directory for this tutorial) and run:

```
cd myproject
pipenv install requests
```

Pipenv will install the excellent Requests library and create a `Pipfile` for you in your project's directory. The *Pipfile* is used to track which dependencies your project needs in case you need to re-install them, such as when you share your project with others. You should get output similar to this (although the exact paths shown will vary):

```
Creating a Pipfile for this project...
Creating a virtualenv for this project...
Using base prefix '/usr/local/Cellar/python3/3.6.2/Frameworks/Python.framework/
↪Versions/3.6'
New python executable in ~/.local/share/virtualenvs/tmp-agwWamBd/bin/python3.6
Also creating executable in ~/.local/share/virtualenvs/tmp-agwWamBd/bin/python
Installing setuptools, pip, wheel...done.

Virtualenv location: ~/.local/share/virtualenvs/tmp-agwWamBd
Installing requests...
Collecting requests
  Using cached requests-2.18.4-py2.py3-none-any.whl
Collecting idna<2.7,>=2.5 (from requests)
  Using cached idna-2.6-py2.py3-none-any.whl
Collecting urllib3<1.23,>=1.21.1 (from requests)
  Using cached urllib3-1.22-py2.py3-none-any.whl
Collecting chardet<3.1.0,>=3.0.2 (from requests)
  Using cached chardet-3.0.4-py2.py3-none-any.whl
Collecting certifi>=2017.4.17 (from requests)
  Using cached certifi-2017.7.27.1-py2.py3-none-any.whl
Installing collected packages: idna, urllib3, chardet, certifi, requests
Successfully installed certifi-2017.7.27.1 chardet-3.0.4 idna-2.6 requests-2.18.4␣
↪urllib3-1.22

Adding requests to Pipfile's [packages]...
P.S. You have excellent taste!
```

### 1.2.3 Using installed packages

Now that Requests is installed you can create a simple `main.py` file to use it:

```
import requests

response = requests.get('https://httpbin.org/ip')

print('Your IP is {0}'.format(response.json()['origin']))
```

Then you can run this script using `pipenv run`:

```
pipenv run python main.py
```

You should get output similar to this:

```
Your IP is 8.8.8.8
```

Using `pipenv run` ensures that your installed packages are available to your script. It's also possible to spawn a new shell that ensures all commands have access to your installed packages with `pipenv shell`.

### 1.2.4 Next steps

Congratulations, you now know how to effectively manage dependencies and development environments on a collaborative Python project!

If you find this particular approach isn't working well for you or your use case, you may want to explore these other approaches:

- pip-tools
- hatch

If you're interesting in creating and distributing your own Python packages, see the *tutorial on packaging and distributing packages*.

## 1.3 Packaging and Distributing Projects

This section covers the basics of how to configure, package and distribute your own Python projects. It assumes that you are already familiar with the contents of the Installation page.

The section does *not* aim to cover best practices for Python project development as a whole. For example, it does not provide guidance or tool recommendations for version control, documentation, or testing.

For more reference material, see Building and Distributing Packages in the *setuptools* docs, but note that some advisory content there may be outdated. In the event of conflicts, prefer the advice in the Python Packaging User Guide.

**Contents**

### 1.3.1 Requirements for Packaging and Distributing

1. First, make sure you have already fulfilled the *requirements for installing packages*.
2. Install "twine"[1]:

```
pip install twine
```

   You'll need this to upload your project *distributions* to *PyPI* (see *below*).

### 1.3.2 Configuring your Project

#### Initial Files

#### setup.py

The most important file is "setup.py" which exists at the root of your project directory. For an example, see the setup.py in the PyPA sample project.

"setup.py" serves two primary functions:

1. It's the file where various aspects of your project are configured. The primary feature of `setup.py` is that it contains a global `setup()` function. The keyword arguments to this function are how specific details of your project are defined. The most relevant arguments are explained in *the section below*.
2. It's the command line interface for running various commands that relate to packaging tasks. To get a listing of available commands, run `python setup.py --help-commands`.

#### setup.cfg

"setup.cfg" is an ini file that contains option defaults for `setup.py` commands. For an example, see the setup.cfg in the PyPA sample project.

#### README.rst

All projects should contain a readme file that covers the goal of the project. The most common format is reStructured-Text with an "rst" extension, although this is not a requirement.

For an example, see README.rst from the PyPA sample project.

#### MANIFEST.in

A `MANIFEST.in` is needed when you need to package additional files that are not automatically included in a source distribution. To see a list of what's included by default, see the Specifying the files to distribute section from the *distutils* documentation.

For an example, see the MANIFEST.in from the PyPA sample project.

[1] Depending on your platform, this may require root or Administrator access. *pip* is currently considering changing this by making user installs the default behavior.

For details on writing a `MANIFEST.in` file, see the The MANIFEST.in template section from the *distutils* documentation.

**Note:** `MANIFEST.in` does not affect binary distributions such as wheels.

### LICENSE.txt

Every package should include a license file detailing the terms of distribution. In many jurisdictions, packages without an explicit license can not be legally used or distributed by anyone other than the copyright holder. If you're unsure which license to choose, you can use resources such as GitHub's Choose a License or consult a lawyer.

For an example, see the LICENSE.txt from the PyPA sample project.

### <your package>

Although it's not required, the most common practice is to include your Python modules and packages under a single top-level package that has the same *name* as your project, or something very close.

For an example, see the sample package that's included in the PyPA sample project.

### setup() args

As mentioned above, the primary feature of `setup.py` is that it contains a global `setup()` function. The keyword arguments to this function are how specific details of your project are defined.

The most relevant arguments are explained below. The snippets given are taken from the setup.py contained in the PyPA sample project.

### name

```
name='sample',
```

This is the name of your project, determining how your project is listed on *PyPI*. Per **PEP 508**, valid project names must:

- Consist only of ASCII letters, digits, underscores (_), hyphens (-), and/or periods (`.`), and
- Start & end with an ASCII letter or digit.

Comparison of project names is case insensitive and treats arbitrarily-long runs of underscores, hyphens, and/or periods as equal. For example, if you register a project named `cool-stuff`, users will be able to download it or declare a dependency on it using any of the following spellings:

```
Cool-Stuff
cool.stuff
COOL_STUFF
CoOl__-.-__sTuFF
```

### version

```
version='1.2.0',
```

This is the current version of your project, allowing your users to determine whether or not they have the latest version, and to indicate which specific versions they've tested their own software against.

Versions are displayed on *PyPI* for each release if you publish your project.

See *Choosing a versioning scheme* for more information on ways to use versions to convey compatibility information to your users.

If the project code itself needs run-time access to the version, the simplest way is to keep the version in both `setup.py` and your code. If you'd rather not duplicate the value, there are a few ways to manage this. See the "*Single-sourcing the package version*" Advanced Topics section.

### description

```
description='A sample Python project',
long_description=long_description,
```

Give a short and long description for your project. These values will be displayed on *PyPI* if you publish your project.

### url

```
url='https://github.com/pypa/sampleproject',
```

Give a homepage URL for your project.

### author

```
author='The Python Packaging Authority',
author_email='pypa-dev@googlegroups.com',
```

Provide details about the author.

### license

```
license='MIT',
```

Provide the type of license you are using.

### classifiers

```
classifiers=[
    # How mature is this project? Common values are
    #   3 - Alpha
    #   4 - Beta
    #   5 - Production/Stable
```

```
    'Development Status :: 3 - Alpha',

    # Indicate who your project is intended for
    'Intended Audience :: Developers',
    'Topic :: Software Development :: Build Tools',

    # Pick your license as you wish (should match "license" above)
     'License :: OSI Approved :: MIT License',

    # Specify the Python versions you support here. In particular, ensure
    # that you indicate whether you support Python 2, Python 3 or both.
    'Programming Language :: Python :: 2',
    'Programming Language :: Python :: 2.6',
    'Programming Language :: Python :: 2.7',
    'Programming Language :: Python :: 3',
    'Programming Language :: Python :: 3.2',
    'Programming Language :: Python :: 3.3',
    'Programming Language :: Python :: 3.4',
],
```

Provide a list of classifiers that categorize your project. For a full listing, see https://pypi.python.org/pypi?%3Aaction=list_classifiers.

Although the list of classifiers is often used to declare what Python versions a project supports, this information is only used for searching & browsing projects on PyPI, not for installing projects. To actually restrict what Python versions a project can be installed on, use the *python_requires* argument.

### keywords

```
keywords='sample setuptools development',
```

List keywords that describe your project.

### packages

```
packages=find_packages(exclude=['contrib', 'docs', 'tests*']),
```

It is required to list the *packages* to be included in your project. Although they can be listed manually, `setuptools.find_packages` finds them automatically. Use the `exclude` keyword argument to omit packages that are not intended to be released and installed.

### install_requires

```
install_requires=['peppercorn'],
```

"install_requires" should be used to specify what dependencies a project minimally needs to run. When the project is installed by *pip*, this is the specification that is used to install its dependencies.

For more on using "install_requires" see *install_requires vs Requirements files*.

### python_requires

If your project only runs on certain Python versions, setting the `python_requires` argument to the appropriate **PEP 440** version specifier string will prevent *pip* from installing the project on other Python versions. For example, if your package is for Python 3+ only, write:

```
python_requires='>=3',
```

If your package is for Python 3.3 and up but you're not willing to commit to Python 4 support yet, write:

```
python_requires='~=3.3',
```

If your package is for Python 2.6, 2.7, and all versions of Python 3 starting with 3.3, write:

```
python_requires='>=2.6, !=3.0.*, !=3.1.*, !=3.2.*, <4',
```

And so on.

---

**Note:** Support for this feature is relatively recent. Your project's source distributions and wheels (see *Packaging your Project*) must be built using at least version 24.2.0 of *setuptools* in order for the `python_requires` argument to be recognized and the appropriate metadata generated.

In addition, only versions 9.0.0 and higher of *pip* recognize the `python_requires` metadata. Users with earlier versions of pip will be able to download & install projects on any Python version regardless of the projects' `python_requires` values.

---

### package_data

```
package_data={
    'sample': ['package_data.dat'],
},
```

Often, additional files need to be installed into a *package*. These files are often data that's closely related to the package's implementation, or text files containing documentation that might be of interest to programmers using the package. These files are called "package data".

The value must be a mapping from package name to a list of relative path names that should be copied into the package. The paths are interpreted as relative to the directory containing the package.

For more information, see Including Data Files from the setuptools docs.

### data_files

```
data_files=[('my_data', ['data/data_file'])],
```

Although configuring *package_data* is sufficient for most needs, in some cases you may need to place data files *outside* of your *packages*. The `data_files` directive allows you to do that.

Each (directory, files) pair in the sequence specifies the installation directory and the files to install there. If directory is a relative path, it is interpreted relative to the installation prefix (Python's sys.prefix for pure-Python *distributions*, sys.exec_prefix for distributions that contain extension modules). Each file name in files is interpreted relative to the `setup.py` script at the top of the project source distribution.

For more information see the distutils section on Installing Additional Files.

**Note:** *setuptools* allows absolute "data_files" paths, and pip honors them as absolute, when installing from *sdist*. This is not true when installing from *wheel* distributions. Wheels don't support absolute paths, and they end up being installed relative to "site-packages". For discussion see wheel Issue #92.

### scripts

Although `setup()` supports a scripts keyword for pointing to pre-made scripts to install, the recommended approach to achieve cross-platform compatibility is to use *console_scripts* entry points (see below).

### py_modules

```
py_modules=["six"],
```

It is required to list the names of single file modules that are to be included in your project.

### entry_points

```
entry_points={
  ...
},
```

Use this keyword to specify any plugins that your project provides for any named entry points that may be defined by your project or others that you depend on.

For more information, see the section on Dynamic Discovery of Services and Plugins from the *setuptools* docs.

The most commonly used entry point is "console_scripts" (see below).

### console_scripts

```
entry_points={
    'console_scripts': [
        'sample=sample:main',
    ],
},
```

Use "console_script" entry points to register your script interfaces. You can then let the toolchain handle the work of turning these interfaces into actual scripts[2]. The scripts will be generated during the install of your *distribution*.

For more information, see Automatic Script Creation from the setuptools docs.

[2] Specifically, the "console_script" approach generates `.exe` files on Windows, which are necessary because the OS special-cases `.exe` files. Script-execution features like `PATHEXT` and the **Python Launcher for Windows** allow scripts to be used in many cases, but not all.

### Choosing a versioning scheme

#### Standards compliance for interoperability

Different Python projects may use different versioning schemes based on the needs of that particular project, but all of them are required to comply with the flexible **public version scheme** specified in **PEP 440** in order to be supported in tools and libraries like `pip` and `setuptools`.

Here are some examples of compliant version numbers:

```
1.2.0.dev1  # Development release
1.2.0a1     # Alpha Release
1.2.0b1     # Beta Release
1.2.0rc1    # Release Candidate
1.2.0       # Final Release
1.2.0.post1 # Post Release
15.10       # Date based release
23          # Serial release
```

To further accommodate historical variations in approaches to version numbering, **PEP 440** also defines a comprehensive technique for **version normalisation** that maps variant spellings of different version numbers to a standardised canonical form.

#### Scheme choices

#### Semantic versioning (preferred)

For new projects, the recommended versioning scheme is based on Semantic Versioning, but adopts a different approach to handling pre-releases and build metadata.

The essence of semantic versioning is a 3-part MAJOR.MINOR.MAINTENANCE numbering scheme, where the project author increments:

1. MAJOR version when they make incompatible API changes,
2. MINOR version when they add functionality in a backwards-compatible manner, and
3. MAINTENANCE version when they make backwards-compatible bug fixes.

Adopting this approach as a project author allows users to make use of **"compatible release"** specifiers, where `name ~= X.Y` requires at least release X.Y, but also allows any later release with a matching MAJOR version.

Python projects adopting semantic versioning should abide by clauses 1-8 of the Semantic Versioning 2.0.0 specification.

#### Date based versioning

Semantic versioning is not a suitable choice for all projects, such as those with a regular time based release cadence and a deprecation process that provides warnings for a number of releases prior to removal of a feature.

A key advantage of date based versioning is that it is straightforward to tell how old the base feature set of a particular release is given just the version number.

Version numbers for date based projects typically take the form of YEAR.MONTH (for example, `12.04`, `15.10`).

#### Serial versioning

This is the simplest possible versioning scheme, and consists of a single number which is incremented every release.

While serial versioning is very easy to manage as a developer, it is the hardest to track as an end user, as serial version numbers convey little or no information regarding API backwards compatibility.

#### Hybrid schemes

Combinations of the above schemes are possible. For example, a project may combine date based versioning with serial versioning to create a YEAR.SERIAL numbering scheme that readily conveys the approximate age of a release, but doesn't otherwise commit to a particular release cadence within the year.

#### Pre-release versioning

Regardless of the base versioning scheme, pre-releases for a given final release may be published as:

- zero or more dev releases (denoted with a ".devN" suffix)
- zero or more alpha releases (denoted with a ".aN" suffix)
- zero or more beta releases (denoted with a ".bN" suffix)
- zero or more release candidates (denoted with a ".rcN" suffix)

`pip` and other modern Python package installers ignore pre-releases by default when deciding which versions of dependencies to install.

#### Local version identifiers

Public version identifiers are designed to support distribution via *PyPI*. Python's software distribution tools also support the notion of a **local version identifier**, which can be used to identify local development builds not intended for publication, or modified variants of a release maintained by a redistributor.

A local version identifier takes the form `<public version identifier>+<local version label>`. For example:

```
1.2.0.dev1+hg.5.b11e5e6f0b0b  # 5th VCS commmit since 1.2.0.dev1 release
1.2.1+fedora.4                # Package with downstream Fedora patches applied
```

### 1.3.3 Working in "Development Mode"

Although not required, it's common to locally install your project in "editable" or "develop" mode while you're working on it. This allows your project to be both installed and editable in project form.

Assuming you're in the root of your project directory, then run:

```
pip install -e .
```

Although somewhat cryptic, `-e` is short for `--editable`, and `.` refers to the current working directory, so together, it means to install the current directory (i.e. your project) in editable mode. This will also install any dependencies declared with "install_requires" and any scripts declared with "console_scripts". Dependencies will be installed in the usual, non-editable mode.

It's fairly common to also want to install some of your dependencies in editable mode as well. For example, supposing your project requires "foo" and "bar", but you want "bar" installed from VCS in editable mode, then you could construct a requirements file like so:

```
-e .
-e git+https://somerepo/bar.git#egg=bar
```

The first line says to install your project and any dependencies. The second line overrides the "bar" dependency, such that it's fulfilled from VCS, not PyPI.

If, however, you want "bar" installed from a local directory in editable mode, the requirements file should look like this, with the local paths at the top of the file:

```
-e /path/to/project/bar
-e .
```

Otherwise, the dependency will be fulfilled from PyPI, due to the installation order of the requirements file. For more on requirements files, see the Requirements File section in the pip docs. For more on VCS installs, see the VCS Support section of the pip docs.

Lastly, if you don't want to install any dependencies at all, you can run:

```
pip install -e . --no-deps
```

For more information, see the Development Mode section of the setuptools docs.

### 1.3.4 Packaging your Project

To have your project installable from a *Package Index* like *PyPI*, you'll need to create a *Distribution* (aka "*Package*") for your project.

#### Source Distributions

Minimally, you should create a *Source Distribution*:

```
python setup.py sdist
```

A "source distribution" is unbuilt (i.e. it's not a *Built Distribution*), and requires a build step when installed by pip. Even if the distribution is pure Python (i.e. contains no extensions), it still involves a build step to build out the installation metadata from `setup.py`.

#### Wheels

You should also create a wheel for your project. A wheel is a *built package* that can be installed without needing to go through the "build" process. Installing wheels is substantially faster for the end user than installing from a source distribution.

If your project is pure Python (i.e. contains no compiled extensions) and natively supports both Python 2 and 3, then you'll be creating what's called a **Universal Wheel* (see section below)*.

If your project is pure Python but does not natively support both Python 2 and 3, then you'll be creating a *"Pure Python Wheel" (see section below)*.

If your project contains compiled extensions, then you'll be creating what's called a **Platform Wheel* (see section below)*.

Before you can build wheels for your project, you'll need to install the `wheel` package:

```
pip install wheel
```

### Universal Wheels

*Universal Wheels* are wheels that are pure Python (i.e. contain no compiled extensions) and support Python 2 and 3. This is a wheel that can be installed anywhere by *pip*.

To build the wheel:

```
python setup.py bdist_wheel --universal
```

You can also permanently set the `--universal` flag in "setup.cfg" (e.g., see sampleproject/setup.cfg):

```
[bdist_wheel]
universal=1
```

Only use the `--universal` setting, if:

1. Your project runs on Python 2 and 3 with no changes (i.e. it does not require 2to3).
2. Your project does not have any C extensions.

Beware that `bdist_wheel` does not currently have any checks to warn if you use the setting inappropriately.

If your project has optional C extensions, it is recommended not to publish a universal wheel, because pip will prefer the wheel over a source installation, and prevent the possibility of building the extension.

### Pure Python Wheels

*Pure Python Wheels* that are not "universal" are wheels that are pure Python (i.e. contain no compiled extensions), but don't natively support both Python 2 and 3.

To build the wheel:

```
python setup.py bdist_wheel
```

*bdist_wheel* will detect that the code is pure Python, and build a wheel that's named such that it's usable on any Python installation with the same major version (Python 2 or Python 3) as the version you used to build the wheel. For details on the naming of wheel files, see **PEP 425**.

If your code supports both Python 2 and 3, but with different code (e.g., you use "2to3") you can run `setup.py bdist_wheel` twice, once with Python 2 and once with Python 3. This will produce wheels for each version.

### Platform Wheels

*Platform Wheels* are wheels that are specific to a certain platform like Linux, macOS, or Windows, usually due to containing compiled extensions.

To build the wheel:

```
python setup.py bdist_wheel
```

*bdist_wheel* will detect that the code is not pure Python, and build a wheel that's named such that it's only usable on the platform that it was built on. For details on the naming of wheel files, see **PEP 425**.

---

**Note:** *PyPI* currently supports uploads of platform wheels for Windows, macOS, and the multi-distro `manylinux1` ABI. Details of the latter are defined in **PEP 513**.

---

### 1.3.5 Uploading your Project to PyPI

When you ran the command to create your distribution, a new directory `dist/` was created under your project's root directory. That's where you'll find your distribution file(s) to upload.

---

**Note:** These files are only created when you run the command to create your distribution. This means that any time you change the source of your project or the configuration in your `setup.py` file, you will need to rebuild these files again before you can distribute the changes to PyPI.

---

---

**Note:** Before releasing on main PyPI repo, you might prefer training with the PyPI test site which is cleaned on a semi regular basis. See *Using TestPyPI* on how to setup your configuration in order to use it.

---

**Warning:** In other resources you may encounter references to using `python setup.py register` and `python setup.py upload`. These methods of registering and uploading a package are **strongly discouraged** as it may use a plaintext HTTP or unverified HTTPS connection on some Python versions, allowing your username and password to be intercepted during transmission.

---

**Tip:** The reStructuredText parser used on PyPI is **not** Sphinx! Furthermore, to ensure safety of all users, certain kinds of URLs and directives are forbidden or stripped out (e.g., the `.. raw::` directive). **Before** trying to upload your distribution, you should check to see if your brief / long descriptions provided in `setup.py` are valid. You can do this by following the instructions for the pypa/readme_renderer tool.

---

#### Create an account

First, you need a *PyPI* user account. You can create an account using the form on the PyPI website.

---

**Note:** If you want to avoid entering your username and password when uploading, you can create a `$HOME/.pypirc` file with your username and password:

```
[pypi]
username = <username>
password = <password>
```

**Be aware that this stores your password in plaintext.**

---

### Upload your distributions

Once you have an account you can upload your distributions to *PyPI* using *twine*. If this is your first time uploading a distribution for a new project, twine will handle registering the project.

```
twine upload dist/*
```

---

**Note:** Twine allows you to pre-sign your distribution files using gpg:

```
gpg --detach-sign -a dist/package-1.0.1.tar.gz
```

and pass the gpg-created .asc files into the command line invocation:

```
twine upload dist/package-1.0.1.tar.gz package-1.0.1.tar.gz.asc
```

This enables you to be assured that you're only ever typing your gpg passphrase into gpg itself and not anything else since *you* will be the one directly executing the `gpg` command.

---

---

CHAPTER 2

# Guides

**Guides** are focused on accomplishing a specific task and assume that you are already familiar with the basics of Python packaging. If you're looking for an introduction to packaging, see *Tutorials*.

## 2.1 Tool recommendations

If you're familiar with Python packaging and installation, and just want to know what tools are currently recommended, then here it is.

### 2.1.1 Installation Tool Recommendations

- Use *pip* to install Python *packages* from *PyPI*.[1][2] Depending on how *pip* is installed, you may need to also install *wheel* to get the benefit of wheel caching.[3]
- Use *virtualenv*, or venv to isolate application specific dependencies from a shared Python installation.[4]
- If you're looking for management of fully integrated cross-platform software stacks, consider:
  - *buildout*: primarily focused on the web development community
  - *Spack*, *Hashdist*, or *conda*: primarily focused on the scientific community.

[1] There are some cases where you might choose to use `easy_install` (from *setuptools*), e.g. if you need to install from *Eggs* (which pip doesn't support). For a detailed breakdown, see *pip vs easy_install*.

[2] The acceptance of **PEP 453** means that *pip* will be available by default in most installations of Python 3.4 or later. See the **rationale section** from **PEP 453** as for why pip was chosen.

[3] get-pip.py and *virtualenv* install *wheel*, whereas *ensurepip* and *venv* do not currently. Also, the common "python-pip" package that's found in various linux distros, does not depend on "python-wheel" currently.

[4] Beginning with Python 3.4, `venv` will create virtualenv environments with `pip` installed, thereby making it an equal alternative to *virtualenv*. However, using *virtualenv* will still be recommended for users that need cross-version consistency.

### 2.1.2 Packaging Tool Recommendations

- Use *setuptools* to define projects and create *Source Distributions*.[5][6]
- Use the `bdist_wheel` *setuptools* extension available from the *wheel project* to create *wheels*. This is especially beneficial, if your project contains binary extensions.[7]
- Use twine for uploading distributions to *PyPI*.

### 2.1.3 Publishing Platform Migration

The original Python Package Index implementation (hosted at pypi.python.org) is being phased out in favour of an updated implementation hosted at pypi.org. Both interfaces share a common database backend and file store, allowing the latter to assume more default responsibilities as it becomes more capable.

Users consistently using the latest versions of the recommended tools above with their default settings don't need to worry about this migration, but users running older versions, or not using the default settings, will need to update their configurations in line with the recommendations below.

#### Publishing releases

`pypi.org` became the default upload platform in September 2016.

Uploads through `pypi.python.org` will be *switched off* on **July 3, 2017**.

The default upload settings switched to `pypi.org` in the following versions:

- `twine` 1.8.0
- `setuptools` 27.0.0
- Python 2.7.13 (`distutils` update)
- Python 3.4.6 (`distutils` update)
- Python 3.5.3 (`distutils` update)
- Python 3.6.0 (`distutils` update)

#### Browsing packages

`pypi.python.org` is currently still the default interface for browsing packages (used in links from other PyPA documentation, etc).

`pypi.org` is fully functional for purposes of browsing available packages, and some users may choose to opt in to using it.

`pypi.org` is expected to become the default recommended interface for browsing once the limitations in the next two sections are addressed (at which point attempts to access `pypi.python.org` will automatically be redirected to `pypi.org`).

[5] Although you can use pure `distutils` for many projects, it does not support defining dependencies on other projects and is missing several convenience utilities for automatically populating distribution metadata correctly that are provided by `setuptools`. Being outside the standard library, `setuptools` also offers a more consistent feature set across different versions of Python, and (unlike `distutils`), `setuptools` will be updated to produce the upcoming "Metadata 2.0" standard formats on all supported versions.

Even for projects that do choose to use `distutils`, when *pip* installs such projects directly from source (rather than installing from a prebuilt *wheel* file), it will actually build your project using *setuptools* instead.

[6] distribute (a fork of setuptools) was merged back into *setuptools* in June 2013, thereby making setuptools the default choice for packaging.

[7] *PyPI* currently only allows uploading Windows and macOS wheels, and they should be compatible with the binary installers provided for download from python.org. Enhancements will have to be made to the **wheel compatibility tagging scheme** before linux wheels will be allowed.

#### Downloading packages

`pypi.python.org` is currently still the default host for downloading packages.

`pypi.org` is fully functional for purposes of downloading packages, and some users may choose to opt in to using it, but its current hosting setup isn't capable of handling the full bandwidth requirements of being the default download source (even after accounting for the Fastly CDN).

`pypi.org` is expected to become the default host for downloading packages once it has been redeployed into an environment capable of handling the associated network load.

#### Managing published packages and releases

`pypi.python.org` provides an interface for logged in users to manage their published packages and releases.

`pypi.org` does not currently provide such an interface.

The missing capabilities are being tracked as part of the Shut Down Legacy PyPI milestone.

---

## 2.2 Installing packages using pip and virtualenv

This guide discusses how to install packages using *pip* and *virtualenv*. These are the lowest-level tools for managing Python packages and are recommended if higher-level tools do not suit your needs.

---

**Note:** This doc uses the term **package** to refer to a *Distribution Package* which is different from a *Import Package* that which is used to import modules in your Python source code.

---

### 2.2.1 Installing pip

*pip* is the reference Python package manager. It's used to install and update packages. You'll need to make sure you have the latest version of pip installed.

#### Windows

The Python installers for Windows include pip. You should be able to access pip using:

```
py -m pip --version
pip 9.0.1 from c:\python36\lib\site-packages (Python 3.6.1)
```

You can make sure that pip is up-to-date by running:

```
py -m pip install --upgrade pip
```

#### Linux and macOS

Debian and most other distributions include a python-pip package, if you want to use the Linux distribution-provided versions of pip see *Installing pip/setuptools/wheel with Linux Package Managers*.

You can also install pip yourself to ensure you have the latest version. It's recommended to use the system pip to bootstrap a user installation of pip:

```
python3 -m pip install --user --upgrade pip
```

Afterwards, you should have the newest pip installed in your user site:

```
python3 -m pip --version
pip 9.0.1 from $HOME/.local/lib/python3.6/site-packages (python 3.6)
```

### 2.2.2 Installing virtualenv

*virtualenv* is used to manage Python packages for different projects. Using virtualenv allows you to avoid installing Python packages globally which could break system tools or other projects. You can install virtualenv using pip.

On macOS and Linux:

```
python3 -m pip install --user virtualenv
```

On Windows:

```
py -m pip install --user virtualenv
```

**Note:** If you are using Python 3.3 or newer the `venv` module is included in the Python standard library. This can also create and manage virtual environments, however, it only supports Python 3.

### 2.2.3 Creating a virtualenv

*virtualenv* allows you to manage separate package installations for different projects. It essentially allows you to create a "virtual" isolated Python installation and install packages into that virtual installation. When you switch projects, you can simply create a new virtual environment and not have to worry about breaking the packages installed in the other environments. It is always recommended to use a virtualenv while developing Python applications.

To create a virtual environment, go to your project's directory and run virtualenv.

On macOS and Linux:

```
python3 -m virtualenv env
```

On Windows:

```
py -m virtualenv env
```

The second argument is the location to create the virtualenv. Generally, you can just create this in your project and call it `env`.

virtualenv will create a virtual Python installation in the `env` folder.

**Note:** You should exclude your virtualenv directory from your version control system using `.gitignore` or similar.

### 2.2.4 Activating a virtualenv

Before you can start installing or using packages in your virtualenv you'll need to *activate* it. Activating a virtualenv will put the virtualenv-specific `python` and `pip` executables into your shell's `PATH`.

On macOS and Linux:

```
source env/bin/activate
```

On Windows:

```
.\env\Scripts\activate
```

You can confirm you're in the virtualenv by checking the location of your Python interpreter, it should point to the `env` directory.

On macOS and Linux:

```
which python
.../env/bin/python
```

On Windows:

```
where python
.../env/bin/python.exe
```

As long as your virtualenv is activated pip will install packages into that specific environment and you'll be able to import and use packages in your Python application.

### 2.2.5 Leaving the virtualenv

If you want to switch projects or otherwise leave your virtualenv, simply run:

```
deactivate
```

If you want to re-enter the virtualenv just follow the same instructions above about activating a virtualenv. There's no need to re-create the virtualenv.

### 2.2.6 Installing packages

Now that you're in your virtualenv you can install packages. Let's install the excellent Requests library from the *Python Package Index (PyPI)*:

```
pip install requests
```

pip should download requests and all of its dependencies and install them:

```
Collecting requests
  Using cached requests-2.18.4-py2.py3-none-any.whl
Collecting chardet<3.1.0,>=3.0.2 (from requests)
```

```
  Using cached chardet-3.0.4-py2.py3-none-any.whl
Collecting urllib3<1.23,>=1.21.1 (from requests)
  Using cached urllib3-1.22-py2.py3-none-any.whl
Collecting certifi>=2017.4.17 (from requests)
  Using cached certifi-2017.7.27.1-py2.py3-none-any.whl
Collecting idna<2.7,>=2.5 (from requests)
  Using cached idna-2.6-py2.py3-none-any.whl
Installing collected packages: chardet, urllib3, certifi, idna, requests
Successfully installed certifi-2017.7.27.1 chardet-3.0.4 idna-2.6 requests-2.18.4␣
↪urllib3-1.22
```

### 2.2.7 Installing specific versions

pip allows you to specify which version of a package to install using *version specifiers*. For example, to install a specific version of `requests`:

```
pip install requests==2.18.4
```

To install the latest `2.x` release of requests:

```
pip install requests>=2.0.0,<3.0.0
```

To install pre-release versions of packages, use the `--pre` flag:

```
pip install --pre requests
```

### 2.2.8 Installing extras

Some packages have optioanl extras. You can tell pip to install these by specifying the extra in brackets:

```
pip install requests[security]
```

### 2.2.9 Installing from source

pip can install a package directly from source, for example:

```
cd google-auth
pip install .
```

Additionally, pip can install packages from source in development mode, meaning that changes to the source directory will immediately affect the installed package without needing to re-install:

```
pip install --editable .
```

### 2.2.10 Installing from version control systems

pip can install packages directly from their version control system. For example, you can install directly from a git repository:

```
git+https://github.com/GoogleCloudPlatform/google-auth-library-python.git#egg=google-
↪auth
```

For more information on supported version control systems and syntax, see pip's documentation on VCS Support.

### 2.2.11 Installing from local archives

If you have a local copy of a *Distribution Package*'s archive (a zip, wheel, or tar file) you can install it directly with pip:

```
pip install requests-2.18.4.tar.gz
```

If you have a directory containing archives of multiple packages, you can tell pip to look for packages there and not to use the *Python Package Index (PyPI)* at all:

```
pip install --no-index --find-links=/local/dir/ requests
```

This is useful if you are installing packages on a system with limited connectivity or if you want to strictly control the origin of distribution packages.

### 2.2.12 Using other package indexes

If you want to download packages from a different index than the *Python Package Index (PyPI)*, you can use the `--index-url` flag:

```
pip install --index-url http://index.example.com/simple/ SomeProject
```

If you want to allow packages from both the *Python Package Index (PyPI)* and a separate index, you can use the `--extra-index-url` flag instead:

```
pip install --extra-index-url http://index.example.com/simple/ SomeProject
```

### 2.2.13 Upgrading packages

pip can upgrade packages in-place using the `--upgrade` flag. For example, to install the latest version of `requests` and all of its dependencies:

```
pip install --upgrade requests
```

### 2.2.14 Using requirements files

Instead of installing packages individually, pip allows you to declare all dependencies in a Requirements File. For example you could create a `requirements.txt` file containing:

```
requests==2.18.4
google-auth==1.1.0
```

And tell pip too install all of the packages in this file using the `-r` flag:

```
pip install -r requirements.txt
```

### 2.2.15 Freezing dependencies

Pip can export a list of all installed packages and their versions using the `freeze` command:

```
pip freeze
```

Which will output a list of package specifiers such as:

```
cachetools==2.0.1
certifi==2017.7.27.1
chardet==3.0.4
google-auth==1.1.1
idna==2.6
pyasn1==0.3.6
pyasn1-modules==0.1.4
requests==2.18.4
rsa==3.4.2
six==1.11.0
urllib3==1.22
```

This is useful for creating Requirements Files that can re-create the exact versions of all packages installed in an environment.

## 2.3 Installing pip/setuptools/wheel with Linux Package Managers

**Page Status** Incomplete

**Last Reviewed** 2015-09-17

This section covers how to install *pip*, *setuptools*, and *wheel* using Linux package managers.

If you're using a Python that was downloaded from python.org, then this section does not apply. See the *Requirements for Installing Packages* section instead.

Note that it's common for the versions of *pip*, *setuptools*, and *wheel* supported by a specific Linux Distribution to be outdated by the time it's released to the public, and updates generally only occur for security reasons, not for feature updates. For certain Distributions, there are additional repositories that can be enabled to provide newer versions. The repositories we know about are explained below.

Also note that it's somewhat common for Distributions to apply patches for the sake of security and normalization to their own standards. In some cases, this can lead to bugs or unexpected behaviors that vary from the original unpatched versions. When this is known, we will make note of it below.

### 2.3.1 Fedora

- Fedora 21:
  - Python 2:

    ```
    sudo yum upgrade python-setuptools
    sudo yum install python-pip python-wheel
    ```

  - Python 3: `sudo yum install python3 python3-wheel`
- Fedora 22:
  - Python 2:

```
sudo dnf upgrade python-setuptools
sudo dnf install python-pip python-wheel
```

- Python 3: `sudo dnf install python3 python3-wheel`

To get newer versions of pip, setuptools, and wheel for Python 2, you can enable the PyPA Copr Repo using the Copr Repo instructions, and then run:

```
sudo yum|dnf upgrade python-setuptools
sudo yum|dnf install python-pip python-wheel
```

### 2.3.2 CentOS/RHEL

CentOS and RHEL don't offer *pip* or *wheel* in their core repositories, although *setuptools* is installed by default.

To install pip and wheel for the system Python, there are two options:

1. Enable the EPEL repository using these instructions. On EPEL 6 and EPEL7, you can install pip like so:

   ```
   sudo yum install python-pip
   ```

   On EPEL 7 (but not EPEL 6), you can install wheel like so:

   ```
   sudo yum install python-wheel
   ```

   Since EPEL only offers extra, non-conflicting packages, EPEL does not offer setuptools, since it's in the core repository.

2. Enable the PyPA Copr Repo using these instructions[1]. You can install pip and wheel like so:

   ```
   sudo yum install python-pip python-wheel
   ```

   To additionally upgrade setuptools, run:

   ```
   sudo yum upgrade python-setuptools
   ```

To install pip, wheel, and setuptools, in a parallel, non-system environment (using yum) then there are two options:

1. Use the "Sofware Collections" feature to enable a parallel collection that includes pip, setuptools, and wheel.

   - For Redhat, see here: http://developers.redhat.com/products/softwarecollections/overview/
   - For CentOS, see here: https://www.softwarecollections.org/en/

   Be aware that collections may not contain the most recent versions.

2. Enable the IUS repository and install one of the parallel-installable Pythons, along with pip, setuptools, and wheel, which are kept fairly up to date.

   For example, for Python 3.4 on CentOS7/RHEL7:

   ```
   sudo yum install python34u python34u-wheel
   ```

[1] Currently, there is no "copr" yum plugin available for CentOS/RHEL, so the only option is to manually place the repo files as described.

### 2.3.3 openSUSE

- Python 2:

```
sudo zypper install python-pip python-setuptools python-wheel
```

- Python 3:

```
sudo zypper install python3-pip python3-setuptools python3-wheel
```

### 2.3.4 Debian/Ubuntu

```
sudo apt-get install python-pip
```

Replace "python" with "python3" for Python 3.

**Warning:** Recent Debian/Ubuntu versions have modified pip to use the "User Scheme" by default, which is a significant behavior change that can be surprising to some users.

### 2.3.5 Arch Linux

- Python 2:

```
sudo pacman -S python2-pip
```

- Python 3:

```
sudo pacman -S python-pip
```

## 2.4 Installing Scientific Packages

**Page Status** Incomplete

**Last Reviewed** 2014-07-24

**Contents**

Scientific software tends to have more complex dependencies than most, and it will often have multiple build options to take advantage of different kinds of hardware, or to interoperate with different pieces of external software.

In particular, NumPy, which provides the basis for most of the software in the scientific Python stack can be configured to interoperate with different FORTRAN libraries, and can take advantage of different levels of vectorised instructions available in modern CPUs.

Unfortunately, as of December 2013, given NumPy's current build and distribution model, the standard tools currently aren't quite up to the task of distributing pre-built NumPy binaries, as most users aren't going to know which version they need, and the `wheel` format currently doesn't allow the installer to make that decision on the user's behalf at install time.

It is expected that this situation will eventually be resolved either by future iterations of the standard tools providing full support for the intricacies of NumPy's current build and distribution process, or by the NumPy developers choosing one build variant as the "lowest acceptable common denominator" and publishing that as a wheel file on PyPI.

In the meantime, however, there are a number of alternative options for obtaining scientific Python libraries (or any other Python libraries that require a compilation environment to install from source and don't provide pre-built wheel files on PyPI).

### 2.4.1 Building from source

The same complexity which makes it difficult to distribute NumPy (and many of the projects that depend on it) as wheel files also make them difficult to build from source yourself. However, for intrepid folks that are willing to spend the time wrangling compilers and linkers for both C and FORTRAN, building from source is always an option.

### 2.4.2 Linux distribution packages

For Linux users, the system package manager will often have pre-compiled versions of various pieces of scientific software, including NumPy and other parts of the scientific Python stack.

If using versions which may be several months old is acceptable, then this is likely to be a good option (just make sure to allow access to distributions installed into the system Python when using virtual environments).

### 2.4.3 Windows installers

Many Python projects that don't (or can't) currently publish wheel files at least publish Windows installers, either on PyPI or on their project download page. Using these installers allows users to avoid the need to set up a suitable environment to build extensions locally.

The extensions provided in these installers are typically compatible with the CPython Windows installers published on python.org.

For projects which don't provide their own Windows installers (and even some which do), Christoph Gohlke at the University of California provides a collection of Windows installers. Many Python users on Windows have reported a positive experience with these prebuilt versions.

As with Linux system packages, the Windows installers will only install into a system Python installation - they do not support installation in virtual environments. Allowing access to distributions installed into the system Python when using virtual environments is a common approach to working around this limitation.

The *wheel* project also provides a *wheel convert* subcommand that can convert a Windows *bdist_wininst* installer to a wheel.

### 2.4.4 macOS installers and package managers

Similar to the situation on Windows, many projects (including NumPy) publish macOS installers that are compatible with the macOS CPython binaries published on python.org.

macOS users also have access to Linux distribution style package managers such as `MacPorts`. The SciPy site has more details on using MacPorts to install the scientific Python stack

### 2.4.5 SciPy distributions

The SciPy site lists several distributions that provide the full SciPy stack to end users in an easy to use and update format.

Some of these distributions may not be compatible with the standard `pip` and `virtualenv` based toolchain.

### 2.4.6 Spack

Spack is a flexible package manager designed to support multiple versions, configurations, platforms, and compilers. It was built to support the needs of large supercomputing centers and scientific application teams, who must often build software many different ways. Spack is not limited to Python; it can install packages for `C`, `C++`, `Fortran`, `R`, and other languages. It is non-destructive; installing a new version of one package does not break existing installations, so many configurations can coexist on the same system.

Spack offers a simple but powerful syntax that allows users to specify versions and configuration options concisely. Package files are written in pure Python, and they are templated so that it is easy to swap compilers, dependency implementations (like MPI), versions, and build options with a single package file. Spack also generates *module* files so that packages can be loaded and unloaded from the user's environment.

### 2.4.7 The conda cross-platform package manager

Anaconda is a Python distribution published by Continuum Analytics. It is a stable collection of Open Source packages for big data and scientific use. About 100 are installed with Anaconda 2.2, and a total of 279 can be installed and updated from the Anaconda repository.

`conda` an open source (BSD licensed) package management system and environment management system included in Anaconda that allows users to install multiple versions of binary software packages and their dependencies, and easily switch between them. It is a cross-platform tool working on Windows, macOS, and Linux. Conda can be used to package up and distribute all kinds of packages, it is not limited to just Python packages. It has full support for native virtual environments. Conda makes environments first-class citizens, making it easy to create independent environments even for C libraries. It is written in Python, but is Python-agnostic. Conda manages Python itself as a package, so that *conda update python* is possible, in contrast to pip, which only manages Python packages. Conda is available in Anaconda and Miniconda (an easy-to-install download with just Python and conda).

## 2.5 Multi-version Installs

easy_install allows simultaneous installation of different versions of the same project into a single environment shared by multiple programs which must `require` the appropriate version of the project at run time (using `pkg_resources`).

For many use cases, virtual environments address this need without the complication of the `require` directive. However, the advantage of parallel installations within the same environment is that it works for an environment shared by multiple applications, such as the system Python in a Linux distribution.

The major limitation of `pkg_resources` based parallel installation is that as soon as you import `pkg_resources` it locks in the *default* version of everything which is already available on sys.path. This can cause problems, since `setuptools` created command line scripts use `pkg_resources` to find the entry point to execute. This means that, for example, you can't use `require` tests invoked through `nose` or a WSGI application invoked through `gunicorn` if your application needs a non-default version of anything that is available on the standard `sys.path` - the script wrapper for the main application will lock in the version that is available by default, so the subsequent `require` call in your own code fails with a spurious version conflict.

This can be worked around by setting all dependencies in `__main__.__requires__` before importing `pkg_resources` for the first time, but that approach does mean that standard command line invocations of the affected tools can't be used - it's necessary to write a custom wrapper script or use `python -c '<commmand>'` to invoke the application's main entry point directly.

Refer to the pkg_resources documentation for more details.

## 2.6 Single-sourcing the package version

There are many techniques to maintain a single source of truth for the version number of your project:

1. Read the file in `setup.py` and parse the version with a regex. Example ( from pip setup.py):

```
here = os.path.abspath(os.path.dirname(__file__))

def read(*parts):
    with codecs.open(os.path.join(here, *parts), 'r') as fp:
        return fp.read()

def find_version(*file_paths):
    version_file = read(*file_paths)
    version_match = re.search(r"^__version__ = ['\"]([^'\"]*)['\"]",
                              version_file, re.M)
    if version_match:
        return version_match.group(1)
    raise RuntimeError("Unable to find version string.")

setup(
   ...
   version=find_version("package", "__init__.py")
   ...
)
```

   **Note:** This technique has the disadvantage of having to deal with complexities of regular expressions.

2. Use an external build tool that either manages updating both locations, or offers an API that both locations can use.

   Few tools you could use, in no particular order, and not necessarily complete: bumpversion, changes, zest.releaser.

3. Set the value to a `__version__` global variable in a dedicated module in your project (e.g. `version.py`), then have `setup.py` read and `exec` the value into a variable.

   Using `execfile`:

```
execfile('...sample/version.py')
# now we have a `__version__` variable
# later on we use: __version__
```

Using `exec`:

```
version = {}
with open("...sample/version.py") as fp:
    exec(fp.read(), version)
# later on we use: version['__version__']
```

Example using this technique: warehouse.

4. Place the value in a simple `VERSION` text file and have both `setup.py` and the project code read it.

```
with open(os.path.join(mypackage_root_dir, 'VERSION')) as version_file:
    version = version_file.read().strip()
```

An advantage with this technique is that it's not specific to Python. Any tool can read the version.

> **Warning:** With this approach you must make sure that the `VERSION` file is included in all your source and binary distributions (e.g. add `include VERSION` to your `MANIFEST.in`).

5. Set the value in `setup.py`, and have the project code use the `pkg_resources` API.

```
import pkg_resources
assert pkg_resources.get_distribution('pip').version == '1.2.0'
```

Be aware that the `pkg_resources` API only knows about what's in the installation metadata, which is not necessarily the code that's currently imported.

6. Set the value to `__version__` in `sample/__init__.py` and import `sample` in `setup.py`.

```
import sample
setup(
    ...
    version=sample.__version__
    ...
)
```

Although this technique is common, beware that it will fail if `sample/__init__.py` imports packages from `install_requires` dependencies, which will very likely not be installed yet when `setup.py` is run.

7. Keep the version number in the tags of a version control system (Git, Mercurial, etc) instead of in the code, and automatically extract it from there using setuptools_scm.

## 2.7 Supporting multiple Python versions

**Page Status** Incomplete

**Last Reviewed** 2014-12-24

**Contents**



```
FIXME

Useful projects/resources to reference:

- DONE six
- DONE python-future (http://python-future.org)
- tox
- DONE Travis and Shining Panda CI (Shining Panda no longer available)
- DONE Appveyor
- DONE Ned Batchelder's "What's in Which Python"
  - http://nedbatchelder.com/blog/201310/whats_in_which_python_3.html
    - http://nedbatchelder.com/blog/201109/whats_in_which_python.html
- Lennart Regebro's "Porting to Python 3"
- Greg Hewgill's script to identify the minimum version of Python
  required to run a particular script:
  https://github.com/ghewgill/pyqver
- the Python 3 porting how to in the main docs
- cross reference to the stable ABI discussion
  in the binary extensions topic (once that exists)
- mention version classifiers for distribution metadata
```

In addition to the work required to create a Python package, it is often necessary that the package must be made available on different versions of Python. Different Python versions may contain different (or renamed) standard library packages, and the changes between Python versions 2.x and 3.x include changes in the language syntax.

Performed manually, all the testing required to ensure that the package works correctly on all the target Python versions (and OSs!) could be very time-consuming. Fortunately, several tools are available for dealing with this, and these will briefly be discussed here.

### 2.7.1 Automated Testing and Continuous Integration

Several hosted services for automated testing are available. These services will typically monitor your source code repository (e.g. at Github or Bitbucket) and run your project's test suite every time a new commit is made.

These services also offer facilities to run your project's test suite on *multiple versions of Python*, giving rapid feedback about whether the code will work, without the developer having to perform such tests themselves.

Wikipedia has an extensive comparison of many continuous-integration systems. There are two hosted services which when used in conjunction provide automated testing across Linux, Mac and Windows:

- Travis CI provides both a Linux and a macOS environment. The Linux environment is Ubuntu 12.04 LTS Server Edition 64 bit while the macOS is 10.9.2 at the time of writing.
- Appveyor provides a Windows environment (Windows Server 2012).

```
TODO Either link to or provide example .yml files for these two
services.

TODO How do we keep the Travis Linux and macOS versions up-to-date in this
document?
```

Both Travis CI and Appveyor require a YAML-formatted file as specification for the instructions for testing. If any tests fail, the output log for that specific configuration can be inspected.

For Python projects that are intended to be deployed on both Python 2 and 3 with a single-source strategy, there are a number of options.

### 2.7.2 Tools for single-source Python packages

six is a tool developed by Benjamin Peterson for wrapping over the differences between Python 2 and Python 3. The six package has enjoyed widespread use and may be regarded as a reliable way to write a single-source Python module that can be use in both Python 2 and 3. The six module can be used from as early as Python 2.5. A tool called modernize, developed by Armin Ronacher, can be used to automatically apply the code modifications provided by six.

Similar to six, python-future is a package that provides a compatibility layer between Python 2 and Python 3 source code; however, unlike six, this package aims to provide interoperability between Python 2 and Python 3 with a language syntax that matches one of the two Python versions: one may use

- a Python 2 (by syntax) module in a Python 3 project.
- a Python 3 (by syntax) module in a *Python 2* project.

Because of the bi-directionality, python-future offers a pathway to converting a Python 2 package to Python 3 syntax module-by-module. However, in contrast to six, python-future is supported only from Python 2.6. Similar to modernize for six, python-future comes with two scripts called `futurize` and `pasteurize` that can be applied to either a Python 2 module or a Python 3 module respectively.

Use of six or python-future adds an additional runtime dependency to your package: with python-future, the `futurize` script can be called with the `--stage1` option to apply only the changes that Python 2.6+ already provides for forward-compatibility to Python 3. Any remaining compatibility problems would require manual changes.

### 2.7.3 What's in Which Python?

Ned Batchelder provides a list of changes in each Python release for Python 2 and separately for Python 3. These lists may be used to check whether any changes between Python versions may affect your package.

```
TODO These lists should be reproduced here (with permission).

TODO The py3 list should be updated to include 3.4
```

## 2.8 Packaging binary extensions

**Page Status** Incomplete

**Last Reviewed** 2013-12-08

One of the features of the CPython reference interpreter is that, in addition to allowing the execution of Python code, it also exposes a rich C API for use by other software. One of the most common uses of this C API is to create importable C extensions that allow things which aren't always easy to achieve in pure Python code.

**Contents**

### 2.8.1 An overview of binary extensions

#### Use cases

The typical use cases for binary extensions break down into just three conventional categories:

- accelerator modules: these modules are completely self-contained, and are created solely to run faster than the equivalent pure Python code runs in CPython. Ideally, accelerator modules will always have a pure Python equivalent to use as a fallback if the accelerated version isn't available on a given system. The CPython standard library makes extensive use of accelerator modules.
- wrapper modules: these modules are created to expose existing C interfaces to Python code. They may either expose the underlying C interface directly, or else expose a more "Pythonic" API that makes use of Python language features to make the API easier to use. The CPython standard library makes extensive use of wrapper modules.
- low level system access: these modules are created to access lower level features of the CPython runtime, the operating system, or the underlying hardware. Through platform specific code, extension modules may achieve things that aren't possible in pure Python code. A number of CPython standard library modules are written in C in order to access interpreter internals that aren't exposed at the language level.

  One particularly notable feature of C extensions is that, when they don't need to call back into the interpreter runtime, they can release CPython's global interpreter lock around long-running operations (regardless of whether those operations are CPU or IO bound).

Not all extension modules will fit neatly into the above categories. The extension modules included with NumPy, for example, span all three use cases - they move inner loops to C for speed reasons, wrap external libraries written in C, FORTRAN and other languages, and use low level system interfaces for both CPython and the underlying operation system to support concurrent execution of vectorised operations and to tightly control the exact memory layout of created objects.

#### Disadvantages

The main disadvantage of using binary extensions is the fact that it makes subsequent distribution of the software more difficult. One of the advantages of using Python is that it is largely cross platform, and the languages used to write extension modules (typically C or C++, but really any language that can bind to the CPython C API) typically require that custom binaries be created for different platforms.

This means that binary extensions:

- require that end users be able to either build them from source, or else that someone publish pre-built binaries for common platforms
- may not be compatible with different builds of the CPython reference interpreter
- often will not work correctly with alternative interpreters such as PyPy, IronPython or Jython
- if handcoded, make maintenance more difficult by requiring that maintainers be familiar not only with Python, but also with the language used to create the binary extension, as well as with the details of the CPython C API.
- if a pure Python fallback implementation is provided, make maintenance more difficult by requiring that changes be implemented in two places, and introducing additional complexity in the test suite to ensure both versions are always executed.

Another disadvantage of relying on binary extensions is that alternative import mechanisms (such as the ability to import modules directly from zipfiles) often won't work for extension modules (as the dynamic loading mechanisms on most platforms can only load libraries from disk).

### Alternatives to handcoded accelerator modules

When extension modules are just being used to make code run faster (after profiling has identified the code where the speed increase is worth additional maintenance effort), a number of other alternatives should also be considered:

- look for existing optimised alternatives. The CPython standard libary includes a number of optimised data structures and algorithms (especially in the builtins and the `collections` and `itertools` modules). The Python Package Index also offers additional alternatives. Sometimes, the appropriate choice of standard library or third party module can avoid the need to create your own accelerator module.
- for long running applications, the JIT compiled PyPy interpreter may offer a suitable alternative to the standard CPython runtime. The main barrier to adopting PyPy is typically reliance on other binary extension modules - while PyPy does emulate the CPython C API, modules that rely on that cause problems for the PyPy JIT, and the emulation layer can often expose latent defects in extension modules that CPython currently tolerates (frequently around reference counting errors - an object having one live reference instead of two often won't break anything, but no references instead of one is a major problem).
- Cython is a mature static compiler that can compile most Python code to C extension modules. The initial compilation provides some speed increases (by bypassing the CPython interpreter layer), and Cython's optional static typing features can offer additional opportunities for speed increases. Using Cython still has the disadvantage of increasing the complexity of distributing the resulting application, but has the benefit of having a reduced barrier to entry for Python programmers (relative to other languages like C or C++).
- Numba is a newer tool, created by members of the scientific Python community, that aims to leverage LLVM to allow selective compilation of pieces of a Python application to native machine code at runtime. It requires that LLVM be available on the system where the code is running, but can provide significant speed increases, especially for operations that are amenable to vectorisation.

### Alternatives to handcoded wrapper modules

The C ABI (Application Binary Interface) is a common standard for sharing functionality between multiple applications. One of the strengths of the CPython C API (Application Programming Interface) is allowing Python users to tap into that functionality. However, wrapping modules by hand is quite tedious, so a number of other alternative approaches should be considered.

The approaches described below don't simplify the distribution case at all, but they *can* significantly reduce the maintenance burden of keeping wrapper modules up to date.

- In addition to being useful for the creation of accelerator modules, Cython is also useful for creating wrapper modules. It still involves wrapping the interfaces by hand, however, so may not be a good choice for wrapping large APIs.
- cffi is a project created by some of the PyPy developers to make it straightforward for developers that already know both Python and C to expose their C modules to Python applications. It also makes it relatively straightforward to wrap a C module based on its header files, even if you don't know C yourself.

  One of the key advantages of `cffi` is that it is compatible with the PyPy JIT, allowing CFFI wrapper modules to participate fully in PyPy's tracing JIT optimisations.
- SWIG is a wrapper interface generator that allows a variety of programming languages, including Python, to interface with C *and C++* code.
- The standard library's `ctypes` module, while useful for getting access to C level interfaces when header information isn't available, suffers from the fact that it operates solely at the C ABI level, and thus has no automatic consistency checking between the interface actually being exported by the library and the one declared in the Python code. By contrast, the above alternatives are all able to operate at the C *API* level, using C header files to ensure consistency between the interface exported by the library being wrapped and the one expected by the Python wrapper module. While `cffi` *can* operate directly at the C ABI level, it suffers from the same interface inconsistency problems as `ctypes` when it is used that way.

#### Alternatives for low level system access

For applications that need low level system access (regardless of the reason), a binary extension module often *is* the best way to go about it. This is particularly true for low level access to the CPython runtime itself, since some operations (like releasing the Global Interpreter Lock) are simply invalid when the interpreter is running code, even if a module like `ctypes` or `cffi` is used to obtain access to the relevant C API interfaces.

For cases where the extension module is manipulating the underlying operating system or hardware (rather than the CPython runtime), it may sometimes be better to just write an ordinary C library (or a library in another systems programming language like C++ or Rust that can export a C compatible ABI), and then use one of the wrapping techniques described above to make the interface available as an importable Python module.

### 2.8.2 Implementing binary extensions

The CPython Extending and Embedding guide includes an introduction to writing a custom extension module in C.

```
mention the stable ABI (3.2+, link to the CPython C API docs)
mention the module lifecycle
mention the challenges of shared static state and subinterpreters
mention the implications of the GIL for extension modules
mention the memory allocation APIs in 3.4+

mention again that all this is one of the reasons why you probably
*don't* want to handcode your extension modules :)
```

### 2.8.3 Building binary extensions

#### Setting up a build environment on Windows

Before it is possible to build a binary extension, it is necessary to ensure that you have a suitable compiler available. On Windows, Visual C is used to build the official CPython interpreter, and should be used to build compatible binary extensions.

Python 2.7 used Visual Studio 2008, Python 3.3 and 3.4 used Visual Studio 2010, and Python 3.5+ uses Visual Studio 2015. Unfortunately, older versions of Visual Studio are no longer easily available from Microsoft, so for versions of Python prior to 3.5, the compilers must be obtained differently if you do not already have a copy of the relevant version of Visual Studio.

To set up a build environment for binary extensions, the steps are as follows:

For Python 2.7

1. Install "Visual C++ Compiler Package for Python 2.7", which is available from Microsoft's website.
2. Use (a recent version of) setuptools in your setup.py (pip will do this for you, in any case).
3. Done.

For Python 3.4

1. Install "Windows SDK for Windows 7 and .NET Framework 4" (v7.1), which is available from Microsoft's website.
2. Work from an SDK command prompt (with the environment variables set, and the SDK on PATH).
3. Set DISTUTILS_USE_SDK=1
4. Done.

For Python 3.5

1. Install Visual Studio 2015 Community Edition (or any later version, when these are released).
2. Done.

Note that from Python 3.5 onwards, Visual Studio works in a backward compatible way, which means that any future version of Visual Studio will be able to build Python extensions for all Python versions from 3.5 onwards.

```
FIXME

cover Windows binary compatibility requirements
cover macOS binary compatibility requirements
cover the vagaries of Linux distros and other *nix systems
```

### 2.8.4 Publishing binary extensions

For interim guidance on this topic, see the discussion in this issue.

```
FIXME

cover publishing as wheel files on PyPI or a custom index server
cover creation of Windows and macOS installers
mention the fact that Linux distros have a requirement to build from
source in their own build systems, so binary-only releases are strongly
discouraged
```

## 2.9 Supporting Windows using Appveyor

**Page Status** Incomplete

**Last Reviewed** 2015-12-03

This section covers how to use the free Appveyor continuous integration service to provide Windows support for your project. This includes testing the code on Windows, and building Windows-targeted binaries for projects that use C extensions.

**Contents**



### 2.9.1 Background

Many projects are developed on Unix by default, and providing Windows support can be a challenge, because setting up a suitable Windows test environment is non-trivial, and may require buying software licenses.

The Appveyor service is a continuous integration service, much like the better-known Travis service that is commonly used for testing by projects hosted on Github. However, unlike Travis, the build workers on Appveyor are Windows hosts and have the necessary compilers installed to build Python extensions.

Windows users typically do not have access to a C compiler, and therefore are reliant on projects that use C extensions distributing binary wheels on PyPI in order for the distribution to be installable via `pip install <dist>`. By using Appveyor as a build service (even if not using it for testing) it is possible for projects without a dedicated Windows environment to provide Windows-targeted binaries.

### 2.9.2 Setting Up

In order to use Appveyor to build Windows wheels for your project, you must have an account on the service. Instructions on setting up an account are given in the Appveyor documentation. The free tier of account is perfectly adequate for open source projects.

Appveyor provides integration with Github and Bitbucket, so as long as your project is hosted on one of those two services, setting up Appveyor integration is straightforward.

Once you have set up your Appveyor account and added your project, Appveyor will automatically build your project each time a commit occurs. This behaviour will be familiar to users of Travis.

### 2.9.3 Adding Appveyor support to your project

In order to define how Appveyor should build your project, you need to add an `appveyor.yml` file to your project. The full details of what can be included in the file are covered in the Appveyor documentation. This guide will provide the details necessary to set up wheel builds.

Appveyor includes by default all of the compiler toolchains needed to build extensions for Python. For Python 2.7, 3.5+ and 32-bit versions of 3.3 and 3.4, the tools work out of the box. But for 64-bit versions of Python 3.3 and 3.4, there is a small amount of additional configuration needed to let distutils know where to find the 64-bit compilers. (From 3.5 onwards, the version of Visual Studio used includes 64-bit compilers with no additional setup).

#### appveyor.yml

```
environment:

  matrix:

    # For Python versions available on Appveyor, see
    # http://www.appveyor.com/docs/installed-software#python
    # The list here is complete (excluding Python 2.6, which
    # isn't covered by this document) at the time of writing.

    - PYTHON: "C:\\Python27"
    - PYTHON: "C:\\Python33"
    - PYTHON: "C:\\Python34"
    - PYTHON: "C:\\Python35"
    - PYTHON: "C:\\Python27-x64"
    - PYTHON: "C:\\Python33-x64"
      DISTUTILS_USE_SDK: "1"
    - PYTHON: "C:\\Python34-x64"
      DISTUTILS_USE_SDK: "1"
    - PYTHON: "C:\\Python35-x64"
    - PYTHON: "C:\\Python36-x64"

install:
  # We need wheel installed to build wheels
  - "%PYTHON%\\python.exe -m pip install wheel"

build: off

test_script:
  # Put your test command here.
  # If you don't need to build C extensions on 64-bit Python 3.3 or 3.4,
  # you can remove "build.cmd" from the front of the command, as it's
  # only needed to support those cases.
  # Note that you must use the environment variable %PYTHON% to refer to
  # the interpreter you're using - Appveyor does not do anything special
  # to put the Python version you want to use on PATH.
  - "build.cmd %PYTHON%\\python.exe setup.py test"

after_test:
  # This step builds your wheels.
  # Again, you only need build.cmd if you're building C extensions for
  # 64-bit Python 3.3/3.4. And you need to use %PYTHON% to get the correct
  # interpreter
  - "build.cmd %PYTHON%\\python.exe setup.py bdist_wheel"

```

```
artifacts:
  # bdist_wheel puts your built wheel in the dist directory
  - path: dist\*

#on_success:
#  You can use this step to upload your artifacts to a public website.
#  See Appveyor's documentation for more details. Or you can simply
#  access your wheels from the Appveyor "artifacts" tab for your build.
```

This file can be downloaded from here.

The `appveyor.yml` file must be located in the root directory of your project. It is in `YAML` format, and consists of a number of sections.

The `environment` section is the key to defining the Python versions for which your wheels will be created. Appveyor comes with Python 2.6, 2.7, 3.3, 3.4 and 3.5 installed, in both 32-bit and 64-bit builds. The example file builds for all of these environments except Python 2.6. Installing for Python 2.6 is more complex, as it does not come with pip included. We don't support 2.6 in this document (as Windows users still using Python 2 are generally able to move to Python 2.7 without too much difficulty).

The `install` section uses pip to install any additional software that the project may require. The only requirement for building wheels is the `wheel` project, but projects may wish to customise this code in certain circumstances (for example, to install additional build packages such as `Cython`, or test tools such as `tox`).

The `build` section simply switches off builds - there is no build step needed for Python, unlike languages like `C#`.

The main sections that will need to be tailored to your project are `test_script` and `after_test`.

The `test_script` section is where you will run your project's tests. The supplied file runs your test suite using `setup.py test`. If you are only interested in building wheels, and not in running your tests on Windows, you can replace this section with a dummy command such as `echo Skipped Tests`. You may wish to use another test tool, such as `nose` or `py.test`. Or you may wish to use a test driver like `tox` - however if you are using `tox` there are some additional configuration changes you will need to consider, which are described below.

The `after_test` runs once your tests have completed, and so is where the wheels should be built. Assuming your project uses the recommended tools (specifically, `setuptools`) then the `setup.py bdist_wheel` command will build your wheels.

Note that wheels will only be built if your tests succeed. If you expect your tests to fail on Windows, you can skip them as described above.

### Support script

The `appveyor.yml` file relies on a single support script, which sets up the environment to use the SDK compiler for 64-bit builds on Python 3.3 and 3.4. For projects which do not need a compiler, or which don't support 3.3 or 3.4 on 64-bit Windows, only the `appveyor.yml` file is needed.

build.cmd is a Windows batch script that runs a single command in an environment with the appropriate compiler for the selected Python version. All you need to do is to set the single environment variable `DISTUTILS_USE_SDK` to a value of `1` and the script does the rest. It sets up the SDK needed for 64-bit builds of Python 3.3 or 3.4, so don't set the environment variable for any other builds.

You can simply download the batch file and include it in your project unchanged.

### Access to the built wheels

When your build completes, the built wheels will be available from the Appveyor control panel for your project. They can be found by going to the build status page for each build in turn. At the top of the build output there is a series

of links, one of which is "Artifacts". That page will include a list of links to the wheels for that Python version / architecture. You can download those wheels and upload them to PyPI as part of your release process.

### 2.9.4 Additional Notes

#### Testing with tox

Many projects use the Tox tool to run their tests. It ensures that tests are run in an isolated environment using the exact files that will be distributed by the project.

In order to use `tox` on Appveyor there are a couple of additional considerations (in actual fact, these issues are not specific to Appveyor, and may well affect other CI systems).

1. By default, `tox` only passes a chosen subset of environment variables to the test processes. Because `distutils` uses environment variables to control the compiler, this "test isolation" feature will cause the tests to use the wrong compiler by default.

   To force `tox` to pass the necessary environment variables to the subprocess, you need to set the `tox` configuration option `passenv` to list the additional environment variables to be passed to the subprocess. For the SDK compilers, you need

   - `DISTUTILS_USE_SDK`
   - `MSSdk`
   - `INCLUDE`
   - `LIB`

   The `passenv` option can be set in your `tox.ini`, or if you prefer to avoid adding Windows-specific settings to your general project files, it can be set by setting the `TOX_TESTENV_PASSENV` environment variable. The supplied `build.cmd` script does this by default whenever `DISTUTILS_USE_SDK` is set.

2. When used interactively, `tox` allows you to run your tests against multiple environments (often, this means multiple Python versions). This feature is not as useful in a CI environment like Travis or Appveyor, where all tests are run in isolated environments for each configuration. As a result, projects often supply an argument `-e ENVNAME` to `tox` to specify which environment to use (there are default environments for most versions of Python).

   However, this does *not* work well with a Windows CI system like Appveyor, where there are (for example) two installations of Python 3.4 (32-bit and 64-bit) available, but only one `py34` environment in `tox`.

   In order to run tests using `tox`, therefore, projects should probably use the default `py` environment in `tox`, which uses the Python interpreter that was used to run `tox`. This will ensure that when Appveyor runs the tests, they will be run with the configured interpreter.

   In order to support running under the `py` environment, it is possible that projects with complex `tox` configurations might need to modify their `tox.ini` file. Doing so is, however, outside the scope of this document.

#### Automatically uploading wheels

It is possible to request Appveyor to automatically upload wheels. There is a `deployment` step available in `appveyor.yml` that can be used to (for example) copy the built artifacts to a FTP site, or an Amazon S3 instance. Documentation on how to do this is included in the Appveyor guides.

Alternatively, it would be possible to add a `twine upload` step to the build. The supplied `appveyor.yml` does not do this, as it is not clear that uploading new wheels after every commit is desirable (although some projects may wish to do this).

### External dependencies

The supplied scripts will successfully build any distribution that does not rely on 3rd party external libraries for the build.

It is possible to add steps to the `appveyor.yml` configuration (typically in the "install" section) to download and/or build external libraries needed by the distribution. And if needed, it is possible to add extra configuration for the build to supply the location of these libraries to the compiler. However, this level of configuration is beyond the scope of this document.

### Support scripts

For reference, the SDK setup support script is listed here:

`appveyor-sample/build.cmd`

```
@echo off
:: To build extensions for 64 bit Python 3, we need to configure environment
:: variables to use the MSVC 2010 C++ compilers from GRMSDKX_EN_DVD.iso of:
:: MS Windows SDK for Windows 7 and .NET Framework 4
::
:: More details at:
:: https://github.com/cython/cython/wiki/CythonExtensionsOnWindows

IF "%DISTUTILS_USE_SDK%"=="1" (
    ECHO Configuring environment to build with MSVC on a 64bit architecture
    ECHO Using Windows SDK 7.1
    "C:\Program Files\Microsoft SDKs\Windows\v7.1\Setup\WindowsSdkVer.exe" -q -
↪version:v7.1
    CALL "C:\Program Files\Microsoft SDKs\Windows\v7.1\Bin\SetEnv.cmd" /x64 /release
    SET MSSdk=1
    REM Need the following to allow tox to see the SDK compiler
    SET TOX_TESTENV_PASSENV=DISTUTILS_USE_SDK MSSdk INCLUDE LIB
) ELSE (
    ECHO Using default MSVC build environment
)

CALL %*
```

## 2.10 Packaging namespace packages

Namespace packages allow you to split the sub-packages and modules within a single *package* across multiple, separate *distribution packages* (referred to as **distributions** in this document to avoid ambiguity). For example, if you have the following package structure:

```
mynamespace/
    __init__.py
    subpackage_a/
        __init__.py
        ...
```

```
    subpackage_b/
        __init__.py
        ...
    module_b.py
setup.py
```

And you use this package in your code like so:

```
from mynamespace import subpackage_a
from mynamespace import subpackage_b
```

Then you can break these sub-packages into two separate distributions:

```
mynamespace-subpackage-a/
    setup.py
    mynamespace/
        subpackage_a/
            __init__.py

mynamespace-subpackage-b/
    setup.py
    mynamespace/
        subpackage_b/
            __init__.py
        module_b.py
```

Each sub-package can now be separately installed, used, and versioned.

Namespace packages can be useful for a large collection of loosely-related packages (such as a large corpus of client libraries for multiple products from a single company). However, namespace packages come with several caveats and are not appropriate in all cases. A simple alternative is to use a prefix on all of your distributions such as `import mynamespace_subpackage_a` (you could even use `import mynamespace_subpackage_a as subpackage_a` to keep the import object short).

### 2.10.1 Creating a namespace package

There are currently three different approaches to creating namespace packages:

1. Use *native namespace packages*. This type of namespace package is defined in **PEP 420** and is available in Python 3.3 and later. This is recommended if packages in your namespace only ever need to support Python 3 and installation via `pip`.
2. Use *pkgutil-style namespace packages*. This is recommended for new packages that need to support Python 2 and 3 and installation via both `pip` and `python setup.py install`.
3. Use *pkg_resources-style namespace packages*. This method is recommended if you need compatibility with packages already using this method or if your package needs to be zip-safe.

**Warning:** While native namespace packages and pkgutil-style namespace packages are largely compatible, pkg_resources-style namespace packages are not compatible with the other methods. It's inadvisable to use different methods in different distributions that provide packages to the same namespace.

### Native namespace packages

Python 3.3 added **implicit** namespace packages from **PEP 420**. All that is required to create a native namespace package is that you just omit `__init__.py` from the namespace package directory. An example file structure:

```
setup.py
mynamespace/
    # No __init__.py here.
    subpackage_a/
        # Sub-packages have __init__.py.
        __init__.py
        module.py
```

It is extremely important that every distribution that uses the namespace package omits the `__init__.py` or uses a pkgutil-style `__init__.py`. If any distribution does not, it will cause the namespace logic to fail and the other sub-packages will not be importable.

Because `mynamespace` doesn't contain an `__init__.py`, `setuptools.find_packages()` won't find the sub-package. You must explicitly list all packages in your `setup.py`. For example:

```
from setuptools import setup

setup(
    name='mynamespace-subpackage-a',
    ...
    packages=['mynamespace.subpackage_a']
)
```

A complete working example of two native namespace packages can be found in the native namespace package example project.

---

**Note:** Because native and pkgutil-style namespace packages are largely compatible, you can use native namespace packages in the distributions that only support Python 3 and pkgutil-style namespace packages in the distributions that need to support Python 2 and 3.

---

### pkgutil-style namespace packages

Python 2.3 introduced the pkgutil module and the extend_path function. This can be used to declare namespace packages that need to be compatible with both Python 2.3+ and Python 3. This is the recommended approach for the highest level of compatibility.

To create a pkgutil-style namespace package, you need to provide an `__init__.py` file for the namespace package:

```
setup.py
mynamespace/
    __init__.py  # Namespace package __init__.py
    subpackage_a/
        __init__.py  # Sub-package __init__.py
        module.py
```

The `__init__.py` file for the namespace package needs to contain **only** the following:

```
__path__ = __import__('pkgutil').extend_path(__path__, __name__)
```

**Every** distribution that uses the namespace package must include an identical `__init__.py`. If any distribution does not, it will cause the namespace logic to fail and the other sub-packages will not be importable. Any additional code in `__init__.py` will be inaccessible.

A complete working example of two pkgutil-style namespace packages can be found in the pkgutil namespace example project.

### pkg_resources-style namespace packages

Setuptools provides the pkg_resources.declare_namespace function and the `namespace_packages` argument to `setup()`. Together these can be used to declare namespace packages. While this approach is no longer recommended, it is widely present in most existing namespace packages. If you are creating a new distribution within an existing namespace package that uses this method then it's recommended to continue using this as the different methods are not cross-compatible and it's not advisable to try to migrate an existing package.

To create a pkg_resources-style namespace package, you need to provide an `__init__.py` file for the namespace package:

```
setup.py
mynamespace/
    __init__.py  # Namespace package __init__.py
    subpackage_a/
        __init__.py  # Sub-package __init__.py
        module.py
```

The `__init__.py` file for the namespace package needs to contain **only** the following:

```
__import__('pkg_resources').declare_namespace(__name__)
```

**Every** distribution that uses the namespace package must include an identical `__init__.py`. If any distribution does not, it will cause the namespace logic to fail and the other sub-packages will not be importable. Any additional code in `__init__.py` will be inaccessible.

---

**Note:** Some older recommendations advise the following in the namespace package `__init__.py`:

```
try:
    __import__('pkg_resources').declare_namespace(__name__)
except ImportError:
    __path__ = __import__('pkgutil').extend_path(__path__, __name__)
```

The idea behind this was that in the rare case that setuptools isn't available packages would fall-back to the pkgutil-style packages. This isn't advisable because pkgutil and pkg_resources-style namespace packages are not cross-compatible. If the presence of setuptools is a concern then the package should just explicitly depend on setuptools via `install_requires`.

---

Finally, every distribution must provide the `namespace_packages` argument to `setup()` in `setup.py`. For example:

```
from setuptools import find_packages, setup

setup(
    name='mynamespace-subpackage-a',
    ...
    packages=find_packages()
    namespace_packages=['mynamespace']
)
```

A complete working example of two pkg_resources-style namespace packages can be found in the pkg_resources namespace example project.

## 2.11 Creating and discovering plugins

Often when creating a Python application or library you'll want the ability to provide customizations or extra features via **plugins**. Because Python packages can be separately distributed, your application or library may want to automatically **discover** all of the plugins available.

There are three major approaches to doing automatic plugin discovery:

1. *Using naming convention*.
2. *Using namespace packages*.
3. *Using package metadata*.

### 2.11.1 Using naming convention

If all of the plugins for your application follow the same naming convention, you can use `pkgutil.iter_modules()` to discover all of the top-level modules that match the naming convention. For example, Flask uses the naming convention `flask_{plugin_name}`. If you wanted to automatically discover all of the Flask plugins installed:

```
import importlib
import pkgutil

flask_plugins = {
    name: importlib.import_module(name)
    for finder, name, ispkg
    in pkgutil.iter_modules()
    if name.startswith('flask_')
}
```

If you had both the Flask-SQLAlchemy and Flask-Talisman plugins installed then `flask_plugins` would be:

```
{
    'flask_sqlachemy': <module: 'flask_sqlalchemy'>,
    'flask_talisman': <module: 'flask_talisman'>,
}
```

Using naming convention for plugins also allows you to query the Python Package Index's simple API for all packages that conform to your naming convention.

### 2.11.2 Using namespace packages

*Namespace packages* can be used to provide a convention for where to place plugins and also provides a way to perform discovery. For example, if you make the sub-package `myapp.plugins` a namespace package then other *distributions* can provide modules and packages to that namespace. Once installed, you can use `pkgutil.iter_modules()` to discover all modules and packages installed under that namespace:

```
import importlib
import pkgutil

import myapp.plugins

def iter_namespace(ns_pkg):
    # Specifying the second argument (prefix) to iter_modules makes the
    # returned name an absolute name instead of a relative one. This allows
    # import_module to work without having to do additional modification to
    # the name.
    return pkgutil.iter_modules(ns_pkg.__path__, ns_pkg.__name__ + ".")

myapp_plugins = {
    name: importlib.import_module(name)
    for finder, name, ispkg
    in iter_namespace(myapp.plugins)
}
```

Specifying `myapp.plugins.__path__` to `iter_modules()` causes it to only look for the modules directly under that namespace. For example, if you have installed distributions that provide the modules `myapp.plugin.a` and `myapp.plugin.b` then `myapp_plugins` in this case would be:

```
{
    'a': <module: 'myapp.plugins.a'>,
    'b': <module: 'myapp.plugins.b'>,
}
```

This sample uses a sub-package as the namespace package (`myapp.plugin`), but it's also possible to use a top-level package for this purpose (such as `myapp_plugins`). How to pick the namespace to use is a matter of preference, but it's not recommended to make your project's main top-level package ( `myapp` in this case) a namespace package for the purpose of plugins, as one bad plugin could cause the entire namespace to break which would in turn make your project unimportable. For the "namespace sub-package" approach to work, the plugin packages must omit the `__init__.py` for your top-level package directory (`myapp` in this case) and include the namespace-package style `__init__.py` in the namespace sub-package directory (`myapp/plugins`). This also means that plugins will need to explicitly pass a list of packages to `setup()`'s `packages` argument instead of using `setuptools.find_packages()`.

**Warning:** Namespace packages are a complex feature and there are several different ways to create them. It's highly recommended to read the *Packaging namespace packages* documentation and clearly document which approach is preferred for plugins to your project.

### 2.11.3 Using package metadata

Setuptools provides special support for plugins. By providing the `entry_points` argument to `setup()` in `setup.py` plugins can register themselves for discovery.

For example if you have a package named `myapp-plugin-a` and it includes in its `setup.py`:

```
setup(
    ...
    entry_points={'myapp.plugins': 'a = myapp_plugin_a'},
    ...
)
```

Then you can discover and load all of the registered entry points by using `pkg_resources.iter_entry_points()`:

```
import pkg_resources

plugins = {
    entry_point.name: entry_point.load()
    for entry_point
    in pkg_resources.iter_entry_points('myapp.plugins')
}
```

In this example, `plugins` would be :

```
{
    'a': <module: 'myapp_plugin_a'>,
}
```

**Note:** The `entry_point` specification in `setup.py` is fairly flexible and has a lot of options. It's recommended to read over the entire section on entry points.

## 2.12 Package index mirrors and caches

**Page Status** Incomplete

**Last Reviewed** 2014-12-24

**Contents**



Mirroring or caching of PyPI can be used to speed up local package installation, allow offline work, handle corporate firewalls or just plain Internet flakiness.

Three options are available in this area:

1. pip provides local caching options,
2. devpi provides higher-level caching option, potentially shared amongst many users or machines, and
3. bandersnatch provides a local complete mirror of all PyPI *packages*.

### 2.12.1 Caching with pip

pip provides a number of facilities for speeding up installation by using local cached copies of *packages*:

1. Fast & local installs by downloading all the requirements for a project and then pointing pip at those downloaded files instead of going to PyPI.
2. A variation on the above which pre-builds the installation files for the requirements using pip wheel:

```
$ pip wheel --wheel-dir=/tmp/wheelhouse SomeProject
$ pip install --no-index --find-links=/tmp/wheelhouse SomeProject
```

### 2.12.2 Caching with devpi

devpi is a caching proxy server which you run on your laptop, or some other machine you know will always be available to you. See the devpi documentation for getting started.

### 2.12.3 Complete mirror with bandersnatch

bandersnatch will set up a complete local mirror of all PyPI *packages* (externally-hosted packages are not mirrored). See the bandersnatch documentation for getting that going.

A benefit of devpi is that it will create a mirror which includes *packages* that are external to PyPI, unlike bandersnatch which will only cache *packages* hosted on PyPI.

## 2.13 Hosting your own simple repository

If you wish to host your own simple repository[1], you can either use a software package like devpi or you can use simply create the proper directory structure and use any web server that can serve static files and generate an autoindex.

In either case, since you'll be hosting a repository that is likely not in your user's default repositories, you should instruct them in your project's description to configure their installer appropriately. For example with pip:

```
pip install --extra-index-url https://python.example.com/ foobar
```

In addition, it is **highly** recommended that you serve your repository with valid HTTPS. At this time, the security of your user's installations depends on all repositories using a valid HTTPS setup.

### 2.13.1 "Manual" Repository

The directory layout is fairly simple, within a root directory you need to create a directory for each project. This directory should be the normalized name (as defined by PEP 503) of the project. Within each of these directories simply place each of the downloadable files. If you have the projects "Foo" (with the versions 1.0 and 2.0) and "bar" (with the version 0.1) You should end up with a structure that looks like:

```
.
- bar
|   - bar-0.1.tar.gz
- foo
    - Foo-1.0.tar.gz
    - Foo-2.0.tar.gz
```

Once you have this layout, simply configure your webserver to serve the root directory with autoindex enabled. For an example using the built in Web server in Twisted, you would simply run `twistd -n web --path .` and then instruct users to add the URL to their installer's configuration.

[1] For complete documentation of the simple repository protocol, see PEP 503.

## 2.14 Migrating to PyPI.org

*PyPI.org* is a new, rewritten version of PyPI that is replacing the legacy code base located at *pypi.python.org*. As it becomes the default, and eventually only, version of PyPI people are expected to interact with, there will be a transition period where tooling and processes are expected to need to update to deal with the new location.

This section covers how to migrate to the new PyPI.org for different tasks.

### 2.14.1 Uploading

The recommended way to migrate to PyPI.org for uploading is to ensure that you are using a new enough version of your upload tool. Tools that support PyPI.org by default are twine v1.8.0+ (recommended tool), setuptools 27+, or the distutils included with Python 3.4.6+, Python 3.5.3+, Python 3.6+, and 2.7.13+.

In addition to ensuring you're on a new enough version of the tool for the tool's default to have switched, you must also make sure that you have not configured the tool to override its default upload URL. Typically this is configured in a file located at `$HOME/.pypirc`. If you see a file like:

```
[distutils]
index-servers =
    pypi

[pypi]
repository:https://pypi.python.org/pypi
username:yourusername
password:yourpassword
```

Then simply delete the line starting with `repository` and you will use your upload tool's default URL.

If for some reason you're unable to upgrade the version of your tool to a version that defaults to using PyPI.org, then you may edit `$HOME/.pypirc` and include the `repository:` line, but use the value `https://upload.pypi.org/legacy/` instead:

```
[distutils]
index-servers =
    pypi

[pypi]
repository: https://upload.pypi.org/legacy/
username: your username
password: your password
```

### 2.14.2 Registering package names & metadata

Explicit pre-registration of package names with the `setup.py register` command prior to the first upload is no longer required, and is not currently supported by the legacy upload API emulation on PyPI.org.

As a result, attempting explicit registration after switching to using PyPI.org for uploads will give the following error message:

```
Server response (410): This API is no longer supported, instead simply upload the␣
↪file.
```

The solution is to skip the registration step, and proceed directly to uploading artifacts.

### 2.14.3 Using TestPyPI

If you use TestPyPI, you must update your `$HOME/.pypirc` to handle TestPyPI's new location, by replacing `https://testpypi.python.org/pypi` with `https://test.pypi.org/legacy/`, for example:

```
[distutils]
index-servers=
    pypi
    testpypi

[testpypi]
repository: https://test.pypi.org/legacy/
username: your testpypi username
password: your testpypi password
```

## 2.15 Using TestPyPI

*TestPyPI* is a separate instance of the *Python Package Index (PyPI)* that allows you to try out the distribution tools and process without worrying about affecting the real index. TestPyPI is hosted at test.pypi.org

### 2.15.1 Registering your account

Because TestPyPI has a separate database from the live PyPI, you'll need a separate user account for specifically for TestPyPI. Go to https://test.pypi.org/account/register/ to register your account.

---

**Note:** The database for TestPyPI may be periodically pruned, so it is not unusual for user accounts to be deleted.

---

### 2.15.2 Using TestPyPI with Twine

You can upload your distributions to TestPyPI using *twine* by passing in the `--repository-url` flag

```
$ twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

### 2.15.3 Using TestPyPI with pip

You can tell pip to download packages from TestPyPI instead of PyPI by specifying the `--index-url` flag

```
$ pip install --index-url https://test.pypi.org/simple/ your-package
```

If you want to allow pip to also pull other packages from PyPI you can specify `--extra-index-url` to point to PyPI. This is useful when the package you're testing has dependencies:

```
pip install --index-url https://test.pypi.org/simple/ --extra-index-url https://pypi.
↪org/simple your-package
```

### 2.15.4 Setting up TestPyPI in pypirc

If you want to avoid entering the TestPyPI url and your username and password you can configure TestPyPI in your `$HOME/.pypirc`.

Create or modify `$HOME/.pypirc` with the following:

```
[distutils]
index-servers=
    pypi
    testpypi

[testpypi]
repository: https://test.pypi.org/legacy/
username: your testpypi username
password: your testpypi password
```

**Warning:** This stores your password in plaintext. It's recommended to set narrow permissions on this file.

You can then tell *twine* to upload to TestPyPI by specifying the `--repository` flag:

```
$ twine upload --repository testpypi dist/*
```

CHAPTER 3

# Discussions

**Discussions** are focused on providing comprehensive information about a specific topic. If you're just trying to get stuff done, see *Guides*.

## 3.1 Deploying Python applications

**Page Status** Incomplete

**Last Reviewed** 2014-11-11

**Contents**



### 3.1.1 Overview

#### Supporting multiple hardware platforms

```
FIXME
```

```
Meaning: x86, x64, ARM, others?

For Python-only distributions, it *should* be straightforward to deploy on all
platforms where Python can run.

For distributions with binary extensions, deployment is major headache.  Not only
must the extensions be built on all the combinations of operating system and
hardware platform, but they must also be tested, preferably on continuous
integration platforms.  The issues are similar to the "multiple Python
versions" section above, not sure whether this should be a separate section.
Even on Windows x64, both the 32 bit and 64 bit versions of Python enjoy
significant usage.
```

### 3.1.2 OS Packaging & Installers

```
FIXME

- Building rpm/debs for projects
- Building rpms/debs for whole virtualenvs
- Building macOS installers for Python projects
- Building Android APKs with Kivy+P4A or P4A & Buildozer
```

#### Windows

```
FIXME

- Building Windows installers for Python projects
```

#### Pynsist

Pynsist is a tool that bundles Python programs together with the Python-interpreter into a single installer based on NSIS. In most cases, packaging only requires the user to choose a version of the Python-interpreter and declare the dependencies of the program. The tool downloads the specified Python-interpreter for Windows and packages it with all the dependencies in a single Windows-executable installer.

The installer installs or updates the Python-interpreter on the users system, which can be used independently of the packaged program. The program itself, can be started from a shortcut, that the installer places in the start-menu. Uninstalling the program leaves the Python installation of the user intact.

A big advantage of pynsist is that the Windows packages can be built on Linux. There are several examples for different kinds of programs (console, GUI) in the documentation. The tool is released under the MIT-licence.

### 3.1.3 Application Bundles

```
FIXME

- py2exe/py2app/PEX
- wheels kinda/sorta
```

### 3.1.4 Configuration Management

```
FIXME

puppet
salt
chef
ansible
fabric
```

## 3.2 pip vs easy_install

*easy_install* was released in 2004, as part of *setuptools*. It was notable at the time for installing *packages* from *PyPI* using requirement specifiers, and automatically installing dependencies.

*pip* came later in 2008, as alternative to *easy_install*, although still largely built on top of *setuptools* components. It was notable at the time for *not* installing packages as *Eggs* or from *Eggs* (but rather simply as 'flat' packages from *sdists*), and introducing the idea of Requirements Files, which gave users the power to easily replicate environments.

Here's a breakdown of the important differences between pip and easy_install now:

| | **pip** | **easy_install** |
|---|---|---|
| Installs from *Wheels* | Yes | No |
| Uninstall Packages | Yes (`pip uninstall`) | No |
| Dependency Overrides | Yes (Requirements Files) | No |
| List Installed Packages | Yes (`pip list` and `pip freeze`) | No |
| **PEP 438** Support | Yes | No |
| Installation format | 'Flat' packages with *egg-info* metadata. | Encapsulated Egg format |
| sys.path modification | No | Yes |
| Installs from *Eggs* | No | Yes |
| pylauncher support | No | Yes[1] |
| *Multi-version Installs* | No | Yes |
| Exclude scripts during install | No | Yes |
| per project index | Only in virtualenv | Yes, via setup.cfg |

## 3.3 install_requires vs Requirements files

**Contents**

- *install_requires*
- *Requirements files*

[1] https://setuptools.readthedocs.io/en/latest/easy_install.html#natural-script-launcher

### 3.3.1 install_requires

`install_requires` is a *setuptools* `setup.py` keyword that should be used to specify what a project **minimally** needs to run correctly. When the project is installed by *pip*, this is the specification that is used to install its dependencies.

For example, if the project requires A and B, your `install_requires` would be like so:

```
install_requires=[
   'A',
   'B'
]
```

Additionally, it's best practice to indicate any known lower or upper bounds.

For example, it may be known, that your project requires at least v1 of 'A', and v2 of 'B', so it would be like so:

```
install_requires=[
   'A>=1',
   'B>=2'
]
```

It may also be known that project A follows semantic versioning, and that v2 of 'A' will indicate a break in compatibility, so it makes sense to not allow v2:

```
install_requires=[
   'A>=1,<2',
   'B>=2'
]
```

It is not considered best practice to use `install_requires` to pin dependencies to specific versions, or to specify sub-dependencies (i.e. dependencies of your dependencies). This is overly-restrictive, and prevents the user from gaining the benefit of dependency upgrades.

Lastly, it's important to understand that `install_requires` is a listing of "Abstract" requirements, i.e just names and version restrictions that don't determine where the dependencies will be fulfilled from (i.e. from what index or source). The where (i.e. how they are to be made "Concrete") is to be determined at install time using *pip* options.[1]

### 3.3.2 Requirements files

Requirements Files described most simply, are just a list of pip install arguments placed into a file.

Whereas `install_requires` defines the dependencies for a single project, Requirements Files are often used to define the requirements for a complete Python environment.

Whereas `install_requires` requirements are minimal, requirements files often contain an exhaustive listing of pinned versions for the purpose of achieving repeatable installations of a complete environment.

Whereas `install_requires` requirements are "Abstract", i.e. not associated with any particular index, requirements files often contain pip options like `--index-url` or `--find-links` to make requirements "Concrete", i.e. associated with a particular index or directory of packages.[1]

Whereas `install_requires` metadata is automatically analyzed by pip during an install, requirements files are not, and only are used when a user specifically installs them using `pip install -r`.

---

[1] For more on "Abstract" vs "Concrete" requirements, see https://caremad.io/2013/07/setup-vs-requirement/.

## 3.4 Wheel vs Egg

*Wheel* and *Egg* are both packaging formats that aim to support the use case of needing an install artifact that doesn't require building or compilation, which can be costly in testing and production workflows.

The *Egg* format was introduced by *setuptools* in 2004, whereas the *Wheel* format was introduced by **PEP 427** in 2012.

*Wheel* is currently considered the standard for *built* and *binary* packaging for Python.

Here's a breakdown of the important differences between *Wheel* and *Egg*.

- *Wheel* has an **official PEP**. *Egg* did not.
- *Wheel* is a *distribution* format, i.e a packaging format.[1] *Egg* was both a distribution format and a runtime installation format (if left zipped), and was designed to be importable.
- *Wheel* archives do not include .pyc files. Therefore, when the distribution only contains Python files (i.e. no compiled extensions), and is compatible with Python 2 and 3, it's possible for a wheel to be "universal", similar to an *sdist*.
- *Wheel* uses **PEP376-compliant** `.dist-info` directories. Egg used `.egg-info`.
- *Wheel* has a **richer file naming convention**. A single wheel archive can indicate its compatibility with a number of Python language versions and implementations, ABIs, and system architectures.
- *Wheel* is versioned. Every wheel file contains the version of the wheel specification and the implementation that packaged it.
- *Wheel* is internally organized by sysconfig path type, therefore making it easier to convert to other formats.

---

[1] Circumstantially, in some cases, wheels can be used as an importable runtime format, although **this is not officially supported at this time**.

CHAPTER 4

# PyPA Specifications

This is a list of currently active interoperability specifications maintained by the Python Packaging Authority.

## 4.1 Core Metadata Specifications

The current core metadata file format, version 1.2, is specified in **PEP 345**.

However, in a forthcoming PEP, the following specification will be defined as the canonical source for the core metadata file format. Fields which are defined in this specification, but do not currently appear in any accepted PEP, are marked as "New in version 1.3".

Fields defined in the following specification should be considered valid, complete and not subject to change. Fields should be considered "optional" for versions which predate their introduction.

**Contents**

- *Download-URL*
- *Author (optional)*
- *Author-email (optional)*
- *Maintainer (optional)*
- *Maintainer-email (optional)*
- *License (optional)*
- *Classifier (multiple use)*
- *Requires-Dist (multiple use)*
- *Provides-Dist (multiple use)*
- *Obsoletes-Dist (multiple use)*
- *Requires-Python*
- *Requires-External (multiple use)*
- *Project-URL (multiple-use)*
- *Provides-Extra (optional, multiple use)*

### 4.1.1 Metadata-Version

New in version 1.0.

Version of the file format; legal values are "1.0", "1.1" and "1.2".

Automated tools consuming metadata SHOULD warn if `metadata_version` is greater than the highest version they support, and MUST fail if `metadata_version` has a greater major version than the highest version they support (as described in **PEP 440**, the major version is the value before the first dot).

For broader compatibility, build tools MAY choose to produce distribution metadata using the lowest metadata version that includes all of the needed fields.

Example:

```
Metadata-Version: 1.2
```

### 4.1.2 Name

New in version 1.0.

The name of the distributions.

Example:

```
Name: BeagleVote
```

### 4.1.3 Version

New in version 1.0.

A string containing the distribution's version number. This field must be in the format specified in PEP 440.

Example:

```
Version: 1.0a2
```

### 4.1.4 Platform (multiple use)

New in version 1.0.

A Platform specification describing an operating system supported by the distribution which is not listed in the "Operating System" Trove classifiers. See "Classifier" below.

Examples:

```
Platform: ObscureUnix
Platform: RareDOS
```

### 4.1.5 Supported-Platform (multiple use)

New in version 1.1.

Binary distributions containing a PKG-INFO file will use the Supported-Platform field in their metadata to specify the OS and CPU for which the binary distribution was compiled. The semantics of the Supported-Platform field are not specified in this PEP.

Example:

```
Supported-Platform: RedHat 7.2
Supported-Platform: i386-win32-2791
```

### 4.1.6 Summary

New in version 1.0.

A one-line summary of what the distribution does.

Example:

```
Summary: A module for collecting votes from beagles.
```

### 4.1.7 Description (optional)

New in version 1.0.

A longer description of the distribution that can run to several paragraphs. Software that deals with metadata should not assume any maximum size for this field, though people shouldn't include their instruction manual as the description.

The contents of this field can be written using reStructuredText markup[1]. For programs that work with the metadata, supporting markup is optional; programs can also display the contents of the field as-is. This means that authors should be conservative in the markup they use.

To support empty lines and lines with indentation with respect to the RFC 822 format, any CRLF character has to be suffixed by 7 spaces followed by a pipe ("|") char. As a result, the Description field is encoded into a folded field that can be interpreted by RFC822 parser[2].

[1] reStructuredText markup: http://docutils.sourceforge.net/

[2] RFC 822 Long Header Fields: http://www.freesoft.org/CIE/RFC/822/7.htm

Example:

```
Description: This project provides powerful math functions
        |For example, you can use `sum()` to sum numbers:
        |
        |Example::
        |
        |    >>> sum(1, 2)
        |    3
        |
```

This encoding implies that any occurrences of a CRLF followed by 7 spaces and a pipe char have to be replaced by a single CRLF when the field is unfolded using a RFC822 reader.

### 4.1.8 Description-Content-Type (optional)

New in version 1.3.

A string stating the markup syntax (if any) used in the distribution's description, so that tools can intelligently render the description.

Historically, PyPI supported descriptions in plain text and reStructuredText (reST), and could render reST into HTML. However, it is common for distribution authors to write the description in Markdown (RFC 7763) as many code hosting sites render Markdown READMEs, and authors would reuse the file for the description. PyPI didn't recognize the format and so could not render the description correctly. This resulted in many packages on PyPI with poorly-rendered descriptions when Markdown is left as plain text, or worse, was attempted to be rendered as reST. This field allows the distribution author to specify the format of their description, opening up the possibility for PyPI and other tools to be able to render Markdown and other formats.

The format of this field is the same as the `Content-Type` header in HTTP (i.e.: RFC 1341). Briefly, this means that it has a `type/subtype` part and then it can optionally have a number of parameters:

Format:

```
Description-Content-Type: <type>/<subtype>; charset=<charset>[; <param_name>=<param␣
↪value> ...]
```

The `type/subtype` part has only a few legal values:

- `text/plain`
- `text/x-rst`
- `text/markdown`

The `charset` parameter can be used to specify the character encoding of the description. The only legal value is `UTF-8`. If omitted, it is assumed to be `UTF-8`.

Other parameters might be specific to the chosen subtype. For example, for the `markdown` subtype, there is an optional `variant` parameter that allows specifying the variant of Markdown in use (defaults to `CommonMark` if not specified). Currently, the only value that is recognized is:

- `CommonMark` for CommonMark

Example:

```
Description-Content-Type: text/plain; charset=UTF-8
```

Example:

```
Description-Content-Type: text/x-rst; charset=UTF-8
```

Example:

```
Description-Content-Type: text/markdown; charset=UTF-8; variant=CommonMark
```

Example:

```
Description-Content-Type: text/markdown
```

If a `Description-Content-Type` is not specified, then applications should attempt to render it as `text/x-rst; charset=UTF-8` and fall back to `text/plain` if it is not valid rst.

If a `Description-Content-Type` is an unrecognized value, then the assumed content type is `text/plain` (Although PyPI will probably reject anything with an unrecognized value).

If the `Description-Content-Type` is `text/markdown` and `variant` is not specified or is set to an unrecognized value, then the assumed `variant` is `CommonMark`.

So for the last example above, the `charset` defaults to `UTF-8` and the `variant` defaults to `CommonMark` and thus it is equivalent to the example before it.

### 4.1.9 Keywords (optional)

New in version 1.0.

A list of additional keywords to be used to assist searching for the distribution in a larger catalog.

Example:

```
Keywords: dog puppy voting election
```

### 4.1.10 Home-page (optional)

New in version 1.0.

A string containing the URL for the distribution's home page.

Example:

```
Home-page: http://www.example.com/~cschultz/bvote/
```

### 4.1.11 Download-URL

New in version 1.1.

A string containing the URL from which this version of the distribution can be downloaded. (This means that the URL can't be something like ". . . /BeagleVote-latest.tgz", but instead must be ". . . /BeagleVote-0.45.tgz".)

### 4.1.12 Author (optional)

New in version 1.0.

A string containing the author's name at a minimum; additional contact information may be provided.

Example:

```
Author: C. Schultz, Universal Features Syndicate,
        Los Angeles, CA <cschultz@peanuts.example.com>
```

### 4.1.13 Author-email (optional)

New in version 1.0.

A string containing the author's e-mail address. It can contain a name and e-mail address in the legal forms for a RFC-822 `From:` header.

Example:

```
Author-email: "C. Schultz" <cschultz@example.com>
```

### 4.1.14 Maintainer (optional)

New in version 1.2.

A string containing the maintainer's name at a minimum; additional contact information may be provided.

Note that this field is intended for use when a project is being maintained by someone other than the original author: it should be omitted if it is identical to `Author`.

Example:

```
Maintainer: C. Schultz, Universal Features Syndicate,
        Los Angeles, CA <cschultz@peanuts.example.com>
```

### 4.1.15 Maintainer-email (optional)

New in version 1.2.

A string containing the maintainer's e-mail address. It can contain a name and e-mail address in the legal forms for a RFC-822 `From:` header.

Note that this field is intended for use when a project is being maintained by someone other than the original author: it should be omitted if it is identical to `Author-email`.

Example:

```
Maintainer-email: "C. Schultz" <cschultz@example.com>
```

### 4.1.16 License (optional)

New in version 1.0.

Text indicating the license covering the distribution where the license is not a selection from the "License" Trove classifiers. See "Classifier" below. This field may also be used to specify a particular version of a licencse which is named via the `Classifier` field, or to indicate a variation or exception to such a license.

Examples:

```
License: This software may only be obtained by sending the
        author a postcard, and then the user promises not
        to redistribute it.

License: GPL version 3, excluding DRM provisions
```

### 4.1.17 Classifier (multiple use)

New in version 1.1.

Each entry is a string giving a single classification value for the distribution. Classifiers are described in PEP 301[3].

This field may be followed by an environment marker after a semicolon.

Examples:

```
Classifier: Development Status :: 4 - Beta
Classifier: Environment :: Console (Text Based)
```

### 4.1.18 Requires-Dist (multiple use)

New in version 1.2.

Each entry contains a string naming some other distutils project required by this distribution.

The format of a requirement string contains from one to four parts:

- A project name, in the same format as the `Name:` field. The only mandatory part.
- A comma-separated list of 'extra' names. These are defined by the required project, referring to specific features which may need extra dependencies.
- A version specifier. Tools parsing the format should accept optional parentheses around this, but tools generating it should not use parentheses.
- An environment marker after a semicolon. This means that the requirement is only needed in the specified conditions.

See **PEP 508** for full details of the allowed format.

The project names should correspond to names as found on the Python Package Index.

Version specifiers must follow the rules described in *Version Specifiers*.

Examples:

```
Requires-Dist: pkginfo
Requires-Dist: PasteDeploy
Requires-Dist: zope.interface (>3.5.0)
Requires-Dist: pywin32 >1.0; sys_platform == 'win32'
```

### 4.1.19 Provides-Dist (multiple use)

New in version 1.2.

[3] PEP 301, Package Index and Metadata for Distutils: http://www.python.org/dev/peps/pep-0301/

Each entry contains a string naming a Distutils project which is contained within this distribution. This field *must* include the project identified in the `Name` field, followed by the version : Name (Version).

A distribution may provide additional names, e.g. to indicate that multiple projects have been bundled together. For instance, source distributions of the `ZODB` project have historically included the `transaction` project, which is now available as a separate distribution. Installing such a source distribution satisfies requirements for both `ZODB` and `transaction`.

A distribution may also provide a "virtual" project name, which does not correspond to any separately-distributed project: such a name might be used to indicate an abstract capability which could be supplied by one of multiple projects. E.g., multiple projects might supply RDBMS bindings for use by a given ORM: each project might declare that it provides `ORM-bindings`, allowing other projects to depend only on having at most one of them installed.

A version declaration may be supplied and must follow the rules described in *Version Specifiers*. The distribution's version number will be implied if none is specified.

This field may be followed by an environment marker after a semicolon.

Examples:

```
Provides-Dist: OtherProject
Provides-Dist: AnotherProject (3.4)
Provides-Dist: virtual_package; python_version >= "3.4"
```

### 4.1.20 Obsoletes-Dist (multiple use)

New in version 1.2.

Each entry contains a string describing a distutils project's distribution which this distribution renders obsolete, meaning that the two projects should not be installed at the same time.

Version declarations can be supplied. Version numbers must be in the format specified in *Version Specifiers*.

This field may be followed by an environment marker after a semicolon.

The most common use of this field will be in case a project name changes, e.g. Gorgon 2.3 gets subsumed into Torqued Python 1.0. When you install Torqued Python, the Gorgon distribution should be removed.

Examples:

```
Obsoletes-Dist: Gorgon
Obsoletes-Dist: OtherProject (<3.0)
Obsoletes-Dist: Foo; os_name == "posix"
```

### 4.1.21 Requires-Python

New in version 1.2.

This field specifies the Python version(s) that the distribution is guaranteed to be compatible with. Installation tools may look at this when picking which version of a project to install.

The value must be in the format specified in *Version Specifiers*.

This field may be followed by an environment marker after a semicolon.

Examples:

```
Requires-Python: >=3
Requires-Python: >2.6,!=3.0.*,!=3.1.*
Requires-Python: ~=2.6
Requires-Python: >=3; sys_platform == 'win32'
```

### 4.1.22 Requires-External (multiple use)

New in version 1.2.

Each entry contains a string describing some dependency in the system that the distribution is to be used. This field is intended to serve as a hint to downstream project maintainers, and has no semantics which are meaningful to the `distutils` distribution.

The format of a requirement string is a name of an external dependency, optionally followed by a version declaration within parentheses.

This field may be followed by an environment marker after a semicolon.

Because they refer to non-Python software releases, version numbers for this field are **not** required to conform to the format specified in PEP 440: they should correspond to the version scheme used by the external dependency.

Notice that there's is no particular rule on the strings to be used.

Examples:

```
Requires-External: C
Requires-External: libpng (>=1.5)
Requires-External: make; sys_platform != "win32"
```

### 4.1.23 Project-URL (multiple-use)

New in version 1.2.

A string containing a browsable URL for the project and a label for it, separated by a comma.

Example:

```
Bug Tracker, http://bitbucket.org/tarek/distribute/issues/
```

The label is a free text limited to 32 signs.

### 4.1.24 Provides-Extra (optional, multiple use)

New in version 1.3.

A string containing the name of an optional feature. Must be a valid Python identifier. May be used to make a dependency conditional on whether the optional feature has been requested.

Example:

```
Provides-Extra: pdf
Requires-Dist: reportlab; extra == 'pdf'
```

A second distribution requires an optional dependency by placing it inside square brackets, and can request multiple features by separating them with a comma (,). The requirements are evaluated for each requested feature and added to the set of requirements for the distribution.

Example:

```
Requires-Dist: beaglevote[pdf]
Requires-Dist: libexample[test, doc]
```

Two feature names *test* and *doc* are reserved to mark dependencies that are needed for running automated tests and generating documentation, respectively.

It is legal to specify `Provides-Extra:` without referencing it in any `Requires-Dist:`.

---

## 4.2 Version Specifiers

Version numbering requirements and the semantics for specifying comparisons between versions are defined in **PEP 440**.

The version specifiers section in this PEP supersedes the version specifiers section in **PEP 345**.

## 4.3 Dependency Specifiers

The dependency specifier format used to declare a dependency on another component is defined in **PEP 508**.

The environment markers section in this PEP supersedes the environment markers section in **PEP 345**.

## 4.4 Declaring Build System Dependencies

*pyproject.toml* is a build system independent file format defined in **PEP 518** that projects may provide in order to declare any Python level dependencies that must be installed in order to run the project's build system successfully.

## 4.5 Distribution Formats

### 4.5.1 Source Distribution Format

The source distribution format (`sdist`) is not currently formally defined. Instead, its format is implicitly defined by the behaviour of the standard library's `distutils` module when executing the `setup.py sdist` command.

### 4.5.2 Binary Distribution Format

The binary distribution format (`wheel`) is defined in **PEP 427**.

## 4.6 Platform Compatibility Tags

The platform compatibility tagging model used for `wheel` distribution is defined in **PEP 425**.

The scheme defined in that PEP is insufficient for public distribution of Linux wheel files (and *nix wheel files in general), so **PEP 513** was created to define the `manylinux1` tag.

## 4.7 Recording Installed Distributions

The format used to record installed packages and their contents is defined in **PEP 376**.

Note that only the `dist-info` directory and the `RECORD` file format from that PEP are currently implemented in the default packaging toolchain.

## 4.8 Simple repository API

The current interface for querying available package versions and retrieving packages from an index server is defined in **PEP 503**.

## 4.9 Entry points specification

*Entry points* are a mechanism for an installed distribution to advertise components it provides to be discovered and used by other code. For example:

- Distributions can specify `console_scripts` entry points, each referring to a function. When *pip* (or another console_scripts aware installer) installs the distribution, it will create a command-line wrapper for each entry point.
- Applications can use entry points to load plugins; e.g. Pygments (a syntax highlighting tool) can use additional lexers and styles from separately installed packages. For more about this, see *Creating and discovering plugins*.

The entry point file format was originally developed to allow packages built with setuptools to provide integration point metadata that would be read at runtime with `pkg_resources`. It is now defined as a PyPA interoperability specification in order to allow build tools other than setuptools to publish `pkg_resources` compatible entry point metadata, and runtime libraries other than `pkg_resources` to portably read published entry point metadata (potentially with different caching and conflict resolution strategies).

### 4.9.1 Data model

Conceptually, an entry point is defined by three required properties:

- The **group** that an entry point belongs to indicates what sort of object it provides. For instance, the group `console_scripts` is for entry points referring to functions which can be used as a command, while `pygments.styles` is the group for classes defining pygments styles. The consumer typically defines the expected interface. To avoid clashes, consumers defining a new group should use names starting with a PyPI name owned by the consumer project, followed by `.`. Group names must be one or more groups of letters, numbers and underscores, separated by dots (regex `^\w+(\.\w+)*$`).
- The **name** identifies this entry point within its group. The precise meaning of this is up to the consumer. For console scripts, the name of the entry point is the command that will be used to launch it. Within a distribution, entry point names should be unique. If different distributions provide the same name, the consumer decides how to handle such conflicts. The name may contain any characters except =, but it cannot start or end with any whitespace character, or start with `[`. For new entry points, it is recommended to use only letters, numbers, underscores, dashes and dots (regex `[\w-.]+`).
- The **object reference** points to a Python object. It is either in the form `importable.module`, or `importable.module:object.attr`. Each of the parts delimited by dots and the colon is a valid Python identifier. It is intended to be looked up like this:

```
import importlib
modname, qualname_separator, qualname = object_ref.partition(':')
obj = importlib.import_module(modname)
if qualname_separator:
    for attr in qualname.split('.'):
        obj = getattr(obj, attr)
```

**Note:** Some tools call this kind of object reference by itself an 'entry point', for want of a better term, especially where it points to a function to launch a program.

There is also an optional property: the **extras** are a set of strings identifying optional features of the distribution providing the entry point. If these are specified, the entry point requires the dependencies of those 'extras'. See the metadata field *Provides-Extra (optional, multiple use)*.

Using extras for an entry point is no longer recommended. Consumers should support parsing them from existing distributions, but may then ignore them. New publishing tools need not support specifying extras. The functionality of handling extras was tied to setuptools' model of managing 'egg' packages, but newer tools such as pip and virtualenv use a different model.

### 4.9.2 File format

Entry points are defined in a file called `entry_points.txt` in the `*.dist-info` directory of the distribution. This is the directory described in **PEP 376** for installed distributions, and in **PEP 427** for wheels. The file uses the UTF-8 character encoding.

The file contents are in INI format, as read by Python's `configparser` module. However, configparser treats names as case-insensitive by default, whereas entry point names are case sensitive. A case-sensitive config parser can be made like this:

```
import configparser

class CaseSensitiveConfigParser(configparser.ConfigParser):
    optionxform = staticmethod(str)
```

The entry points file must always use = to delimit names from values (whereas configparser also allows using `:`).

The sections of the config file represent entry point groups, the names are names, and the values encode both the object reference and the optional extras. If extras are used, they are a comma-separated list inside square brackets.

Within a value, readers must accept and ignore spaces (including multiple consecutive spaces) before or after the colon, between the object reference and the left square bracket, between the extra names and the square brackets and colons delimiting them, and after the right square bracket. The syntax for extras is formally specified as part of **PEP 508** (as `extras`). For tools writing the file, it is recommended only to insert a space between the object reference and the left square bracket.

For example:

```
[console_scripts]
foo = foomod:main
# One which depends on extras:
foobar = foomod:main_bar [bar,baz]

# pytest plugins refer to a module, so there is no ':obj'
[pytest11]
nbval = nbval.plugin
```

### 4.9.3 Use for scripts

Two groups of entry points have special significance in packaging: `console_scripts` and `gui_scripts`. In both groups, the name of the entry point should be usable as a command in a system shell after the package is installed. The object reference points to a function which will be called with no arguments when this command is run. The function may return an integer to be used as a process exit code, and returning `None` is equivalent to returning `0`.

For instance, the entry point `mycmd = mymod:main` would create a command `mycmd` launching a script like this:

```
import sys
from mymod import main
sys.exit(main())
```

The difference between `console_scripts` and `gui_scripts` only affects Windows systems. `console_scripts` are wrapped in a console executable, so they are attached to a console and can use `sys.stdin`, `sys.stdout` and `sys.stderr` for input and output. `gui_scripts` are wrapped in a GUI executable, so they can be started without a console, but cannot use standard streams unless application code redirects them. Other platforms do not have the same distinction.

Install tools are expected to set up wrappers for both `console_scripts` and `gui_scripts` in the scripts directory of the install scheme. They are not responsible for putting this directory in the `PATH` environment variable which defines where command-line tools are found.

As files are created from the names, and some filesystems are case-insensitive, packages should avoid using names in these groups which differ only in case. The behaviour of install tools when names differ only in case is undefined.

CHAPTER 5

# Project Summaries

Summaries and links for the most relevant projects in the space of Python installation and packaging.

## 5.1 PyPA Projects

### 5.1.1 bandersnatch

Mailing list[2] | Issues | Bitbucket | PyPI

bandersnatch is a PyPI mirroring client designed to efficiently create a complete mirror of the contents of PyPI.

### 5.1.2 distlib

Docs | Mailing list[2] | Issues | Bitbucket | PyPI

Distlib is a library which implements low-level functions that relate to packaging and distribution of Python software. It consists in part of the functions from the distutils2 project, which was intended to be released as `packaging` in the Python 3.3 stdlib, but was removed shortly before Python 3.3 entered beta testing.

### 5.1.3 packaging

Dev list | Issues | Github | PyPI | User irc:#pypa | Dev irc:#pypa-dev

Core utilities for Python packaging used by *pip* and *setuptools*.

[2] Multiple projects reuse the distutils-sig mailing list as their user list.

### 5.1.4 pip

Docs | User list[1] | Dev list | Issues | Github | PyPI | User irc:#pypa | Dev irc:#pypa-dev

A tool for installing Python packages.

### 5.1.5 Pipfile

Source

`Pipfile` and its sister `Pipfile.lock` are a higher-level application-centric alternative to *pip*'s lower-level `requirements.txt` file.

### 5.1.6 Python Packaging User Guide

Docs | Mailing list | Issues | Github | User irc:#pypa | Dev irc:#pypa-dev

This guide!

### 5.1.7 setuptools

Docs | User list[2] | Dev list | Issues | GitHub | PyPI | User irc:#pypa | Dev irc:#pypa-dev

setuptools (which includes `easy_install`) is a collection of enhancements to the Python distutils that allow you to more easily build and distribute Python distributions, especially ones that have dependencies on other packages.

distribute was a fork of setuptools that was merged back into setuptools (in v0.7), thereby making setuptools the primary choice for Python packaging.

### 5.1.8 twine

Mailing list[2] | Issues | Github | PyPI

Twine is a utility for interacting with PyPI, that offers a secure replacement for `setup.py upload`.

### 5.1.9 virtualenv

Docs | User list | Dev list | Issues | Github | PyPI | User irc:#pypa | Dev irc:#pypa-dev

A tool for creating isolated Python environments.

### 5.1.10 Warehouse

Docs | Mailing list[2] | Issues | Github | Dev irc:#pypa-dev

The current codebase powering the *Python Package Index (PyPI)*. It is hosted at pypi.org.

[1] pip was created by the same developer as virtualenv, and early on adopted the virtualenv mailing list, and it's stuck ever since.

### 5.1.11 wheel

Docs | Mailing list[2] | Issues | Bitbucket | PyPI | User irc:#pypa | Dev irc:#pypa-dev

Primarily, the wheel project offers the `bdist_wheel` *setuptools* extension for creating *wheel distributions*. Additionally, it offers its own command line utility for creating and installing wheels.

## 5.2 Non-PyPA Projects

### 5.2.1 bento

Docs | Mailing list | Issues | Github | PyPI

Bento is a packaging tool solution for Python software, targeted as an alternative to distutils, setuptools, distribute, etc.... Bento's philosophy is reproducibility, extensibility and simplicity (in that order).

### 5.2.2 buildout

Docs | Mailing list[2] | Issues | PyPI | irc:#buildout

Buildout is a Python-based build system for creating, assembling and deploying applications from multiple parts, some of which may be non-Python-based. It lets you create a buildout configuration and reproduce the same software later.

### 5.2.3 conda

Docs

conda is the package management tool for Anaconda Python installations. Anaconda Python is a distribution from Continuum Analytics specifically aimed at the scientific community, and in particular on Windows where the installation of binary extensions is often difficult.

Conda is a completely separate tool to pip, virtualenv and wheel, but provides many of their combined features in terms of package management, virtual environment management and deployment of binary extensions.

Conda does not install packages from PyPI and can install only from the official Continuum repositories, or anaconda.org (a place for user-contributed *conda* packages), or a local (e.g. intranet) package server. However, note that pip can be installed into, and work side-by-side with conda for managing distributions from PyPI.

### 5.2.4 devpi

Docs | Mailing List | Issues | PyPI

devpi features a powerful PyPI-compatible server and PyPI proxy cache with a complimentary command line tool to drive packaging, testing and release activities with Python.

### 5.2.5 flit

Docs | Issues | PyPI

Flit is a simple way to put Python packages and modules on PyPI. Flit packages a single importable module or package at a time, using the import name as the name on PyPI. All subpackages and data files within a package are included

automatically. Flit requires Python 3, but you can use it to distribute modules for Python 2, so long as they can be imported on Python 3.

### 5.2.6 enscons

Source | Issues | PyPI

Enscons is a Python packaging tool based on SCons. It builds pip-compatible source distributions and wheels without using distutils or setuptools, including distributions with C extensions. Enscons has a different architecture and philosophy than distutils. Rather than adding build features to a Python packaging system, enscons adds Python packaging to a general purpose build system. Enscons helps you to build sdists that can be automatically built by pip, and wheels that are independent of enscons.

### 5.2.7 Hashdist

Docs | Github

Hashdist is a library for building non-root software distributions. Hashdist is trying to be "the Debian of choice for cases where Debian technology doesn't work". The best way for Pythonistas to think about Hashdist may be a more powerful hybrid of virtualenv and buildout.

### 5.2.8 pex

Docs | Github | PyPI

pex is both a library and tool for generating `.pex` (Python EXecutable) files, standalone Python environments in the spirit of *virtualenv*. `.pex` files are just carefully constructed zip files with a `#!/usr/bin/env python` and special `__main__.py`, and are designed to make deployment of Python applications as simple as `cp`.

### 5.2.9 Pipenv

Docs | Source | Issues | PyPI

Pipenv is a project that aims to bring the best of all packaging worlds to the Python world. It harnesses *Pipfile*, *pip*, and *virtualenv* into one single toolchain. It features very pretty terminal colors.

### 5.2.10 Spack

Docs | Github | Paper | Slides

A flexible package manager designed to support multiple versions, configurations, platforms, and compilers. Spack is like homebrew, but packages are written in Python and parameterized to allow easy swapping of compilers, library versions, build options, etc. Arbitrarily many versions of packages can coexist on the same system. Spack was designed for rapidly building high performance scientific applications on clusters and supercomputers.

Spack is not in PyPI (yet), but it requires no installation and can be used immediately after cloning from github.

## 5.3 Standard Library Projects

### 5.3.1 ensurepip

Docs | Issues

A package in the Python Standard Library that provides support for bootstrapping *pip* into an existing Python installation or virtual environment. In most cases, end users won't use this module, but rather it will be used during the build of the Python distribution.

### 5.3.2 distutils

Docs | User list[2] | Issues | User irc:#pypa | Dev irc:#pypa-dev

A package in the Python Standard Library that has support for creating and installing *distributions*. *setuptools* provides enhancements to distutils, and is much more commonly used than just using distutils by itself.

### 5.3.3 venv

Docs | Issues

A package in the Python Standard Library (starting with Python 3.3) for creating *Virtual Environments*. For more information, see the section on *Creating Virtual Environments*.

CHAPTER 6

# Glossary

**Binary Distribution** A specific kind of *Built Distribution* that contains compiled extensions.

**Built Distribution** A *Distribution* format containing files and metadata that only need to be moved to the correct location on the target system, to be installed. *Wheel* is such a format, whereas distutil's *Source Distribution* is not, in that it requires a build step before it can be installed. This format does not imply that Python files have to be precompiled (*Wheel* intentionally does not include compiled Python files).

**Distribution Package** A versioned archive file that contains Python *packages*, *modules*, and other resource files that are used to distribute a *Release*. The archive file is what an end-user will download from the internet and install.

A distribution package is more commonly referred to with the single words "package" or "distribution", but this guide may use the expanded term when more clarity is needed to prevent confusion with an *Import Package* (which is also commonly called a "package") or another kind of distribution (e.g. a Linux distribution or the Python language distribution), which are often referred to with the single term "distribution".

**Egg** A *Built Distribution* format introduced by *setuptools*, which is being replaced by *Wheel*. For details, see The Internal Structure of Python Eggs and Python Eggs

**Extension Module** A *module* written in the low-level language of the Python implementation: C/C++ for Python, Java for Jython. Typically contained in a single dynamically loadable pre-compiled file, e.g. a shared object (.so) file for Python extensions on Unix, a DLL (given the .pyd extension) for Python extensions on Windows, or a Java class file for Jython extensions.

**Known Good Set (KGS)** A set of distributions at specified versions which are compatible with each other. Typically a test suite will be run which passes all tests before a specific set of packages is declared a known good set. This term is commonly used by frameworks and toolkits which are comprised of multiple individual distributions.

**Import Package** A Python module which can contain other modules or recursively, other packages.

An import package is more commonly referred to with the single word "package", but this guide will use the expanded term when more clarity is needed to prevent confusion with a *Distribution Package* which is also commonly called a "package".

**Module** The basic unit of code reusability in Python, existing in one of two types: *Pure Module*, or *Extension Module*.

**Package Index** A repository of distributions with a web interface to automate *package* discovery and consumption.

**Per Project Index** A private or other non-canonical *Package Index* indicated by a specific *Project* as the index preferred or required to resolve dependencies of that project.

**Project** A library, framework, script, plugin, application, or collection of data or other resources, or some combination thereof that is intended to be packaged into a *Distribution*.

Since most projects create *Distributions* using *distutils* or *setuptools*, another practical way to define projects currently is something that contains a *setup.py* at the root of the project src directory, where "setup.py" is the project specification filename used by *distutils* and *setuptools*.

Python projects must have unique names, which are registered on *PyPI*. Each project will then contain one or more *Releases*, and each release may comprise one or more *distributions*.

Note that there is a strong convention to name a project after the name of the package that is imported to run that project. However, this doesn't have to hold true. It's possible to install a distribution from the project 'foo' and have it provide a package importable only as 'bar'.

**Pure Module** A *module* written in Python and contained in a single .py file (and possibly associated .pyc and/or .pyo files).

**Python Packaging Authority (PyPA)** PyPA is a working group that maintains many of the relevant projects in Python packaging. They maintain a site at https://www.pypa.io, host projects on github and bitbucket, and discuss issues on the pypa-dev mailing list.

**Python Package Index (PyPI)** PyPI is the default *Package Index* for the Python community. It is open to all Python developers to consume and distribute their distributions.

**pypi.org** pypi.org is the domain name for the *Python Package Index (PyPI)*. It replaced the legacy index domain name, *pypi.python.org*, in 2017. It is powered by *Warehouse*.

**Release** A snapshot of a *Project* at a particular point in time, denoted by a version identifier.

Making a release may entail the publishing of multiple *Distributions*. For example, if version 1.0 of a project was released, it could be available in both a source distribution format and a Windows installer distribution format.

**Requirement** A specification for a *package* to be installed. *pip*, the *PYPA* recommended installer, allows various forms of specification that can all be considered a "requirement". For more information, see the pip install reference.

**Requirement Specifier** A format used by *pip* to install packages from a *Package Index*. For an EBNF diagram of the format, see the pkg_resources.Requirement entry in the *setuptools* docs. For example, "foo>=1.3" is a requirement specifier, where "foo" is the project name, and the ">=1.3" portion is the *Version Specifier*

**Requirements File** A file containing a list of *Requirements* that can be installed using *pip*. For more information, see the *pip* docs on Requirements Files.

**setup.py** The project specification file for *distutils* and *setuptools*.

**Source Archive** An archive containing the raw source code for a *Release*, prior to creation of an *Source Distribution* or *Built Distribution*.

**Source Distribution (or "sdist")** A *distribution* format (usually generated using `python setup.py sdist`) that provides metadata and the essential source files needed for installing by a tool like *pip*, or for generating a *Built Distribution*.

**System Package** A package provided in a format native to the operating system, e.g. an rpm or dpkg file.

**Version Specifier** The version component of a *Requirement Specifier*. For example, the ">=1.3" portion of "foo>=1.3". **PEP 440** contains a **full specification** of the specifiers that Python packaging currently supports. Support for PEP440 was implemented in *setuptools* v8.0 and *pip* v6.0.

**Virtual Environment** An isolated Python environment that allows packages to be installed for use by a particular application, rather than being installed system wide. For more information, see the section on *Creating Virtual Environments*.

**Wheel** A *Built Distribution* format introduced by **PEP 427**, which is intended to replace the *Egg* format. Wheel is currently supported by *pip*.

**Working Set** A collection of *distributions* available for importing. These are the distributions that are on the *sys.path* variable. At most, one *Distribution* for a project is possible in a working set.

CHAPTER 7

# How to Get Support

For support related to a specific project, see the links on the *Projects* page.

For something more general, or when you're just not sure, use the distutils-sig list.

CHAPTER 8

# Contribute to this guide

The Python Packaging User Guide welcomes contributors! There are lots of ways to help out, including:

- Reading the guide and giving feedback
- Reviewing new contributions
- Revising existing content
- Writing new content

Most of the work on the Python Packaging User Guide takes place on the project's GitHub repository. To get started, check out the list of open issues and pull requests. If you're planning to write or edit the guide, please read the *style guide*.

By contributing to the Python Packaging User Guide, you're expected to follow the Python Packaging Authority's Contributor Code of Conduct. Harassment, personal attacks, and other unprofessional conduct is not acceptable.

## 8.1 Style guide

This style guide has recommendations for how you should write the Python Packaging User Guide. Before you start writing, please review it. By following the style guide, your contributions will help add to a cohesive whole and make it easier for your contributions to be accepted into the project.

### 8.1.1 Purpose

The purpose of the Python Packaging User Guide is

> to be the authoritative resource on how to package, publish, and install Python projects using current tools.

### 8.1.2 Scope

The guide is meant to answer questions and solve problems with accurate and focused recommendations.

The guide isn't meant to be comprehensive and it's not meant to replace individual projects' documentation. For example, pip has dozens of commands, options, and settings. The pip documentation describes each of them in detail, while this guide describes only the parts of pip that are needed to complete the specific tasks described in this guide.

### 8.1.3 Audience

The audience of this guide is anyone who uses Python with packages.

Don't forget that the Python community is big and welcoming. Readers may not share your age, gender, education, culture, and more, but they deserve to learn about packaging just as much as you do.

In particular, keep in mind that not all people who use Python see themselves as programmers. The audience of this guide includes astronomers or painters or students as well as professional software developers.

### 8.1.4 Voice and tone

When writing this guide, strive to write with a voice that's approachable and humble, even if you have all the answers.

Imagine you're working on a Python project with someone you know to be smart and skilled. You like working with them and they like working with you. That person has asked you a question and you know the answer. How do you respond? *That* is how you should write this guide.

Here's a quick check: try reading aloud to get a sense for your writing's voice and tone. Does it sound like something you would say or does it sound like you're acting out a part or giving a speech? Feel free to use contractions and don't worry about sticking to fussy grammar rules. You are hereby granted permission to end a sentence in a preposition, if that's what you want to end it with.

When writing the guide, adjust your tone for the seriousness and difficulty of the topic. If you're writing an introductory tutorial, it's OK to make a joke, but if you're covering a sensitive security recommendation, you might want to avoid jokes altogether.

### 8.1.5 Conventions and mechanics

**Write to the reader** When giving recommendations or steps to take, address the reader as *you* or use the imperative mood.

> Wrong: To install it, the user runs…
> Right: You can install it by running…
> Right: To install it, run…

**State assumptions** Avoid making unstated assumptions. Reading on the web means that any page of the guide may be the first page of the guide that the reader ever sees. If you're going to make assumptions, then say what assumptions that you're going to make.

**Cross-reference generously** The first time you mention a tool or practice, link to the part of the guide that covers it, or link to a relevant document elsewhere. Save the reader a search.

**Respect naming practices** When naming tools, sites, people, and other proper nouns, use their preferred capitalization.

> Wrong: Pip uses…

Right: pip uses...

Wrong: ...hosted on github.
Right: ...hosted on GitHub.

**Use a gender-neutral style** Often, you'll address the reader directly with *you*, *your* and *yours*. Otherwise, use gender-neutral pronouns *they*, *their*, and *theirs* or avoid pronouns entirely.

Wrong: A maintainer uploads the file. Then he...
Right: A maintainer uploads the file. Then they...
Right: A maintainer uploads the file. Then the maintainer...

**Headings** Write headings that use words the reader is searching for. A good way to do this is to have your heading complete an implied question. For example, a reader might want to know *How do I install MyLibrary?* so a good heading might be *Install MyLibrary*.

In section headings, use sentence case. In other words, write headings as you would write a typical sentence.

Wrong: Things You Should Know About Python
Right: Things you should know about Python

**Numbers** In body text, write numbers one through nine as words. For other numbers or numbers in tables, use numerals.

CHAPTER 9

# News

## 9.1 November 2017

- Introduced a new dependency management tutorial based on Pipenv. (#402)
- Updated the *Single Sourcing Package Version* tutorial to reflect pip's current strategy. (#400)
- Added documentation about the `py_modules` argument to `setup`. (#398)
- Simplified the wording for the `manifest.in` section. (#395)

## 9.2 October 2017

- Added a specification for the `entry_points.txt` file. (#398)
- Created a new guide for managing packages using `pip` and `virtualenv`. (#385)
- Split the specifications page into multiple pages. (#386)

## 9.3 September 2017

- Encouraged using `readme_renderer` to validate `README.rst`. (#379)
- Recommended using the *–user-base* option. (#374)

## 9.4 August 2017

- Added a new, experimental tutorial on installing packages using `Pipenv`. (#369)
- Added a new guide on how to use `TestPyPI`. (#366)

- Added `pypi.org` as a term. (#365)

## 9.5 July 2017

- Added `flit` to the key projects list. (#358)
- Added `enscons` to the list of key projects. (#357)
- Updated this guide's `readme` with instructions on how to build the guide locally. (#356)
- Made the new `TestPyPI` URL more visible, adding note to homepage about pypi.org. (#354)
- Added a note about the removal of the explicit registration API. (#347)

## 9.6 June 2017

- Added a document on migrating uploads to `PyPI.org`. (#339)
- Added documentation for `python_requires`. (#338)
- Added a note about PyPI migration in the *Tool Recommendations* tutorial. (#335)
- Added a note that `manifest.in` does not affect wheels. (#332)
- Added a license section to the distributing guide. (#331)
- Expanded the section on the `name` argument. (#329)
- Adjusted the landing page. (#327, #326, #324)
- Updated to Sphinx 1.6.2. (#323)
- Switched to the PyPA theme. (#305)
- Re-organized the documentation into the new structure. (#318)

## 9.7 May 2017

- Added documentation for the `Description-Content-Type` field. (#258)
- Added contributor and style guide. (#307)
- Documented `pip` and `easy_install`'s differences for per-project indexes. (#233)

## 9.8 April 2017

- Added travis configuration for testing pull requests. (#300)
- Mentioned the requirement of the `wheel` package for creating wheels (#299)
- Removed the `twine register` reference in the *Distributing Packages* tutorial. (#271)
- Added a topic on plugin discovery. (#294, #296)
- Added a topic on namespace packages. (#290)
- Added documentation explaining prominently how to install `pip` in `/usr/local`. (#230)

- Updated development mode documentation to mention that order of local packages matters. (#208)
- Convert readthedocs link for their `.org -> .io` migration for hosted projects (#239)
- Swaped order of `setup.py` arguments for the upload command, as order is significant. (#260)
- Explained how to install from unsupported sources using a helper application. (#289)

## 9.9 March 2017

- Covered `manylinux1` in *Platform Wheels*. (#283)

## 9.10 February 2017

- Added **PEP 518**. (#281)

Welcome to the *Python Packaging User Guide*, a collection of tutorials and references to help you distribute and install Python packages with modern tools.

This guide is maintained on GitHub by the Python Packaging Authority. We happily accept any *contributions and feedback*.

---

**Note:** Looking for guidance on migrating from legacy PyPI to *pypi.org*? Please see *Migrating to PyPI.org*.

---

CHAPTER 10

# Get started

Essential tools and concepts for working with the Python packaging ecosystem are covered in our *Tutorials* section:

- to learn how to install packages, see the *tutorial on installing packages*.
- to learn how to manage dependencies in a version controlled project, see the *tutorial on managing application dependencies*.
- to learn how to package and distribute your projects, see the *tutorial on packaging and distributing*

CHAPTER 11

# Learn more

Beyond our *Tutorials*, this guide has several other resources:

- the *Guides* section for walk throughs, such as *Installing pip/setuptools/wheel with Linux Package Managers* or *Packaging binary extensions*
- the *Discussions* section for in-depth references on topics such as *Deploying Python applications* or *pip vs easy_install*
- the *PyPA Specifications* section for packaging interoperability specifications

Additionally, there is a list of *other projects* maintained by members of the Python Packaging Authority.

# Index