Python Packaging Guide

Release

PyPA

June 24, 2016

Contents

Last Reviewed 2015-09-08

The "Python Packaging User Guide" (PyPUG) aims to be the authoritative resource on how to package, publish and install Python distributions using current tools.

To follow the development of Python packaging, see the Python Packaging Authority.

This guide is maintained on github.

Tool Recommendations

Page Status Complete

Last Reviewed 2016-06-24

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

1.1 Installation Tool Recommendations

- Use *pip* to install Python *packages* from *PyPI*. [1] [2] Depending 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) or *Hashdist*, or *conda* (both primarily focused on the scientific community).

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]

---

[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.

[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 Mac OS X 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.

- Use twine for uploading distributions to *PyPI*.

Installing Packages

Page Status Complete

Last Reviewed 2016-06-24

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

2.1 Requirements for Installing Packages

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

2.1.1 Install pip, setuptools, and wheel

- If you have Python 2 >=2.7.9 or Python 3 >=3.4 installed from python.org, you will already have *pip* and *setuptools*, but will need to upgrade to the latest version:

    On Linux or OS X:

    ```
    pip install -U pip setuptools
    ```

    On Windows:

    ```
    python -m pip install -U pip setuptools
    ```

- If you're using a Python install on Linux that's managed by the system package manager (e.g "yum", "apt-get" etc...), and you want to use the system package manager to install or upgrade pip, then see *Installing pip/setuptools/wheel with Linux Package Managers*

- Otherwise:

- 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.

2.1.2 Optionally, Create a virtual environment

See *section below* for details, but here's the basic commands:

    Using *virtualenv*:

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

    Using venv: [3]

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

2.2 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

---

[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*.

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 viable 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.

2.3 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.

There are a few cases where you might want to use easy_install instead of pip. For details, see the the *pip vs easy_install* breakdown in the Advanced Topics section.

2.4 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".

2.5 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.

2.6 Upgrading packages

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

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

2.7 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.

2.8 Requirements files

Install a list of requirements specified in a Requirements File.

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

2.9 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.git#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
```

---

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

2.10 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
```

2.11 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>
```

2.12 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
```

2.13 Installing Prereleases

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

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

2.14 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
```

Packaging and Distributing Projects

Page Status Complete

Last Reviewed 2015-09-08

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 Installing Packages.

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

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*).

3.2 Configuring your Project

3.2.1 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 reStructuredText 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 in certain cases where you need to package additional files that `python setup.py sdist (or bdist_wheel)` don't automatically include. 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

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

---

[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.

<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 include in the PyPA sample project

3.2.2 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, and will determine how your project is listed on *PyPI*. For details on permitted characters, see the **name** section from **PEP 426**.

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 Project Version*" Advanced Topics section.

description

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

Give a short and long description for you 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 the categorize your project. For a full listing, see https://pypi.python.org/pypi?%3Aaction=list_classifiers.

keywords

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

List keywords that describe your project.

packages

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

It's 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*.

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).

---

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.

3.2.3 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.

---

[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.

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
```

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 not be installed in 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. 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.

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.

3.4.1 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`.

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

Universal Wheels

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

To build a Universal 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. contains 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, OSX, 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 only allows uploads of platform wheels for Windows and OS X, NOT linux. Currently, the wheel tag specification (**PEP 425**) does not handle the variation that can exist across linux distros.

3.5 Uploading your Project to PyPI

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

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.

3.5.1 Create an account

First, you need a *PyPI* user account. There are two options:

1. Create an account manually using the form on the PyPI website.

2. **(Not recommended):** Have an account created as part of registering your first project (not recommended due to the related security concerns, see option #3 below).

If you created your account using option #1 (the form), you'll need to manually write a ~/.pypirc file like so.

```
[distutils]
index-servers=pypi

[pypi]
repository = https://upload.pypi.io/legacy/
username = <username>
password = <password>
```

You can leave out the password line if you use twine with its `-p PASSWORD` argument or prefer to simply enter your password when prompted.

3.5.2 Register your project

Next, if this is the first release, you currently need to explicitly register your project prior to uploading.

There are three ways to do this:

1. Use the form on the PyPI website, to upload your `PKG-INFO` info located in your local project tree at `myproject.egg-info/PKG-INFO`. If you don't have that file or directory, then run `python setup.py egg_info` to have it generated.

2. Run `twine register dist/mypkg.whl`, and *twine* will register your project based on the package metadata in the specified files. Your `~/.pypirc` must already be appropriately configured for twine to work.

3. **(Not recommended):** Run `python setup.py register`. If you don't have a user account already, a wizard will create one for you. This approach is covered here due to it being mentioned in other guides, but it is not recommended 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.

3.5.3 Upload your distributions

Finally, you can upload your distributions to *PyPI*.

There are two options:

1. Use *twine*

```
twine upload dist/*
```

The biggest reason to use twine is that `python setup.py upload` (option #2 below) uploads files over plaintext. This means anytime you use it you expose your username and password to a MITM attack. Twine uses only verified TLS to upload to PyPI in order to protect your credentials from theft.

Secondly it allows you to precreate your distribution files. `python setup.py upload` only allows you to upload something that you've created in the same command invocation. This means that you cannot test the exact file you're going to upload to PyPI to ensure that it works before uploading it.

Finally it allows you to pre-sign your files and pass the .asc files into the command line invocation (`twine upload twine-1.0.1.tar.gz twine-1.0.1.tar.gz.asc`). This enables you to be assured that you're typing your gpg passphrase into gpg itself and not anything else since *you* will be the one directly executing `gpg --detach-sign -a <filename>`.

2. **(Not recommended):** Use *setuptools*:

```
python setup.py sdist bdist_wheel upload
```

This approach is covered here due to it being mentioned in other guides, but it is not recommended 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.

Additional Topics

Last Reviewed 2015-09-08

This section covers a variety of packaging concepts and topics that don't fit neatly into the documentation of any particular *project* or in either of our guides on Installing Packages or Packaging and Distributing Projects.

4.1 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.

4.1.1 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
```

4.1.2 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
```

4.1.3 openSUSE

- Python 2:

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

- Python 3:

---

[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.

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

4.1.4 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.

4.1.5 Arch Linux

- Python 2:

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

- Python 3:

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

4.2 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

- *An overview of binary extensions*
  - *Use cases*
  - *Disadvantages*
  - *Alternatives to handcoded accelerator modules*
  - *Alternatives to handcoded wrapper modules*
  - *Alternatives for low level system access*
- *Implementing binary extensions*
- *Building binary extensions*
  - *Setting up a build environment on Windows*
- *Publishing binary extensions*

4.2.1 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 (regardling 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.

4.2.2 Implementing binary extensions

```
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 :)
```

4.2.3 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 Mac OS X binary compatibility requirements
cover the vagaries of Linux distros and other *nix systems
```

4.2.4 Publishing binary extensions

```
FIXME

cover publishing as wheel files on PyPI or a custom index server
cover creation of Windows and Mac OS X installers
mention the fact that Linux distros have a requirement to build from
source in their own build systems anyway, so pre-built binaries for
*nix systems currently aren't common
```

4.3 Supporting multiple Python versions

Page Status Incomplete

Last Reviewed 2014-12-24

> **Contents**
>
> - *Automated Testing and Continuous Integration*
> - *Tools for single-source Python packages*
> - *What's in Which Python?*

```
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.

4.3.1 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 Mac OSX environment. The Linux environment is Ubuntu 12.04 LTS Server Edition 64 bit while the OSX 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 OSX 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.

4.3.2 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.

4.3.3 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
```

4.4 Patching & Forking

Page Status Incomplete

Last Reviewed 2014-12-24

```
FIXME

- locally patch 3rd-part projects to deal with unfixed bugs
   - old style pkg_resources "patch releases": 1.3-fork1
   - PEP440's local identifiers: http://www.python.org/dev/peps/pep-0440/#local-version-identifiers
- fork and publish when you need to publish a project that depends on the fork
   (DONT use dependency links)
```

4.5 PyPI mirrors and caches

Page Status Incomplete

Last Reviewed 2014-12-24

Contents

- *Caching with pip*
- *Caching with devpi*
- *Complete mirror with bandersnatch*

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*.

4.5.1 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
```

4.5.2 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.

4.5.3 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.

4.6 Single-sourcing the Project Version

Page Status Complete

Last Reviewed 2015-12-03

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):

```
def read(*names, **kwargs):
    with io.open(
        os.path.join(os.path.dirname(__file__), *names),
        encoding=kwargs.get("encoding", "utf8")
    ) 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.

4.7 install_requires vs Requirements files

Page Status Complete

Last Reviewed 2015-09-08

> **Contents**
>
> - *install_requires*
> - *Requirements files*

4.7.1 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]

4.7.2 Requirements files

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

---

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

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`.

4.8 pip vs easy_install

Page Status Complete

Last Reviewed 2015-09-08

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

4.9 Wheel vs Egg

Page Status Complete

Last Reviewed 2015-09-10

---

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

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.

4.10 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.

---

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

4.10.1 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.

4.10.2 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).

4.10.3 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.

4.10.4 Mac OS X installers and package managers

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

Mac OS X 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

4.10.5 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.

4.10.6 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, OSX, 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).

4.11 Application Deployment

Page Status Incomplete

Last Reviewed 2014-11-11

Contents

- *Overview*
    - *Supporting multiple hardware platforms*
- *OS Packaging & Installers*
    - *Windows*
        - *Pynsist*
- *Application Bundles*
- *Configuration Management*

4.11.1 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.
```

4.11.2 OS Packaging & Installers

```
FIXME

- Building rpm/debs for projects
- Building rpms/debs for whole virtualenvs
- Building Mac OS X installers for Python projects
```

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.

4.11.3 Application Bundles

```
FIXME

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

4.11.4 Configuration Management

```
FIXME

puppet
salt
chef
ansible
fabric
```

4.12 Multi-version Installs

Page Status Complete

Last Reviewed 2014-12-27

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.

4.13 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

- *Background*
- *Setting Up*
- *Adding Appveyor support to your project*
    - *appveyor.yml*
    - *Support script*
    - *Access to the built wheels*
- *Additional Notes*
    - *Testing with tox*
    - *Automatically uploading wheels*
    - *External dependencies*
    - *Support scripts*

4.13.1 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.

4.13.2 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.

4.13.3 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

```
1   environment:
2
3     matrix:
4
5       # For Python versions available on Appveyor, see
6       # http://www.appveyor.com/docs/installed-software#python
7       # The list here is complete (excluding Python 2.6, which
8       # isn't covered by this document) at the time of writing.
9
10      - PYTHON: "C:\\Python27"
11      - PYTHON: "C:\\Python33"
12      - PYTHON: "C:\\Python34"
13      - PYTHON: "C:\\Python35"
14      - PYTHON: "C:\\Python27-x64"
15      - PYTHON: "C:\\Python33-x64"
16        DISTUTILS_USE_SDK: "1"
```

```
17      - PYTHON: "C:\\Python34-x64"
18        DISTUTILS_USE_SDK: "1"
19      - PYTHON: "C:\\Python35-x64"
20
21  install:
22    # We need wheel installed to build wheels
23    - "%PYTHON%\\python.exe -m pip install wheel"
24
25  build: off
26
27  test_script:
28    # Put your test command here.
29    # If you don't need to build C extensions on 64-bit Python 3.3 or 3.4,
30    # you can remove "build.cmd" from the front of the command, as it's
31    # only needed to support those cases.
32    # Note that you must use the environment variable %PYTHON% to refer to
33    # the interpreter you're using - Appveyor does not do anything special
34    # to put the Python evrsion you want to use on PATH.
35    - "build.cmd %PYTHON%\\python.exe setup.py test"
36
37  after_test:
38    # This step builds your wheels.
39    # Again, you only need build.cmd if you're building C extensions for
40    # 64-bit Python 3.3/3.4. And you need to use %PYTHON% to get the correct
41    # interpreter
42    - "build.cmd %PYTHON%\\python.exe setup.py bdist_wheel"
43
44  artifacts:
45    # bdist_wheel puts your built wheel in the dist directory
46    - path: dist\*
47
48  #on_success:
49  #  You can use this step to upload your artifacts to a public website.
50  #  See Appveyor's documentation for more details. Or you can simply
51  #  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.

4.13.4 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:

code/build.cmd

```
1  @echo off
2  :: To build extensions for 64 bit Python 3, we need to configure environment
3  :: variables to use the MSVC 2010 C++ compilers from GRMSDKX_EN_DVD.iso of:
4  :: MS Windows SDK for Windows 7 and .NET Framework 4
5  ::
6  :: More details at:
7  :: https://github.com/cython/cython/wiki/64BitCythonExtensionsOnWindows
8
9  IF "%DISTUTILS_USE_SDK%"=="1" (
10     ECHO Configuring environment to build with MSVC on a 64bit architecture
11     ECHO Using Windows SDK 7.1
12     "C:\Program Files\Microsoft SDKs\Windows\v7.1\Setup\WindowsSdkVer.exe" -q -version:v7.1
```

```
13      CALL "C:\Program Files\Microsoft SDKs\Windows\v7.1\Bin\SetEnv.cmd" /x64 /release
14      SET MSSdk=1
15      REM Need the following to allow tox to see the SDK compiler
16      SET TOX_TESTENV_PASSENV=DISTUTILS_USE_SDK MSSdk INCLUDE LIB
17  ) ELSE (
18      ECHO Using default MSVC build environment
19  )
20
21  CALL %*
```

4.14 Hosting your Own Simple Repository

Page Status Complete

Last Reviewed 2015-09-24

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.

4.14.1 "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.

PyPA Specifications

Page Status Complete

Last Reviewed 2016-01-22

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

5.1 Package distribution metadata

5.1.1 Core metadata

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

However, the version specifiers and environment markers sections of that PEP have been superceded as described below. In addition, metadata files are permitted to contain the following additional field:

Provides-Extra (multiple use)

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:`.

5.1.2 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**.

5.1.3 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**.

5.1.4 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.

5.1.5 Binary Distribution Format

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

5.1.6 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.

5.1.7 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.

5.2 Package index interfaces

5.2.1 Simple repository API

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

Project Summaries

Page Status Complete

Last Reviewed 2016-06-24

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

6.1 PyPA Projects

6.1.1 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.

6.1.2 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.

6.1.3 packaging

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

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

6.1.4 pip

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

A tool for installing Python packages.

---

[2] Multiple projects reuse the distutils-sig mailing list as their user list.
[1] pip was created by the same developer as virtualenv, and early on adopted the virtualenv mailing list, and it's stuck ever since.

6.1.5 Python Packaging User Guide

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

This guide!

6.1.6 setuptools

Docs | User list [2] | Dev list | Issues | Bitbucket | 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.

6.1.7 twine

Mailing list [2] | Issues | Github | PyPI

Twine is a utility for interacting with PyPI, that offers a secure replacement for `setup.py upload`.

6.1.8 virtualenv

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

A tool for creating isolated Python environments.

6.1.9 Warehouse

Docs | Mailing list [2] | Issues | Github | Dev irc:#pypa-dev

The new unreleased PyPI application which can be previewed at https://warehouse.python.org/.

6.1.10 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.

6.2 Non-PyPA Projects

6.2.1 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).

6.2.2 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.

6.2.3 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.

6.2.4 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.

6.2.5 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.

6.2.6 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`.

6.3 Standard Library Projects

6.3.1 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.

6.3.2 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.

6.3.3 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*.

Glossary

Page Status Complete

Last Reviewed 2015-09-08

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.

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.

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.

How to Get Support

Page Status Complete

Last Reviewed 2015-09-08

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.