Packaging and Distributing Projects

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

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

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see Building and Distributing Packages in the setuptools docs, but note that some advisory content there may be outdated.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

Contents

- Requirements for Packaging and Distributing
- Configuring your Project
  - Initial Files
    - setup.py
    - setup.cfg
    - README.rst
    - MANIFEST.in
    - LICENSE.txt
    - <your package>
  - setup() args
    - name
    - version
    - description
    - url
    - author
    - license
    - classifiers
    - keywords
    - packages
    - install_requires
    - python_requires
    - package_data
    - data_files
    - scripts
    - entry_points
      - console_scripts
  - Choosing a versioning scheme
    - Standards compliance for interoperability
    - Scheme choices
      - Semantic versioning (preferred)
      - Date based versioning
      - Serial versioning
      - Hybrid schemes
    - Pre-release versioning
    - Local version identifiers

v: latest

- Working in “Development Mode”
- Packaging your Project
  - Source Distributions
  - Wheels
    - Universal Wheels
    - Pure Python Wheels
    - Platform Wheels
- Uploading your Project to PyPI
  - Create an account
  - Upload your distributions

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).

Configuring your Project

Initial Files

setup.py

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

“setup.py” serves two primary functions:

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

setup.cfg

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

README.rst

All projects should contain a readme file that covers the goal of the project. The most common format is reStructuredText with an “rst” extension, although this is not a requirem v: latest

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 are not automatically included in a source distribution. To see a list of what's included by default, see the Specifying the files to distribute section from the distutils documentation.

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

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

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

LICENSE.txt

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

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

<your package>

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

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

setup() args

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

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

name

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

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

v: latest

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

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

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

version

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

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

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

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

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

description

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

Give a short and long description for 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

v: latest

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

Provide details about the author.

license

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

Provide the type of license you are using.

classifiers

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

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

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

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

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

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

keywords

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

List keywords that describe your project.

v: latest

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.

python_requires

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

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

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

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

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

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

And so on.

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

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

package_data

v: latest

```
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.

v: latest

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.

Choosing a versioning scheme

Standards compliance for interoperability

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

Here are some examples of compliant version numbers:

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

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

Scheme choices

Semantic versioning (preferred)

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

v: latest

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)

v: latest

- 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 appliec
```

Working in "Development Mode"

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

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

```
pip install -e .
```

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

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

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

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

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

v: latest

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

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

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

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

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

Packaging your Project

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

Source Distributions

Minimally, you should create a Source Distribution:

```
python setup.py sdist
```

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

Wheels

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

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

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

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

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

```
pip install wheel
```

v: latest

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 the wheel:

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

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

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

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

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

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

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

Pure Python Wheels

Pure Python Wheels that are not "universal" are wheels that are pure python (i.e. 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, macOS, or Windows, usually due to containing compiled extensions.

To build the wheel:

v: latest

```
python setup.py bdist_wheel
```

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

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

Uploading your Project to PyPI

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

Note: 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.

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

Create an account

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

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

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

Be aware that this stores your password in plaintext.

Upload your distributions

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

```
twine upload dist/*
```

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

v: latest

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

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

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

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

[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.

[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.

v: latest