Packaging and distributing projects

This section covers some additional details on configuring, packaging and distributing Python projects with `setuptools` that aren't covered by the introductory tutorial in Packaging Python Projects. It still assumes that you are already familiar with the contents of the Installing Packages 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see Building and Distributing Packages in the setuptools docs, but note that some advisory content there may be outdated.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

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

Unix/macOS **Windows**

```
python3 -m 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 / README.md

as well (look at `setup()`'s long_description_content_type argument).

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

Note: Projects using setuptools 0.6.27+ have standard readme files (`README.rst`, `README.txt`, or `README`) included in source distributions by default. The built-in distutils library adopts this behavior beginning in Python 3.7. Additionally, setuptools 36.4.0+ will include a `README.md` if found. If you are using setuptools, you don't need to list your readme file in `MANIFEST.in`. Otherwise, include it to be explicit.

MANIFEST.in

A `MANIFEST.in` is needed when you need to package additional files that are not automatically included in a source distribution. For details on writing a `MANIFEST.in` file, including a list of what's included by default, see "Including files in source distributions with MANIFEST.in".

However, you may not have to use a `MANIFEST.in`. For an example, the PyPA sample project has removed its manifest file, since all the necessary files have been included by setuptools 43.0.0 and newer.

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

LICENSE.txt

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

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

<your package>

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

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

setup() args

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

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

name

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

- 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,
long_description_content_type='text/x-rst',
```

Give a short and long description for your project.

These values will be displayed on PyPI if you publish your project. On `pypi.org`, the user interface displays `description` in the grey banner and `long_description` in the section named "Project Description".

`description` is also displayed in lists of projects. For example, it's visible in the search results pages such as https://pypi.org/search/?q=jupyter, the front-page lists of trending projects and new releases, and the list of projects you maintain within your account profile (such as https://pypi.org/user/jaraco/).

A content type can be specified with the `long_description_content_type` argument, which can be one of `text/plain`, `text/x-rst`, or `text/markdown`, corresponding to no formatting, reStructuredText (reST), and the Github-flavored Markdown dialect of Markdown respectively.

`url`

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

author

```
author='A. Random Developer',
author_email='author@example.com',
```

Provide details about the author.

license

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

The `license` argument doesn't have to indicate the license under which your package is being released, although you may optionally do so if you want. If you're using a standard, well-known license, then your main indication can and should be via the `classifiers` argument. Classifiers exist for all major open-source licenses.

The `license` argument is more typically used to indicate differences from well-known licenses, or to include your own, unique license. As a general rule, it's a good idea to use a standard, well-known license, both to avoid confusion and because some organizations avoid software whose license is unapproved.

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 :: 2.7',
    'Programming Language :: Python :: 3',
    'Programming Language :: Python :: 3.6',
    'Programming Language :: Python :: 3.7',
    'Programming Language :: Python :: 3.8',
    'Programming Language :: Python :: 3.9',
],
```

Provide a list of classifiers that categorize your project. For a full listing, see https://pypi.org/classifiers/.

Although the list of classifiers is often used to declare what Python versions a project supports, th v: latest s 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='sample setuptools development',
```

List keywords that describe your project.

project_urls

```
project_urls={
    'Documentation': 'https://packaging.python.org/tutorials/distributing-packages/',
    'Funding': 'https://donate.pypi.org',
    'Say Thanks!': 'http://saythanks.io/to/example',
    'Source': 'https://github.com/pypa/sampleproject/',
    'Tracker': 'https://github.com/pypa/sampleproject/issues',
},
```

List additional relevant URLs about your project. This is the place to link to bug trackers, source repositories, or where to support package development. The string of the key is the exact text that will be displayed on PyPI.

packages

```
packages=find_packages(include=['sample', 'sample.*']),
```

Set `packages` to a list of all packages in your project, including their subpackages, sub-subpackages, etc. Although the packages can be listed manually, `setuptools.find_packages()` finds them automatically. Use the `include` keyword argument to find only the given packages. Use the `exclude` keyword argument to omit packages that are not intended to be released and installed.

py_modules

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

If your project contains any single-file Python modules that aren't part of a package, set `py_modules` to a list of the names of the modules (minus the `.py` extension) in order to make setuptools aware of them.

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',
```

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

And so on.

> **Note:** Support for this feature is relatively recent. Your project's source distributions and wheels (see Packaging your project) must be built using at least version 24.2.0 of setuptools in order for the `python_requires` argument to be recognized and the appropriate metadata generated.
>
> In addition, only versions 9.0.0 and higher of pip recognize the `python_requires` metadata. Users with earlier versions of pip will be able to download & install projects on any Python version regardless of the projects' `python_requires` values.

package_data

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

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

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

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

data_files

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

Although configuring package_data is sufficient for most needs, in some cases you may need to place data files *outside* of your packages. The `data_files` directive allows you to do that. It is mostly useful if you need to install files which are used by other programs, which may be unaware of Python packages.

Each `(directory, files)` pair in the sequence specifies the installation directory and the files to install there. The `directory` must be a relative path (although this may change in the future, see wheel Issue #92), and it is interpreted relative to the installation prefix (Python's `sys.prefix` for a default installation; `site.USER_BASE` for a user installation). 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:** When installing packages as egg, `data_files` is not supported. So, if your project uses setuptools, you must use `pip` to install it. Alternatively, if you must use `python setup.py`, then you need to pass the `--old-and-unmanageable` option.

scripts

`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 Advertising Behavior 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)

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 w v: latest f dependencies to install.

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 commit since 1.2.0.dev1 release
1.2.1+fedora.4                # Package with downstream Fedora patches applied
```

Working in "development mode"

You can install a project in "editable" or "develop" mode while you're working on it. When installed as editable, a project can be edited in-place without reinstallation: changes to Python source files in projects installed as editable will be reflected the next time an interpreter process is started.

To install a Python package in "editable"/"development" mode Change directory to the root of the project directory and run:

```
python -m pip install -e .
```

The pip command-line flag `-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.

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

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

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

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

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

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

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


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

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.

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

Unix/macOS **Windows**

```
python3 -m pip install build
```

Source distributions

Minimally, you should create a Source Distribution:

Unix/macOS **Windows**

```
python3 -m build --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` and/or `setup.cfg`.

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 then you'll be creating a "Pure Python Wheel" (see section below).

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

Note: If your project also supports Python 2 *and* contains no C extensions, then you should create what's called a *Universal Wheel* by adding the following to your `setup.cfg` file:

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

Only use this setting if your project does not have any C extensions *and* supports Python 2 and 3.

Pure Python Wheels

Pure Python Wheels contain no compiled extensions, and therefore only require a single Python wheel.

To build the wheel:

Unix/macOS **Windows**

The `wheel` package will detect that the code is pure Python, and build a wheel that's named such that it's usable on any Python 3 installation. For details on the naming of wheel files, see **PEP 425**.

If you run `build` without `--wheel` or `--sdist`, it will build both files for you; this is useful when you don't need multiple wheels.

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:

Unix/macOS **Windows**

```
python3 -m build --wheel
```

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

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

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

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


```
twine check dist/*
```

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

Now you'll create a PyPI API token so you will be able to securely upload your project.

Go to https://pypi.org/manage/account/#api-tokens and create a new API token; don't limit its scope to a particular project, since you are creating a new project.

Don't close the page until you have copied and saved the token — you won't see that token again.

Note: To avoid having to copy and paste the token every time you upload, you can create a `$HOME/.pypirc` file:

```
[pypi]
username = __token__
password = <the token value, including the `pypi-` prefix>
```

Be aware that this stores your token in plaintext.

For more details, see the specification for `.pypirc`.

Upload your distributions

Once you have an account you can upload your distributions to PyPI using twine.

The process for uploading a release is the same regardless of whether or not the project already exists on PyPI - if it doesn't exist yet, it will be automatically created when the first release is uploaded.

For the second and subsequent releases, PyPI only requires that the version number of the new release differ from any previous releases.

```
twine upload dist/*
```

You can see if your package has successfully uploaded by navigating to the URL `https://pypi.org/project/<sampleproject>` where `sampleproject` is the name of your project that you uploaded. It may take a minute or two for your project to appear on the site.

[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.