python-packaging Documentation

Release 0.1

Scott Torborg

Sep 20, 2017

Contents

This tutorial aims to put forth an opinionated and specific pattern to make trouble-free packages for community use. It doesn't describe the *only* way of doing things, merely one specific approach that works well.

In particular, packages should make it easy:

- To install with `pip` or `easy_install`.
- To specify as a dependency for another package.
- For other users to download and run tests.
- For other users to work on and have immediate familiary with the basic directory structure.
- To add and distribute documentation.

We'll walk through the basic steps of building up a contrived package **funniest** to support these things.

CHAPTER 1

Minimal Structure

We'll start with some Python code. Native German speakers, please proceed with caution:

```
def joke():
    return (u'Wenn ist das Nunst\u00fcck git und Slotermeyer? Ja! ... '
            u'Beiherhund das Oder die Flipperwaldt gersput.')
```

The beauty and elegance of this implementation simply demands that it be packaged properly for distribution.

Picking A Name

Python module/package names should generally follow the following constraints:

- All lowercase
- Unique on pypi, even if you don't want to make your package publicly available (you might want to specify it privately as a dependency later)
- Underscore-separated or no word separators at all (don't use hyphens)

We've decided to turn our bit of code into a module called **funniest**.

Creating The Scaffolding

The initial directory structure for **funniest** should look like this:

```
funniest/
    funniest/
        __init__.py
    setup.py
```

The top level directory is the root of our SCM repo, e.g. `funniest.git`. The subdir, also called `funniest`, is the actual Python module.

For starters we'll put the `joke()` function in `__init__.py`, so it just contains:

```
def joke():
    return (u'Wenn ist das Nunst\u00fcck git und Slotermeyer? Ja! ... '
            u'Beiherhund das Oder die Flipperwaldt gersput.')
```

The main setup config file, `setup.py`, should contain a single call to `setuptools.setup()`, like so:

```
from setuptools import setup

setup(name='funniest',
      version='0.1',
      description='The funniest joke in the world',
      url='http://github.com/storborg/funniest',
      author='Flying Circus',
      author_email='flyingcircus@example.com',
      license='MIT',
      packages=['funniest'],
      zip_safe=False)
```

Now we can install the package locally (for use on our system), with:

```
$ pip install .
```

We can also install the package with a symlink, so that changes to the source files will be immediately available to other users of the package on our system:

```
$ pip install -e .
```

Anywhere else in our system using the same Python, we can do this now:

```
>>> import funniest
>>> print funniest.joke()
```

Publishing On PyPI

The `setup.py` script is also our main entrypoint to register the package name on PyPI and upload source distributions.

To "register" the package (this will reserve the name, upload package metadata, and create the pypi.python.org webpage):

```
$ python setup.py register
```

If you haven't published things on PyPI before, you'll need to create an account by following the steps provided at this point.

At this point you can view the (very minimal) page on PyPI describing **funniest**:

http://pypi.python.org/pypi/funniest/0.1

Although users can follow the URL link to find our git repository, we'll probably want to upload a source distribution so that the package can be installed without cloning the repository. This will also enable automated installation and dependency resolution tools to install our package.

First create a source distribution with:

```
$ python setup.py sdist
```

This will create `dist/funniest-0.1.tar.gz` inside our top-level directory. If you like, copy that file to another host and try unpacking it and install it, just to verify that it works for you.

That file can then be uploaded to PyPI with:

```
$ python setup.py sdist upload
```

You can combine all of these steps, to update metadata and publish a new build in a single step:

```
$ python setup.py register sdist upload
```

For a detailed list of all available setup.py commands, do:

```
$ python setup.py --help-commands
```

Installing the Package

At this point, other consumers of this package can install the package with `pip`:

```
$ pip install funniest
```

They can specify it as a dependency for another package, and it will be automatically installed when that package is installed (we'll get to how to do that later).

Adding Additional Files

Most of the time we'll want more than one file containing code inside of our module. Additional files should always be added inside the inner `funniest` directory.

For example, let's move our one function to a new `text` submodule, so our directory hierarchy looks like this:

```
funniest/
    funniest/
        __init__.py
        text.py
    setup.py
```

In `__init__.py`:

```
from .text import joke
```

In `text.py`:

```
def joke():
    return (u'Wenn ist das Nunst\u00fcck git und Slotermeyer? Ja! ... '
            u'Beiherhund das Oder die Flipperwaldt gersput.')
```

All additional Python code belongs in the `funniest/funniest/` directory.

Ignoring Files (.gitignore, etc)

There's one more thing we'll probably want in a 'bare bones' package: a `.gitignore` file, or the equivalent for other SCMs. The Python build system creates a number of intermediary files we'll want to be careful to not commit to source control. Here's an example of what `.gitignore` should look like for **funniest**:

```
# Compiled python modules.
*.pyc

# Setuptools distribution folder.
/dist/

# Python egg metadata, regenerated from source files by setuptools.
/*.egg-info
```

That's All You Need

The structure described so far is all that's necessary to create reusable simple packages with no 'packaging bugs'. If every published Python tool or library used followed these rules, the world would be a better place.

But wait, there's more! Most packages will want to add things like command line scripts, documentation, tests, and analysis tools. Read on for more.

CHAPTER 2

Specifying Dependencies

If you're using Python, odds are you're going to want to use other public packages from PyPI or elsewhere.

Fortunately, setuptools makes it easy for us to specify those dependencies (assuming they are packaged correctly) and automatically install them when our packages is installed.

We can add some formatting spice to the **funniest** joke with Markdown.

In `text.py`:

```
from markdown import markdown

def joke():
    return markdown(u'Wenn ist das Nunst\u00fcck git und Slotermeyer?'
                    u'Ja! ... **Beiherhund** das Oder die Flipperwaldt '
                    u'gersput.')
```

Now our package depends on the `markdown` package. To note that in `setup.py`, we just add an `install_requires` keyword argument:

```
from setuptools import setup

setup(name='funniest',
      version='0.1',
      description='The funniest joke in the world',
      url='http://github.com/storborg/funniest',
      author='Flying Circus',
      author_email='flyingcircus@example.com',
      license='MIT',
      packages=['funniest'],
      install_requires=[
          'markdown',
      ],
      zip_safe=False)
```

To prove this works, we can run `python setup.py develop` again, and we'll see:

```
$ python setup.py develop
running develop
running egg_info
writing requirements to funniest.egg-info/requires.txt
writing funniest.egg-info/PKG-INFO
writing top-level names to funniest.egg-info/top_level.txt
writing dependency_links to funniest.egg-info/dependency_links.txt
reading manifest file 'funniest.egg-info/SOURCES.txt'
writing manifest file 'funniest.egg-info/SOURCES.txt'
running build_ext
Creating /.../site-packages/funniest.egg-link (link to .)
funniest 0.1 is already the active version in easy-install.pth

Installed /Users/scott/local/funniest
Processing dependencies for funniest==0.1
Searching for Markdown==2.1.1
Best match: Markdown 2.1.1
Adding Markdown 2.1.1 to easy-install.pth file

Using /.../site-packages
Finished processing dependencies for funniest==0.1
```

When we publish this to PyPI, calling `pip install funniest` or similar will also install `markdown`.

Packages Not On PyPI

Sometimes you'll want to use packages that are properly arranged with setuptools, but aren't published to PyPI. In those cases, you can specify a list of one or more `dependency_links` URLs where the package can be downloaded, along with some additional hints, and setuptools will find and install the package correctly.

For example, if a library is published on GitHub, you can specify it like:

```
setup(
    ...
    dependency_links=['http://github.com/user/repo/tarball/master#egg=package-1.0']
    ...
)
```

CHAPTER 3

Better Package Metadata

The `setuptools.setup()` call accepts a variety of keyword arguments to specify additional metadata about your package. This can help people find your package and evaluate quickly whether or not it is what they're looking for.:

```
from setuptools import setup

setup(name='funniest',
      version='0.1',
      description='The funniest joke in the world',
      long_description='Really, the funniest around.',
      classifiers=[
        'Development Status :: 3 - Alpha',
        'License :: OSI Approved :: MIT License',
        'Programming Language :: Python :: 2.7',
        'Topic :: Text Processing :: Linguistic',
      ],
      keywords='funniest joke comedy flying circus',
      url='http://github.com/storborg/funniest',
      author='Flying Circus',
      author_email='flyingcircus@example.com',
      license='MIT',
      packages=['funniest'],
      install_requires=[
          'markdown',
      ],
      include_package_data=True,
      zip_safe=False)
```

For a full list of the possible arguments to `classifiers`, visit http://pypi.python.org/pypi?%3Aaction=list_classifiers.

A README / Long Description

You'll probably want a README file in your source distribution, and that file can serve double purpose as the `long_description` specified to PyPI. Further, if that file is written in reStructuredText, it can be formatted nicely.

For **funniest**, let's add two files:

```
funniest/
    funniest/
        __init__.py
    setup.py
    README.rst
    MANIFEST.in
```

`README.rst` contains:

```
Funniest
--------

To use (with caution), simply do::

    >>> import funniest
    >>> print funniest.joke()
```

`MANIFEST.in` contains:

```
include README.rst
```

This file is necessary to tell setuptools to include the README.rst file when generating source distributions. Otherwise, only Python files will be included.

Now we can use it in setup.py like:

```
from setuptools import setup

def readme():
    with open('README.rst') as f:
        return f.read()

setup(name='funniest',
      version='0.1',
      description='The funniest joke in the world',
      long_description=readme(),
      classifiers=[
        'Development Status :: 3 - Alpha',
        'License :: OSI Approved :: MIT License',
        'Programming Language :: Python :: 2.7',
        'Topic :: Text Processing :: Linguistic',
      ],
      keywords='funniest joke comedy flying circus',
      url='http://github.com/storborg/funniest',
      author='Flying Circus',
      author_email='flyingcircus@example.com',
      license='MIT',
      packages=['funniest'],
      install_requires=[
          'markdown',
      ],
```

```
    include_package_data=True,
    zip_safe=False)
```

When the repo is hosted on GitHub or BitBucket, the `README.rst` file will also automatically be picked up and used as a 'homepage' for the project.

CHAPTER 4

Let There Be Tests

The **funniest** package needs some tests. These should be placed in a submodule of `funniest.` so that they can be imported, but won't pollute the global namespace.:

```
funniest/
    funniest/
        __init__.py
        tests/
            __init__.py
            test_joke.py
    setup.py
    ...
```

The `test_joke.py` file is our first test file. Although it's overkill for now, we'll use a `unittest.TestCase` subclass to provide infrastructure for later development.:

```
from unittest import TestCase

import funniest

class TestJoke(TestCase):
    def test_is_string(self):
        s = funniest.joke()
        self.assertTrue(isinstance(s, basestring))
```

The best way to get these tests going (particularly if you're not sure what to use) is Nose. With those files added, it's just a matter of running this from the root of the repository:

```
$ pip install nose
$ nosetests
```

To integrate this with our `setup.py`, and ensure that Nose is installed when we run the tests, we'll add a few lines to `setup()`:

```
setup(
    ...
    test_suite='nose.collector',
    tests_require=['nose'],
)
```

Then, to run tests, we can simply do:

```
$ python setup.py test
```

Setuptools will take care of installing nose and running the test suite.

CHAPTER 5

Command Line Scripts

Many Python packages include command line tools. This is useful for distributing support tools which are associated with a library, or just taking advantage of the setuptools / PyPI infrastructure to distribute a command line tool that happens to use Python.

For **funniest**, we'll add a `funniest-joke` command line tool.

There are two mechanisms that `setuptools.setup()` provides to do this: the `scripts` keyword argument, and the `console_scripts` entry point.

The `scripts` Keyword Argument

The first approach is to write your script in a separate file, such as you might write a shell script.:

```
funniest/
    funniest/
        __init__.py
        ...
    setup.py
    bin/
        funniest-joke
    ...
```

The `funniest-joke` script just looks like this:

```
#!/usr/bin/env python

import funniest
print funniest.joke()
```

Then we can declare the script in `setup()` like this:

```
setup(
    ...
```

```
    scripts=['bin/funniest-joke'],
    ...
)
```

When we install the package, setuptools will copy the script to our PATH and make it available for general use.:

```
$ funniest-joke
```

This has advantage of being generalizeable to non-python scripts, as well: `funniest-joke` could be a shell script, or something completely different.

The `console_scripts` Entry Point

The second approach is called an 'entry point'. Setuptools allows modules to register entrypoints which other packages can hook into to provide certain functionality. It also provides a few itself, including the `console_scripts` entry point.

This allows Python *functions* (not scripts!) to be directly registered as command-line accessible tools.

In this case, we'll add a new file and function to support the command line tool:

```
funniest/
    funniest/
        __init__.py
        command_line.py
        ...
    setup.py
    ...
```

The `command_line.py` submodule exists only to service the command line tool (which is a convenient organization method):

```
import funniest

def main():
    print funniest.joke()
```

You can test the "script" by running it directly, e.g.:

```
$ python
>>> import funniest.command_line
>>> funniest.command_line.main()
...
```

The `main()` function can then be registered like so:

```
setup(
    ...
    entry_points = {
        'console_scripts': ['funniest-joke=funniest.command_line:main'],
    }
    ...
)
```

Again, once the package has been installed, we can use it in the same way. Setuptools will generate a standalone script 'shim' which imports your module and calls the registered function.

This method has the advantage that it's very easily testable. Instead of having to shell out to spawn the script, we can have a test case that just does something like:

```
from unittest import TestCase
from funniest.command_line import main

class TestConsole(TestCase):
    def test_basic(self):
        main()
```

In order to make that more useful, we'll probably want something like a context manager which temporarily captures `sys.stdout`, but that is outside the scope of this tutorial.

CHAPTER 6

Adding Non-Code Files

Often packages will need to depend on files which are not `.py` files: e.g. images, data tables, documentation, etc. Those files need special treatment in order for setuptools to handle them correctly.

The mechanism that provides this is the `MANIFEST.in` file. This is relatively quite simple: `MANIFEST.in` is really just a list of relative file paths specifying files or globs to include.:

```
include README.rst
include docs/*.txt
include funniest/data.json
```

In order for these files to be copied at install time to the package's folder inside `site-packages`, you'll need to supply `include_package_data=True` to the `setup()` function.

Note: Files which are to be used by your installed library (e.g. data files to support a particular computation method) should usually be placed inside of the Python module directory itself. E.g. in our case, a data file might be at `funniest/funniest/data.json`. That way, code which loads those files can easily specify a relative path from the consuming module's `__file__` variable.

CHAPTER 7

Putting It All Together

Our whole package, for reference, looks like this:

```
funniest/
    funniest/
        __init__.py
        command_line.py
        tests/
            __init__.py
            test_joke.py
            test_command_line.py
    MANIFEST.in
    README.rst
    setup.py
    .gitignore
```

Here is the contents of each file:

funniest/__init__.py

```
from markdown import markdown


def joke():
    return markdown(u'Wenn ist das Nunst\u00fcck git und Slotermeyer?'
                    u'Ja! ... **Beiherhund** das Oder die Flipperwaldt '
                    u'gersput.')
```

funniest/command_line.py

```
from . import joke


def main():
    print joke()
```

funniest/tests/__init__.py

(empty)

funniest/tests/test_joke.py

```
from unittest import TestCase

import funniest


class TestJoke(TestCase):
    def test_is_string(self):
        s = funniest.joke()
        self.assertTrue(isinstance(s, basestring))
```

funniest/tests/test_command_line.py

```
from unittest import TestCase

from funniest.command_line import main


class TestCmd(TestCase):
    def test_basic(self):
        main()
```

MANIFEST.in

```
include README.rst
```

README.rst

setup.py

```
from setuptools import setup


def readme():
    with open('README.rst') as f:
        return f.read()


setup(name='funniest',
      version='0.1',
      description='The funniest joke in the world',
      long_description=readme(),
      classifiers=[
        'Development Status :: 3 - Alpha',
        'License :: OSI Approved :: MIT License',
        'Programming Language :: Python :: 2.7',
        'Topic :: Text Processing :: Linguistic',
      ],
      keywords='funniest joke comedy flying circus',
      url='http://github.com/storborg/funniest',
      author='Flying Circus',
      author_email='flyingcircus@example.com',
      license='MIT',
      packages=['funniest'],
      install_requires=[
```

```
            'markdown',
        ],
        test_suite='nose.collector',
        tests_require=['nose', 'nose-cover3'],
        entry_points={
            'console_scripts': ['funniest-joke=funniest.command_line:main'],
        },
        include_package_data=True,
        zip_safe=False)
```

.gitignore

```
# Compiled python modules.
*.pyc

# Setuptools distribution folder.
/dist/

# Python egg metadata, regenerated from source files by setuptools.
/*.egg-info
/*.egg
```

CHAPTER 8

About This Tutorial / Contributing

Scott Torborg - storborg@gmail.com.

I wrote this tutorial in an attempt to improve the state of Python packages at large. Tools like virtualenv and pip, as well as improvements to setuptools, have made the Python package ecosystem a delight to work in.

However, I frequently run across packages I want to use that don't interoperate correctly with others, or find myself in situations where I'm not sure exactly how to structure things. This is an attempt to fix that.

This documentation is written in reStructuredText and built with Sphinx. The source is open and hosted at http://github.com/storborg/python-packaging.

To build the HTML version, just do this in a clone of the repo:

```
$ make html
```

Contributions and fixes are welcome, encouraged, and credited. Please submit a pull request on GitHub or email me.

Note: At this time, this documentation focuses on Python 2.x only, and may not be *as* applicable to packages targeted to Python 3.x.

See also:

Setuptools Documentation Core setuptools documentation and API reference.