Packaging your Python

Calum Chamberlain

March 9, 2016

Outline

Introduction

This is my self-taught, opinion and experience driven guide. It is not necessarily correct, and it shouldn't serve as a do-it-all guide. Rather it is an intro to the key ideas and some things I have found useful.

There is a whole world of cool stuff out there, you won't be short of things to help you on your way!

One of my favourites is:
`https://the-hitchhikers-guide-to-packaging.readthedocs.org/en/latest`

Why package?

Python is great because it is so easy to get packages and use them. This means you don't have to re-invent the wheel - the chances are someone has already done what you want to do, and have probably done it well. If not, then someone else probably wants the code too, so you could help them!

Packaging for Python projects is incredibly simple, it takes about half an hour to go from a (functional) project to one that you can distribute and install.

Why test?

Making a *good* Python package takes more effort, but is worth it.
I will try and show you how to document and test alongside packaging, both of these make your project usable, both for other people, and your future self.
Testing your code is at the heart of modern reproducible research and should not be overlooked.
Tests are probably the most important parts of any package.

File structure

Example for project named: *addition*:

```
addition/
    setup.py
    README.md
    LICENCE.txt
    addition/
        __init__.py
        addition.py
```

Key files: __init__.py

This can have no useful code in it, it just tells the python interpreter that the directory is a package.

At a minimum the __init__.py file should have a comment.

Usually they will contain rules for imports like:
*from addition import **

__init__.py example

```python
"""
init file for base package.
"""
__all__ = ['core', 'utils', 'par']

__version__ = '0.1.1'
```

Listing 1: __init__.py example

The __all__ method is what is called by *import **.

Versioning

Your package needs a version number: versioning allows users to track how recent their system is and work out if it needs to be updated. By putting the version number in the __init__.py file versions can be checked by:

```
import addition
print(addition.__version__)
```

Key files: setup.py

The setup.py file controls a lot of things:

- Installation
- Testing
- Distributing to pypi
- Handling dependencies
- Package meta-data

To do this it uses the distutils or setuptools package.

setup.py example

```
from setupttools import setup, find_packages
import addition
long_description = open('README.md").read()

setup(
  name='addition',
  version=addition.__version__
  description='Simple addition of integers'
  long_description=long_description
  author='Montgomary Viper'
  author_email='albatross_gerkin@walrus.com'
  license='LICENCE.txt'
  packages=['addition']
  install_requires=[
    "numpy",
    "obspy >= 1.0.0"
  ],
)
```

unittest

unittest is a useful way to write tests that will return useful errors when run.
Some rules for writing tests:

- Make test names long and useful;
- Make tests easy to understand and as simple as possible;
- Build in bugs reported by users in as test cases (you don't want to fix something only to break it again later);
- Use full words for variable names, no shorthand, people need to know what's going on!

https://docs.python.org/2/library/unittest.html

unittest: example

```
"""Simple testing suite for addition package
"""
import unittest
from addition.addition import add_indices

class TestAddition(unittest.TestCase):

  def test_add(self):
    """Test adding of indices"""
    self.assertEqual(add_indices(1,2), 3)
    self.assertFalse(add_indices(4,20) == 25)
```

pytest

pytest (`http://pytest.org`) is a way of calling all of your tests in a simple and integrated way.

There are other options, for example nose (`nose.readthedocs.org`) is popular too.

Both work well, I use pytest because it was easier to set up and has simpler syntax if you don't use unittest.

pytest

You can set up tests in your setup.py file with lines like:

```
tests_require=['pytest']
```

which will require that you have that module to run the tests.

In a complementary file, not yet mentioned, you can set aliases, this is the setup.cfg file. To call tests by running:

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

You need an alias like:

```
[aliases]
test=pytest
```

Travis

Travis CI is probably the most popular way to install and test your code as you go, I like it because:

- Webhooks for github;
- Automatically builds and runs tests quickly in the cloud on push;
- Feeds back to github, which can be used to stall merges until tests pass;
- Can run multiple environments (e.g. different operating systems, different Python versions, and different versions of packages).

The docs are pretty damn good: `https://docs.travis-ci.com/`

Tox

Tox helps manage different versions of python and other possibilities.
https://tox.readthedocs.org/en/latest/.

I don't use Tox at the moment, mostly because I don't support Python 2.6 or 3.x (2.6 is too old and multiprocessing isn't built for 3.x, other modules are available though).

Documentation

Golden rule:
If it's not documented, it doesn't exist.

Sphinx

Sphinx makes it easy to generate documentation without leaving your code-base.
Format doc-strings correctly and sphinx (with autodoc) will make you all sorts:

- html;
- latex (pdf);
- epub;
- man-pages.

Sphinx: `http://www.sphinx-doc.org`

Doc-strings

```python
def add_indices(a, b):
    """
    Simple function to add two integers together.

    :type a: int
    :param a: First integer to be added.
    :type b: int
    :param b: Second integer to add to a.

    :returns: int
    """
    return a + b
```

Listing 2: Python Doc-string example

Doc hosting

ReadTheDocs is nice
Otherwise PyPi offers docs hosting.

https://readthedocs.org/

I like ReadTheDocs because it has Sphinx inbuilt and has a github plugin, so docs are continuously updated as the package updates. Which simplifies things and keeps docs always current.

github

Lots of cool, open-source science (and other things) are hosted on github:
`https://github.com/showcases/science`

Organisations like GeoNet and USGS have hosting on github:
`https://github.com/usgs`
`https://github.com/GeoNet`

Cool github things

- Issue tracking, e.g. `https://github.com/obspy/obspy/issues`;
- Branching, e.g. `https://github.com/obspy/obspy/network`
- Native docs (wiki) e.g. `https://github.com/obspy/obspy/wiki`;
- Lots of useful commenting things for collaborative work.

pypi

Go to site for python packages.

Install by pip or easy_install.

`https://pypi.python.org/pypi`

Anaconda

`https://docs.continuum.io/anaconda/index`

Summary

1. Use standard directory structure;
2. Make your modules discoverable with __init__.py files;
3. Use a setup.py file with distutils to manage install and tests;
4. Write tests for all your functions, no matter how small;
   - Use unittest to test functions;
   - Run tests using pytest or nose or similar;
   - Use a continuous integration service to run your tests automatically for you;
5. Use sphinx to generate docs from in-place doc-strings;
6. Distribute your code on pypi (and other services).