A Gentle Introduction to Python Packaging

Nathan Simpson

REG Tech Talks

2023-10-17

Packaging == take your code with you
(to unknown places)

All steps here are encompassed in the Python packaging authority (PyPA) tutorial

don’t get mixed up with PyPI (Python package index), which is where you upload your package for others to use :)

First: writing a function to convert from Celsius to Fahrenheit

We can use this code in a different file, provided it shares the directory we're in!

What if we’re not working in the directory?

pyproject.toml setup

Note: this has auto-scanned for *.py files. (setuptools as default backend). setup.py used find_packages() I think.

Multiple files?

pip install now throws error:

error: Multiple top-level modules discovered in a flat-layout: ['temperature', 'distance’].

It gives some advice — we will choose one, and switch to a src-layout.

src/: look here

Renaming module + adding functions in the top-level scope

Adding a numpy dependency

- update pyproject.toml
- update demo notebook

Adding tests

- install pytest
- python -m pytest (try which pytest to show what happens otherwise)

A Gentle Introduction to Python Packaging, part 2

Nathan Simpson

REG Tech Talks

2023-10-24

Recap

pyproject.toml is all you need to install packages!

All configuration for your library and tools should go in here.

Examples: dependencies, pytest configuration, black options, optional install extras...

Use a src/mymodule folder structure

```
├── pyproject.toml
├── examples
│   └── demo.ipynb
├── src
│   └── convertlib
│   - ├── __init__.py
│   - ├── distance.py
│   - └── temperature.py
├── tests
│   ├── test_distance.py
│   └── test_temperature.py
```

Dependency management

A **dependency** is something that *your library code *needs** to run.
put another way: if it's not imported in your library code, it doesn't go in your library.

- have seen coding patterns in multiple projects @ Turing that accidentally introduce dependencies on libraries that are not used!
- this coding pattern makes sense — it’s often something you definitely use *with* your library (but not inside it), so don't we "depend" on it in a way?

Example: limiting your library by depending on a complicated module

So, where do I put a module that I use, but don't depend on?

First: good examples that show how the libraries are meant to be used together!

Is it in your test suite?

if so, these modules should be in optional extras (we'l ounds on dependencies: introduce one in the test suite.

- Every bound should preferably be documented!
- Think about where your requirements come from.

The unholy commit.

how do we fix it?

code formatting!

but… how do we prevent it?

— code formatting that takes place pre-commit?

Let’s add pre-commit to our project!

Why not depend on pre-commit in dev?

- also exists on brew for mac (brew install pre-commit), so i generally use that pre-commit for all projects (config file is all that matters)
  - Just a habit i picked up from others — i don't see any reason for it to break anything.

Let’s run it and see the results!

Now let's get fancy: pre-commit run automatically by GitHub Actions (GHA)!

This will let us double-check that our code went through pre-commit correctly.

An example

This happened when developing this repo!

Mismatch between local pre-commit and GHA?

We can also add our test suite as part of these checks!

This is done by running pytest in the yml workflow.

Even fancier?

You can also automate the testing of your examples!

Distributing your code

Two steps:

- Building the right files
  - source distribution (sdist): basically just this repo zipped up!
  - built distribution (wheel): a single binary file that has all the code compiled and ready to use

Before uploading your code:

- Make an account on test.pypi.org
  - 2-factor authentication is needed here!
- Get an API token (you can choose to scope it per-project or one for all projects)
- Set ~/.pypirc with the right user (__token__) and the token as the password:

```
[testpypi]
 username = __token__
 password = <your-api-key>
```

The commands

```
python3 -m pip install --upgrade
build twine
```

```
python3 -m build
```

```
python3 -m twine upload --repository
testpypi dist/*
```

Did it work?

Let's do it for real:

- Make an account on pypi.org (same 2FA demand)
- Repeat all steps as we did for test PyPI, but replace testpypi with pypi.

The commands

```
python3 -m pip install --upgrade
build twine
```

```
python3 -m build
```

```
python3 -m twine upload --repository
pypi dist/*
```

How am I expected to remember all this??

You don't have to :)

Soft-launching right now: a Python project template!

github.com/alan-turing-institute/python-project-template

Supports setuptools (this talk), hatch, poetry.

Licences

These are important! Will not spend much time on this today (not Python-specific).

Since you're likely planning on sharing your code in some way (public source/package), you need to let people know what they can & can't do with it.

See: https://choosealicense.com/licenses/

Other important files

- *CONTRIBUTING.md*: A guide for new contributors on how they can work on the repository.
- *CODE_OF_CONDUCT.md*: The expectations for how people involved with this project should conduct themselves (behaviour, communication, what is/isn't tolerated)
- *CITATION.cff*: If your repository is to be made citable or linked to a paper, this is essential! Look up Citation File Format for more info.

Some other ways of doing things

My advice for tools is: **use the tools that your collaborators use** (or are willing to change to).

If it's your own project, do whatever you want! - my first PhD project was done entirely in Jupyter notebooks (search for nbdev)

Poetry

All-in-one solution to environments, package management, development in general.

Great if you know it, can be difficult if you don't. Non-standard pyproject.toml, non-standard commands.

If you use poetry or want to use it, please read this blog post!

It makes some versioning assumptions that can cause issues (and at one point, they added a random 5% chance for features to be deprecated in CI...)

https://iscinumpy.dev/post/poetry-versions/

conda/mamba (check out mamba)

Does fundamentally more than any of these tools (can manage entire software stacks that aren't Python).

You may encounter issues outside of your computer (don't expect all clusters to come with conda installs and environments).

But can be very nice to not cause software bloat if you have a lot of projects that depend on a big data science stack.

The last slide

This talk: github.com/phinate/python-packaging

PyPA tutorial: packaging.python.org/en/latest/tutorials/packaging-projects

Template: github.com/alan-turing-institute/python-project-template [contributions or suggestions are completely welcome!]

Thank you, this was lots of fun! :)