Writing Python Packages

December 6, 2019

“Good” research

What does good research look like?

Good research is:

How can we make sure our research reflects these values?

What does good research look like?

Good research is:

1. Accurate: Mitigate potential errors

How can we make sure our research reflects these values?

What does good research look like?

Good research is:

1. Accurate: Mitigate potential errors
2. Collaborative: Two brains are better than one

How can we make sure our research reflects these values?

What does good research look like?

Good research is:

1. Accurate: Mitigate potential errors
2. Collaborative: Two brains are better than one
3. Constructive: Engages with previous research

How can we make sure our research reflects these values?

What does good research look like?

Good research is:

1. Accurate: Mitigate potential errors
2. Collaborative: Two brains are better than one
3. Constructive: Engages with previous research
4. Foundational: Provides foundation for others to engage with your work

How can we make sure our research reflects these values?

Themes of this weekend

This weekend we will cover tools that help us achieve these values:

- Code style, code modularity, and package development
- Unit and integration testing
- Model testing

Code structure and style

Stylistic Python code

Coding is a form of communication...

> *Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. Code for readability. — John Woods*
>
> *Measuring programming progress by lines of code is like measuring aircraft building progress by weight — Bill Gates*

“Non-negotiable” stylistic rules

These are rules I feel (maybe overly) strongly about,

- A space after EVERY comma… Except when trailing — `foo = (0,)`
- Avoid wildcard imports (i.e. `from <module> import *`)
- Four spaces for block indentation… No more, no less.
- Limit lines of code to roughly 80 characters
- Try to organize code into logical blocks
- No space before or after a colon

Stylistic choices

I feel less strongly about some of these rules, but I find that they improve readability for me

- Two lines before and after a function definition
- Don’t repeat yourself (DRY)
- Don’t align your =

Examples of good code

```python
import numpy as np
from scipy.linalg import lstsq


# Create data
y = np.array([1.0, 4.0, 3.0, 2.0])
x = np.array([
    [1.0, 0.5],
    [1.0, 2.0],
    [1.0, 1.5],
    [1.0, 1.0]
])

# Find coefficients
coeffs, resid, _, _ = lstsq(x, y)
```

Examples of bad code

```
import numpy as np
from scipy.linalg import *
y               = np.array([1.0, 4.0, 3.0,2.0])
x               = np.array([[1.0,0.5],[1.0,2.0],[1.0, 1.5],[1.0,
coeffs,resid,_,_ = lstsq(x,y)
```

Motivational exercise!

Exercise: Why write packages?

We are going to begin by doing some economics for two reasons:

1. We are economists — **How will these tools help us as economists?**
2. Understanding the costs associated with using low quality code helps motivate me to produce high quality code

Hendricks Leukhina 2017

Our exercise will be to replicate analysis done in *How risky is college investment* by Lutz Hendricks and Oksana Leukhina.

- One of the key components of risk to consider when deciding to pursue college is "failure to graduate" risk
- To motivate the credit accumulation component of their structural model, the authors propose a simplified model of credit accumulation
- We will only replicate a small piece of their paper (Section 2)

The model

Each individual begins as a college freshman with $n_0 = 0$ college credits

- Individual's draw an ability level $a_i \sim N(0, 1)$
- An individual's HS GPA is correlated with their ability level, $\text{GPA} = a + \varepsilon$ where $\varepsilon \sim N(0, \sigma^2)$
- Each year, student attempts 12 courses each worth 3 credits. They pass each course with a probability given by $p(a)$
- A student graduates if they accumulate 125 credits within 6 years and fail to graduate otherwise

Data

The authors use restricted-use microdata from National Center for Education Statistics which has college transcript data for each student in the HS&B survey.

They "calibrate" their model through a simulated method of moments with a target of 10 moments:

1. The correlation between credits earned by a student the first two years of college
2. The 10th/20th/.../80th/90th quantiles of total credits earned after two years

Exercise instructions

Everyone should clone the transcripty repository

We will break into 4 groups and each group will receive a group number. There are a few constraints that groups should satisfy:

- The person who is “driving” should not have been the “driver” in the previous session
- The people who “drove” last time should separate themselves amongst the groups
- The groups should roughly be the same size

Exercise

See Project 1

We will work on this for 30-45 mins

Debrief

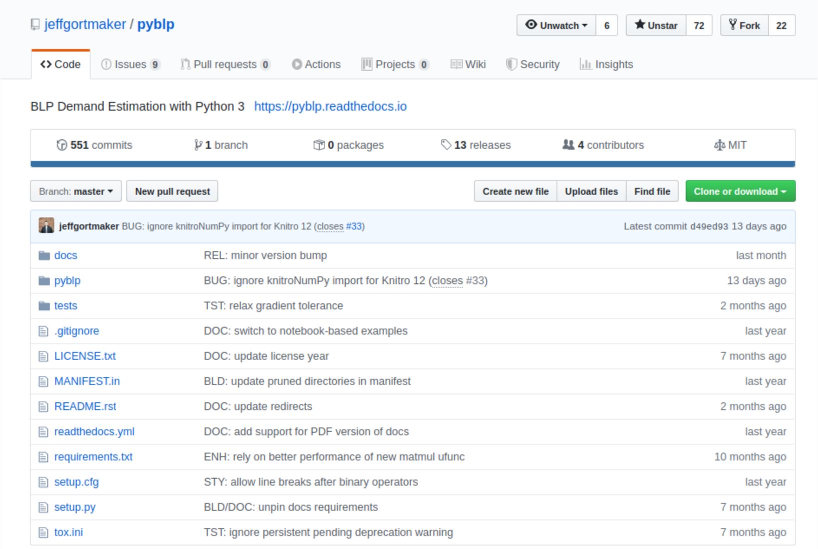
How did you find the exercise?

Which components of code were helpful?

What slowed you down?

Package development

Example package: pyblp



Package structure

Most Python packages share a similar structure:

- `docs`: This folder contains the files that help generate the documentation.
- `<package_name>`: This folder contains the package source code and is what will be loaded by Python when `import <package_name>` is called.
- `tests`: This folder contains the unit and integration tests
- `README.{md,rst,txt}`: This is a file that introduces the package and will be rendered on the github landing page.
- `LICENSE.txt`: You should not use a package that is not licensed because (according to people who know more about the law than I do) your work is under an exclusive copyright by default. Useful to read through https://choosealicense.com to learn a bit more about each license. I tend to prefer relatively permissive Apache/BSD/MIT licenses as opposed to copyleft licenses like GPL.
- Various files used to build and maintain the package. These files include `setup.py`, `readthedocs.yml`, `requirements.txt`, etc...

Package development

Package source code

Package source code: structure

The package source code is organized into a collection of files and folders. We will refer to the folders within the package as "sub-packages"[1].

In `pyblp` there is a class called `SimulationMarket` in `pyblp/markets/simulation_market.py`.

There are two ways to access this class in your Python code...

[1] Some people get technical and refer to the elements inside of a package as modules, sub-modules, and sub-packages, but we're going to try and avoid these details for now...See this SO question if you'd like to know more of the details

Package source code: structure

First, we could do:

```
import pyblp.markets  # We could create an alias here!
pyblp.markets.SimulationMarket(args...)
```

Package source code: structure

or, we could do:

```
from pyblp.markets import SimulationMarket
SimulationMarket(args...)
```

Package source code: `__init__.py`

Your Python package should have an `__init__.py` file within the main source code directory and within each sub-package.

This file used to be required in order to identify a directory as a package, but, as of Python 3.3, it is no longer a requirement.

However, we think it is still a good idea to include this file because it will be run whenever the package is imported and can be a good place to initialize relevant features or expose particular methods.[2]

[2] Can find more details in this blogpost

Package source code: setuptools

To make your package installable, you need to create a `setup.py` file which uses the setuptools package

Package source code: setuptools

To make your package installable, you need to create a `setup.py` file which uses the setuptools package

After creating a `setup.py` file, the package can be installed with
`python setup.py install`

Package source code: setuptools

To make your package installable, you need to create a `setup.py` file which uses the setuptools package

After creating a `setup.py` file, the package can be installed with `python setup.py install`

What's in a `setup.py` file? Only `name`, `version`, and `packages` are required arguments to `setup`, but it's useful to add include many others

Package development

Documentation

Documentation: writing docstrings

Documentation is your chance to provide your users (including yourself!) with instructions on what each function does and how it should be used.

The vast majority of functions should have a doc string.

I like to follow the numpydoc docstring guide, but any format you like is fine as long as you're consistent.

Documentation: numpydoc docstring conventions

```python
def myfunction(arg1, arg2):
    """
    This paragraph provides a description of the function.

    Parameters
    ----------
    arg1 : type(arg1)
        Description of arg1
    arg2 : type(arg2)
        Description of arg2

    Returns
    -------
    return_value : type(return_value)
        Description of return value

    See Also
    --------
    other_function : This is a related function
    """
    ...
```

Documentation: sphinx

Step 1: Create a folder called `docs`. Within that folder, run the command `sphinx-quickstart` from a terminal

Step 2: Edit `conf.py`

- Ensure that the first code in the file includes

```
import os
import sys
sys.path.insert(0, os.path.abspath(".."))
```

  this ensures that your packages will be found.
- Add relevant extensions to `extensions` variable — This includes at least `sphinx.ext.autodoc` and `sphinx.ext.napoleon`[3]

[3] The napoleon extension is required for sphinx to understand documentation that follows the numpy or google doc standards. The autodoc extensions is just to simplify your life.

Documentation: sphinx

Step 3: Generate documentation from your package by running `sphinx-apidoc -o source/ ../<package_name>` in a terminal in the `docs` directory — We could also do this step manually for more control on the documentation organization

Step 4: Run `make html` in a terminal from the `docs` directory

Step 5: Open `_build/html/index.html` in a browser and review the (excruciatingly ugly) docs that were generated! Clicking on *Module Index* will take you to a page which has the documentation for all of your functions!

Documentation: sphinx

The first step to making the documentation slightly less ugly is to edit the `index.rst` file. It helps to add an introduction to your package at the top of the page.

You can (and should) also write some instructions for how users could use your package[4].

[4] `pyblp` has an exceptionally good documentation page…You should mimic the level of detail included.

Documentation: Read the Docs

Read the Docs[5] is a site that hosts documentation (and, more importantly, automates documentation updates).

Many open source projects (especially in the Python world) use *Read the Docs* to host their documentation because of its ease of use and flexibility

[5] Many of the sites on their page can also be accessed with https://rtfd.org

Documentation: Read the Docs

Step 1: Create an account (I log in with my github account)

Step 2: Create a `readthedocs.yml` file for setting project specific configuration — You won't always need this but it doesn't hurt to have a placeholder

Step 3: Import your package using their online tool — Doing this takes care of the webhook that will automatically rebuild your docs whenever a push happens to the desired branch

Package development

Publishing Python packages

Python Package Index

The *Python Package Index* (PyPI) is "a repository of software for the Python programming language." There are over 200,000 projects currently on PyPI

Any package on PyPI should be able to be installed with pip[6] by running `pip install <package_name>`

[6] pip is a recursive acronym and stands for *pip installs packages*

Publish package on PyPI

Step 1: Install the `twine` package with `pip install twine`

Step 2: Register on PyPI (and test PyPI) — Today we'll upload our packages to the test index, but you would typically use the main one

Step 3: Create the *distribution package* which is essentially just a zipped file with your code using `python setup.py sdist bdist_wheel`

Publish package on PyPI

Step 4: Check that the long description will correctly render using `twine check dist/*`

Optional: You can create a keyring so that you won't have to type your username and password each time you upload a package — We won't do it today, but see the twine documentation for details

Step 5: Upload to `test.pypi.org` using `twine upload --repository-url https://test.pypi.org/legacy dist/*` — If you were uploading to main PyPI, you could simply leave out the `-repository-url` argument

Exercise: Writing your own package

Exercise: Writing your own package

Return to your groups from before. You will now be writing your own packages.

The instructions for this exercise can be found in Project 2.