Software Engineering for Data Scientists Course Cheatsheet

By: Ian Ozsvald (https://ianozsvald.com)
Date: 2022-02

Development strategies for younger data science projects

Generally our lives are easier if we follow the same coding practices and standards as others in the community. Contributing to an open source project is a great way to get exposed to these standards. By following the usual conventions (i.e. “the idiomatic way to write Python”) you write code that other teams members and new joiners are likely to understand, decreasing your maintenance burden and improving the lifetime value of your code.

Jupyter Notebooks

Notebooks are a great way to prototype new data science research code but they are poor for production systems. Converting your code to Python modules or scripts is normally the right approach.

Nbdime

One problem with Notebooks is that they package code and output as JSON encoded cells – images are base64 encoded and each cell has metadata.

The Nbdime (https://nbdime.readthedocs.io/en/latest/) project gives you “`nbdiff-web`” which lets you make a graphical diff of two versions of a Notebook in git or as files in the same folder. You can use this to enable collaboration via git with Notebooks, it diffs both code and images. Without a tool like this it becomes hard to see what’s changed between different versions of a Notebook.

Jupytext

Jupytext synchronises a pure-Python version of the code with the Notebook version of the code – you can edit either and changes are automatically reflected. This means you can run traditional code quality tools (such as `black` and `flake8` – noted below) on the Python script version to fix issues whilst working on the Jupyter Notebook version of the code.

I’d suggest using this sparingly at first as editing the “same contents” via two different files can lead to one set of changes overwriting the other (notably if you don’t refresh the Notebook display after you’ve saved a changed to the synchronised module), so go slowly whilst you experiment.

Using Jupytext you can easily extract methods from Notebooks to a module (e.g. a `utility.py` module), you can then add tests for that new function in the module.

Debugging

Understanding how the built-in debugger `pdb` works is important, it'll give you the mental model to understand how the debugging features in IDEs work. The Python docs have a good guide to the built-in commands: https://docs.python.org/3/library/pdb.html#debugger-commands

The easy way to start to use the debugger is to add `breakpoint()` statements into your code, once reached you'll drop into `pdb` (and you can use `q` to quit).

The `stackprinter` project (https://github.com/cknd/stackprinter) adds nicer tracebacks with richer debugging information. `watchpoints` (https://github.com/gaogaotiantian/watchpoints) watches for a change in variable state on variables you've marked and logs them to the screen.

Standard project layouts

The cookiecutter tool is used in all sorts of projects (not just in Python and not just for data science) to create a standard folder structure. The guide from Driven Data (https://drivendata.github.io/cookiecutter-data-science/) has good practice for combing experimental Notebooks, raw and cleaned data, reports, modules and tests in one repository.

setup.py

A clean installation file will let you use the `pip` tool to install your cookiecutter project as a local Python project, without copying it to the traditional `site-packages` folder. The benefit is that `sys.path` is configured to include the root folder of the project in the Python module path so you can import your own code from the `src/` folder.

The following is the basic setup.py included in the cookiecutter data science project, you can use as a template for your own projects:

----setup.py----

```
from setuptools import find_packages, setup
setup(
  name='my_research_project',
  packages=find_packages(),
  version='0.1.0',
  description='A short description of the project.',
  author='Ian',
  license='',
)
```

To configure your `sys.path` use "`pip install -e .`" which installs from this `setup.py` (in folder "." which is your current folder) in editable mode (i.e. it doesn't copy the project to the `site-packages` folder, it sets up a link from there to your folder via an `easy-install.pth` path file).

If you use a setup.py file with the pip install command shown above you can build well structured projects with utility files, data processing scripts and tests that live in separate folders – this is a far more maintainable for growing research projects.

If you *don't* use this process then try printing `sys.path` and you'll see that the current folder is in the search path, but if you want to import code from a parallel folder (e.g. you're working in `./notebooks/` and you want to import code from `./src/utility.py`). You can modify `sys.path` to add a hardcoded folder but using the pip install process is cleaner and can be easily repeated on new machines.

Cleaner code

Python code should conform to the PEP8 coding standards (see the code review document from the course and read https://www.python.org/dev/peps/pep-0008/). Many IDEs help with this formatting.

Many projects adopt `flake8` which applies these rules at the command line. If you go on to adopt `pre-commit` (not covered in the class – see https://pre-commit.com/) then you can add git commit hooks that run `flake8` prior to a commit, which can reject the commit if the code doesn't follow the PEP8 standards. Many open source projects (e.g. Pandas) require `flake8` to pass before code can be contributed to maintain a consistent style in the project.

The `black` project rewrites your `.py` source files to follow PEP8 automatically. Many projects and data scientists have adopted it to avoid internal discussions about "which variant of PEP8 might we prefer" – `black` takes that discussion away from you.

`nbqa` will run Notebook quality check tools on your Notebooks with `nbqa flake8 mynotebook.ipynb` or `nbqa black mynotebook.ipynb`.

The `isort` package will automatically sort imports in a Python module so that system-wide imports come first, well known packages come second and your project's imports come last. This is a useful way to automatically tidy up your imports (note it won't move imports half way through a script to the top of the file – you have to do this manually. `flake8` can raise warnings on this for you).

Read the contribution guidelines for Pandas and Scikit-learn to learn how other successful teams manage changes to the code to keep the style the same:

- https://pandas.pydata.org/pandas-docs/stable/development/contributing.html
- https://scikit-learn.org/stable/developers/contributing.html

The https://pypi.org/project/vulture/ project discovers unused code in your modules – code which you might otherwise waste time supporting or which lowers your unit test coverage metrics (see below). If you identify unused code the general advice would be to delete it.

Testing

Automated approaches to testing remove the need for manual testing (which is error prone and costly on time) and lets us use tools to identify under-tested components of our business critical code.

No tests in a project is generally a bad sign. Projects that make extensive use of Notebooks tend to have no tests because it is hard to integrate tests into the Notebooks. Moving to a module structure, extracting utility code and writing tests on the utilities is a common and sensible practice.

Adding tests after the fact (Test After Development) is better than having no tests for a section of code. Often with engineering code it is faster to develop code by having tests in place ahead of time using a Test Driven Development process.

Reviewing the test driven development guide to using PyTest for Pandas contributors may give you ideas to take back to your team: https://pandas.pydata.org/pandas-docs/stable/development/contributing.html#test-driven-development-code-writing

PyTest

Make predefined test cases (called Fixtures) so that you have pairs of an “expected input” and an “expected output”. If you have a list of these (such as a list of expected temperatures for a Fahrenheit to Celsius converter) you can use PyTest’s `parameterize` decorator so that each pair is executed as a new test – this simplifies your testing code and is a recognised Python testing idiom. See https://docs.pytest.org/en/stable/parametrize.html

`pytest` can drop into a debugger on a failure with the `--pdb` flag, enable more verbose reporting with `-v` and show print output with `-s`.

I like to use `pytest -s -v -l test_thing.py` (you can either specify one file like `test_thing.py` or if you omit that it’ll autodiscover the test files). You can specify one module, a wildcard for many modules, even one function in a module or in a class.

When we change our tests or our code it is convenient to have our tests automatically re-run themselves. If you have a Continuous Integration system then you may be able to run your tests every time you push to your repository and get notified of failures. To get feedback quickly the https://github.com/watchexec/watchexec project will “watch the specified files and run something for you” – we can use this to watch for Python files that change and to re-run our tests with “`watchexec --exts py pytest`”.

Coverage

The coverage project checks if you have lines of source code that are not covered by your unit tests – if you judge that the areas lacking tests are important to your project, you can quickly focus on improving the coverage in those areas. See https://github.com/pytest-dev/pytest-cov

Some teams strive for high code coverage metrics – e.g. aiming for 100% of source lines to be tested by at least one test. Generally I'd argue that lower test coverage, but with a focus on high-value functions that are critical to your business flow, is a better way to focus scarce programmer time. Both tests and source code have a maintenance cost so reducing the number of tests you write means you have less code to support, so a sensible balance needs to be found that offers enough testing value to the organisation without burdening it with tests of low value.

Doctest

It is important to make sure that the examples shown in code documentation correctly specify the behaviour of the function. Python's Doctests look for code snippets in the docstrings on a function, class and module and will run them, validating that the example output that's shown matches what is produced. Learn how to run doctests here: https://docs.python.org/3/library/doctest.html#simple-usage-checking-examples-in-docstrings

The Pandas guide to updating their documentation is very thorough including notes on the style to use when writing documentation and how to write useful doctests: https://pandas.pydata.org/pandas-docs/stable/development/contributing.html#about-the-pandas-documentation

Pandera

Pandera checks our DataFrames for the constraints we specify. Each column that you want to be checked should be specified with at least the type (e.g. `int`).

You can add one or more checks for each column and you can specify interdependencies between columns. You can require `strict` so that 1 check must exist for each column and `ordered` so that your column order must be matched by the dataframe.

I use Pandera whenever I've loaded data – to catch errors at their source, and when I'm manipulating data – so I don't allow bad data to slip downstream into my processing code which might confuse my debugging. They're easy to write and are well recommended.

About Ian Ozsvald

Ian is a Chief Data Scientist and Coach, he's helped co-organise the annual PyDataLondon conference with 700+ attendees and the associated 11,000+ member monthly meetup. He runs the established Mor Consulting Data Science consultancy in London, gives conference talks internationally often as keynote speaker and is the author of the bestselling O'Reilly book High Performance Python (2nd edition).

He has 18 years of experience as a senior data science leader, trainer and team coach. For fun he's walked by his high-energy Springer Spaniel, surfs the Cornish coast and drinks fine coffee. Past talks and articles can be found at: https://ianozsvald.com/

Ian authors the data scientist focused https://buttondown.email/NotANumber email newsletter.