From Raw Recruit Scripts to Perfect Python

Stanley van der Merwe, Petr Wolf

BARCLAYS

6 November 2019

PyData NYC

Disclaimer

The views expressed here are those of the authors and do not necessarily represent or reflect the views of Barclays

Links

Contacts

- https://www.linkedin.com/in/stanley-van-der-merwe/
- https://www.linkedin.com/in/petrwolf

Slides

- https://github.com/PetrWolf/pydata_nyc_2019/tutorial.pdf

Code and setup

- https://github.com/PetrWolf/pydata_nyc_2019

Abstract

Whether developing new models in Jupyter Notebooks or porting existing code from older infrastructure or other technologies (e.g. Excel, SAS), data scientists are often faced with disorganized structure, reproducibility issues or low run-time performance.

Model implementation quality and performance plays a critical role in successful deployment, continued use and future maintenance costs. Key drivers of this success include modularized code and tests that are well defined, both of which often get neglected or left out entirely.

In this tutorial we will start with a Jupyter Notebook that represents a sample model with typical shortcomings, such as a mixing of input data processing with model logic, missing tests, lack of usage examples or confusing code.

In a series of steps, we will incrementally refactor the code into intuitive modular python, using the best tools from the python ecosystem.

Audience level: Novice

https://github.com/PetrWolf/pydata_nyc_2019

You will learn to

- Structure your code in composable and re-usable blocks with in-line documentation and examples
- Catalog and organize boilerplate data sourcing (using Intake)
- Use automated testing (pytest and hypothesis) and static code analysis (PyLint) to guarantee code quality and reproducibility
- Analyze performance (cProfile, line_profiler) to identify hot-spots and guide run-time optimization
- Apply just-in-time compilation (JIT) and vectorization using numba for even faster performance

This tutorial is for you if you

- want to take the next step after beginner python tutorials
- mainly use Jupyter Notebooks for your work and want to add more tools to your toolbox
- want to help your team in improving code quality
- are in the process of migrating code or models from other technologies (SAS, Excel) and want to use best-practices from the start

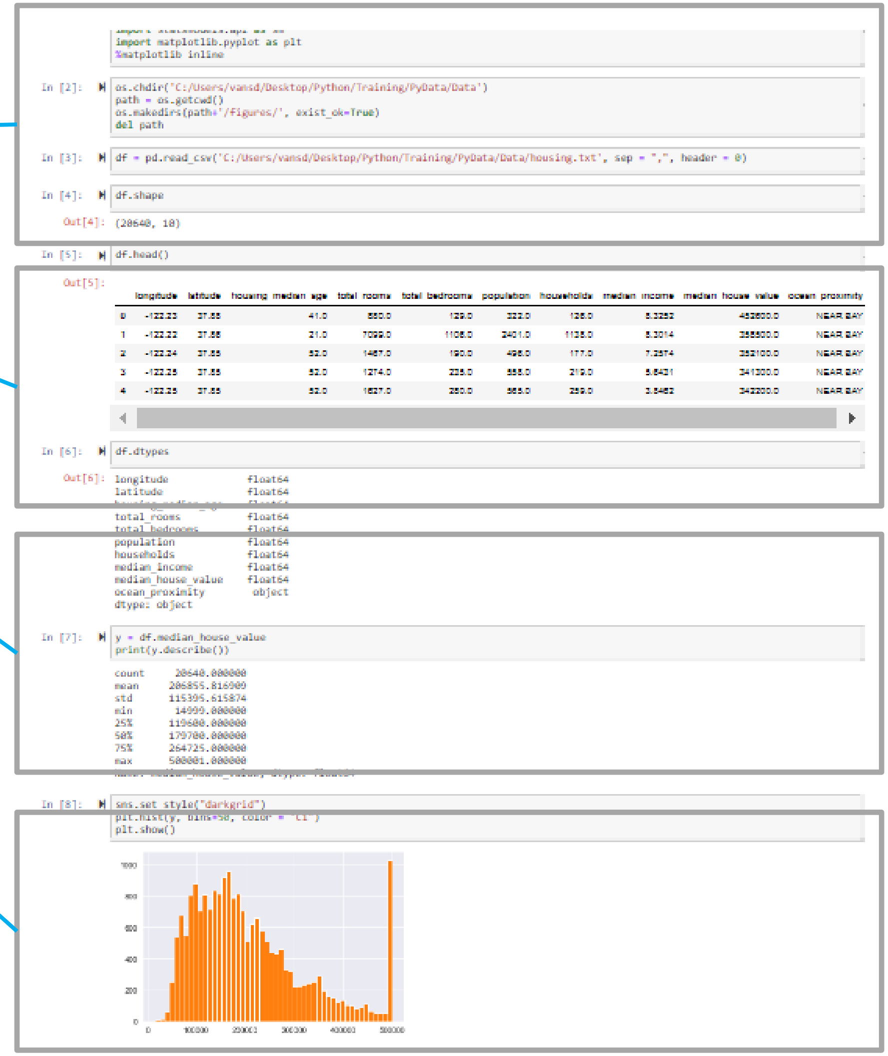
This tutorial may not be for you if you

- have extensive experience with writing modular and testable code
- have notable experience in automated testing and code optimization

Agenda

1. Setup
2. Reproducibility
3. Data sourcing
4. Unit testing
5. Code quality
6. Performance
7. Summary

Typical modeling notebook



Tutorial Structure

1. Instead of …

2. Do … instead

Tutorial Structure

3. Brief explanation/demo

4. Hands-on exercises

Tutorial Structure

5. Sample solution

6. Summary and recap

https://github.com/PetrWolf/pydata_nyc_2019

BARCLAYS

Out of Scope

These best practices are not included in this tutorial

- Version control
- Continuous integration
- Peer review
- Automated data validation
- Documentation

But you should still use them! See:

- [Best Practices for Scientific Computing]
- [Good enough practices in scientific computing]

Setup

On your own computer

1. git clone
https://github.com/PetrWolf/pydata_nyc_2019.git

2. follow README.md

- Create python environment
- Install required packages
- Launch Jupyter Notebook

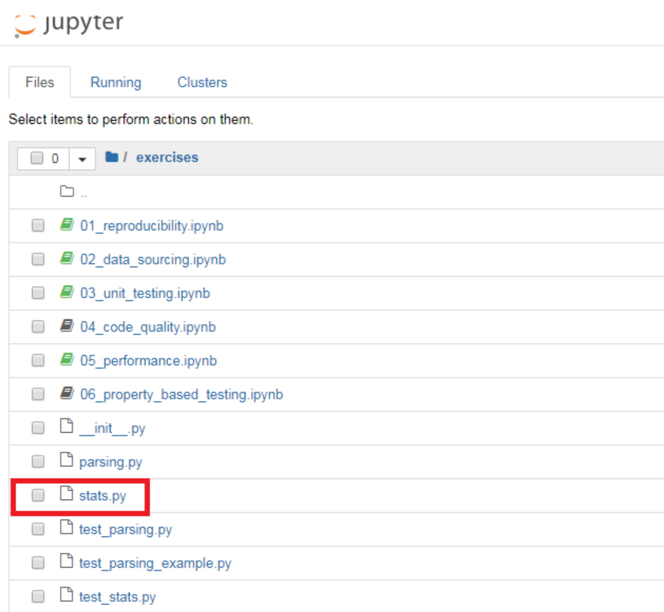
In the browser (using Binder)

Open
https://mybinder.org/v2/gh/PetrWolf/pydata_nyc_2019/

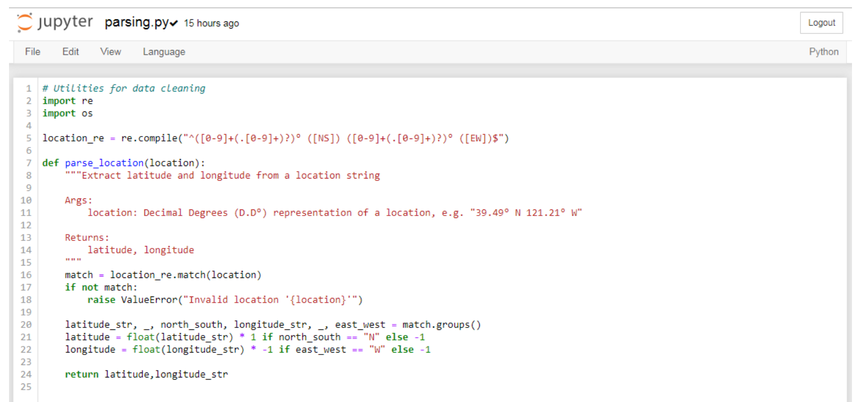
Tooling

- We're mostly going to use Jupyter Notebooks
- But also the built-in text editor

1. Click on a text or python file



2. Edit and save



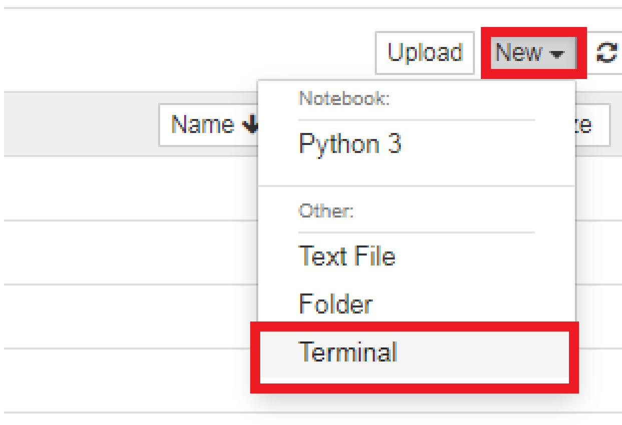
https://github.com/PetrWolf/pydata_nyc_2019

BARCLAYS

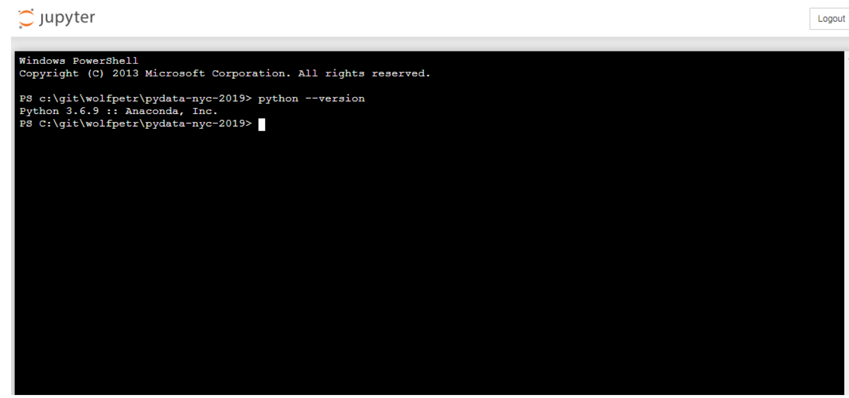
Tooling

- Built-in Terminal

Click New → Terminal



Enter commands



- Special Jupyter commands: [%<magic>]() or [!<command>]()

BARCLAYS

Reproducibility

- What it means?
- Why focus on reproducibility?
  - One-off analysis might have to be repeated
  - Model re-calibration, maintenance, monitoring
  - Reproducible projects promote knowledge sharing
  - Might have to move your project to production

Reproducibility – exercise

Open `exercises/01_reproducibility.ipynb`, follow the prepared steps

1. Use "[`pip freeze`]" (or "[`conda list`]") to list all installed packages
2. Check python version ([`sys.version`])

Advanced:

A. Consult [pandas changelog] and [python release history] for newer versions

B. Review `requirements.txt` in the project root folder

C. Check the [operating system version]

Reproducibility - review

Reproducibility matters – for collaboration, re-use, maintenance

Ways of making projects reproducible

- Make dependencies and requirements explicit
- Use version control
- Add tests to your code to validate it works as expected
- Work with multiple people and use multiple-environments

See also

- Reproducibility in ML Systems: A Netflix Original by Ferras Hamad (PyData NYC 2019)
- Up your Bus Number: A Reproducible Data Science Workflow by Kjell Wooding (PyData NYC 2018)
- Talk Python Episode #227: Maintainable data science: Tips for non-developers
- Creating Reproducible Data Science Projects by Justin Boylan-Toomey

Data Sourcing

- Finding, defining and loading data costs time and effort
- Basic need to know: what is available? what format? how to obtain?
- Intake provides clear split between users of data and providers of data
- Intake seeks to provide a consistent approach to organizing and loading data sources by providing

  A. Catalogs - descriptions, arguments, metadata and plugins

  B. Plugins – to support additional data formats

    – Built-in: csv, numpy, textfiles, …

    – Extensible: avro, parquet, Spark, S3 and many more

Data Sourcing - Exercise

Open `exercises/02_data_sourcing.ipynb` and follow the prepared steps

1. Review the prepared catalog file (`data/catalog.yml`)
2. load the catalog and data in the notebook using `open_catalog()` and `read()` respectively. Is everything correct?
3. Further edit the catalog file in `data/catalog.yml` and reload the data to achieve a clean state

Advanced

- Replace a local “housing.csv” with a corresponding URL
- Allow customizing the Github account in the data source URL via user parameters
- Add plots (see docs for Intake and hvplot)

Data Sourcing - Summary

Summary

- Standardized and re-usable data loading code
- Clean and simple usage in notebook

More info

- Intake Plugin Directory
- Documentation on Catalogs and writing custom plugins
- Intake tutorials
- Intake - taking the pain out of data access by Martin Durant (PyData NYC 2018)

Testing

- How do you know your code works as expected?
- And how will you keep it working when used later/by others?
- Immensely challenging to verify and maintain validity of code over time
- Assertions in the code help with checking assumptions
- Automated testing – unit tests, integration tests and regression tests
- Positive vs negative testing

Testing - Exercise

Open `exercises/03_unit_testing.ipynb` and follow the prepared steps

1. Review the function `parse_location()` in `parsing.py`
2. Use `pytest test_parsing.py` in the terminal to run tests
3. Add more test functions or asserts. Did you find any more issues?
4. Add several sample negative cases. Do they behave as expected?
5. Repeatedly run pytest in the terminal and fix any issues

Advanced

A. Use Pytest Parameters to organize similar test cases

B. Use `pytest.raises` with `match=` to test the expected error message

C. Run "pytest --doctest-modules" to also test the example in the function

Testing - Summary

Summary

- Test driven code builds trust and maintains validity of code
- Testing encourages better design which leads to better understanding and reusability

See also:

- Advanced Software Testing for Data Scientists by Raoul-Gabriel Urma (PyData NYC 2019)

Code Quality

- How does one improve code quality?
- By modularizing, re-using and making code readable
- Modularizing code naturally leads to readable, reusable and testable code
- Decompose programs into functions – not too many parameters
- Re-use code instead of rewriting it
- Explain your function
- Give functions and variables meaningful names
- Lot of this is subjective – use tools to evaluate and automatically check for known issues and detect bugs

Code Quality - PyLint

Open `exercises/04_code_quality.ipynb` and follow the prepared steps

1. Run `pylint parsing.py` and review the output
2. Use `--disable=<error code>` to turn off "pesky" conventions
3. Inspect reported issues and try fixing them
4. Review a sample file `bad_code.py`. Can you see any issues?
5. Run `pylint bad_code.py`. Does the output look useful?

Advanced

A. Use `pylint --generate-rcfile` to generate a config file

B. Review the settings and find the most interesting features

Code Quality - Summary

PyLint (homepage, docs)

- Python tool for static code analysis
- Checks coding style (PEP8) and detects errors and bad patterns
- Integrated with many editors and CI tools
- Customizable and extensible

See also

- Other linters (flake8, pyflakes, pycodestyle)
- Static checkers (mypy)
- Code formatters (black)

Performance

- Optimize code only after it works correctly
- Determine whether it's actually worth speeding it up
- Use a profiler to identify bottlenecks (intuition is often misleading)
- Line_profiler and cProfile will check where your program is spending time
- Once bottlenecks are identified try improving code
- Prefer built-in methods in NumPy/Pandas/Scipy (already optimized)
- Operate on columns instead of rows
- Use Numba to compile code in hotspots

Performance - Exercise

Open `exercises/05_performance.ipynb` and follow the prepared steps

1. Use `%prun` and `%lprun` to profile execution of a RMSE calculation
2. Try improving the code and make it faster
3. Use `%timeit` to quickly test different versions
4. Add `@numba.njit()` and compare timing using `%timeit`

Advanced:

A. Try adding [`parallel=True`](). Did it help performance?

B. Profile and optimize `calc_lift()` from the Tutorial notebook.

Performance - Summary

- Use Jupyter "magics" `%time` and `%timeit` to measure execution speed
- Use `%prun` and `%lprun` to identify hot spots
- Use columnar operations and library implementations (already optimized)
- Use Numba to further speed-up custom code

More info

- Memory profiling (`%memit`, `%mprun`)
- Vectorization in Numba (`@vectorize` and `@guvectorize`)
- Numba talks and tutorials
- Python performance tips, Pandas Enhancing performance

Summary

- Write programs for people not computers
- Make code readable, reusable and testable
- Optimize once you have something that works
- Collaborate and share knowledge

Thank you!

Q&A

BARCLAYS