Upgrade your Python code quality/ Clean code/ Some nice Python tools

Mauro Luzzatto

mauro.luzzatto@gess.ethz.ch

Thursday, 15th April 2021

About me

Background:

- 2014 – 2017 Master in Energy Science and Technology, Department of Mechanical Engineering
- 2018 CAS in Computer Science
- Since 2018 Data Scientist at IBM
- Since 2019 Scientific Assistant, Law, Economics, and Data Science Group, Part-Time (10%)

Projects at the Law, Economics, and Data Science Group:

- Legal Entropy
- Algorithmic Explanations

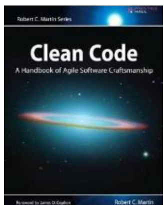
Interests:

- Hiking, Running, Guitar
- Python, NLP, influence of Data Science on our society

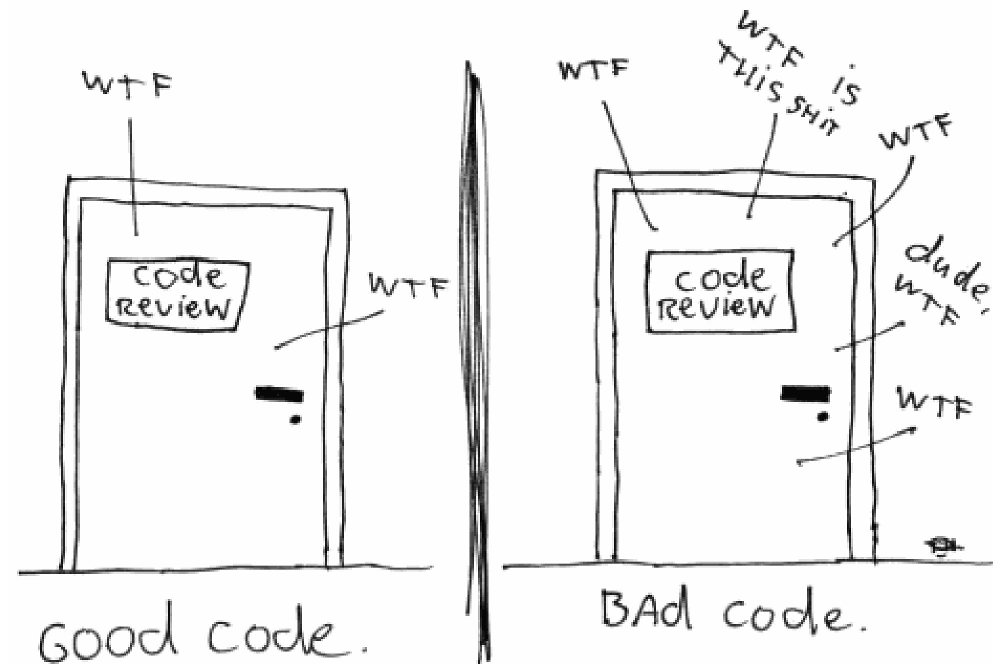
Clean Code

"I like my code to be **elegant and efficient**. the logic should be **straightforward** and make it **hard for bugs to hide**, the **dependencies minimal** to ease maintenance, **error handling** complete according to an articulated strategy, and **performance** close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. clean code does one thing well."

Bjarne Stroustrup, inventor of C++:



Robert C. Martin



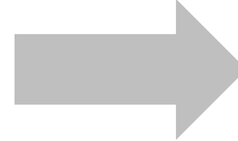
How to write clean code?

- How to organize your code/repository?
- How to find bugs in your code?
- How to format your code?
- How to automate your development workflow?

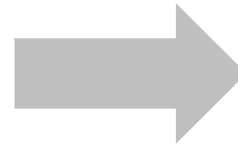
How to write clean code?

- How to organize your code/repository?

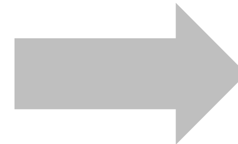
- How to find bugs in your code?



- How to format your code?



- How to automate your development workflow?



GNU Make

Survey - Who has worked/ works with what?

- 0: Python
- 1: Cookiecutter
- 2: mypy (type checking in general)
- 3: Black autoformatting
- 4: make files

Cookiecutter – Create projects from project templates

Overview

A simple command-line tool that creates projects from project templates

- for e.g. Python package projects or Data Science project directories
- Install it with pip
- *cookiecutter-pypackage* : Cookiecutter template for a Python package
- *cookiecutter-data-science*: A logical, reasonably standardized, but flexible project structure for doing and sharing data science work

Advantages

- Use predefined templates with best practice project structures
- Get consistency throughout your repositories
- Reduce the amount of boiler plate code you have to write (e.g. logger, evaluation, etc.)

https://github.com/drivendata/cookiecutter-data-science

https://github.com/audreyfeldroy/cookiecutter-pypackage

Source: https://github.com/topics/cookiecutter-template

mypy - Optional Static Typing for Python

: my[py]

Overview

Mypy is an optional static type checker for Python.

- Static type checking (compiler) vs. dynamic type (Python interpreter)
- You can add type hints (since PEP 484, Python version 3.5) to your Python programs, and use mypy to type check them statically
- You can mix dynamic and static typing in your programs
- Install it with pip

Advantages

- Static typing makes it easier to find bugs with less debugging
- Type declarations act as machine-checked documentation. Static typing makes your code easier to understand and easier to modify without introducing bugs
- You can develop programs with dynamic typing and add static typing after your code has matured, or migrate existing Python code to static typing

mypy - Optional Static Typing for Python

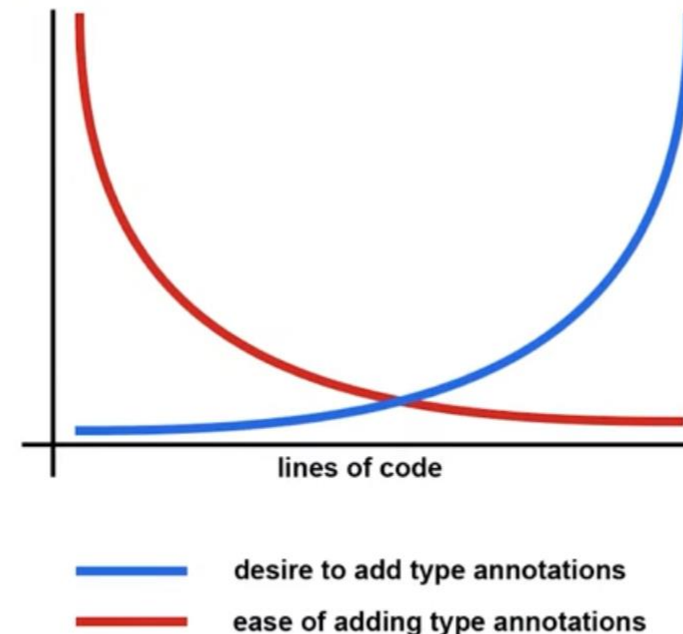
: my[py]

Mypy with static typing

```
class BankAccount:
    def __init__(self, initial_balance: int = 0) -> None:
        self.balance = initial_balance
    def deposit(self, amount: int) -> None:
        self.balance += amount
    def withdraw(self, amount: int) -> None:
        self.balance -= amount
    def overdrawn(self) -> bool:
        return self.balance < 0

my_account = BankAccount(15)
my_account.withdraw(5)
print(my_account.balance)
```

```
$ mypy program.py
```



https://www.youtube.com/watch?v=ST33zDM9vOE

Similar tools: Pytype

Black - The Uncompromising Code Formatter

Overview

- Black is a PEP 8 (Style Guide for Python Code) compliant formatter
- Black reformats entire files in place
- It is not configurable, It doesn't take previous formatting into account
- It has a predefined code style
- Install it with pip

Advantages

- Black provides you with consistent and fast formatting
- Save time and energy for more important matters
- Consistent code format
- Formatting becomes transparent

Black's opinions, to go the_black_code_style:

https://github.com/psf/black/blob/master/docs/the_black_code_style.md

Black - The Uncompromising Code Formatter

```
black {source_file_or_directory}
```

```
# in:

def very_important_function(template: str, *variables, file: os.PathLike, engine: str, header: bool = True, debug: bool = False):
    """Applies `variables` to the `template` and writes to `file`."""
    with open(file, 'w') as f:
        ...

# out:

def very_important_function(
    template: str,
    *variables,
    file: os.PathLike,
    engine: str,
    header: bool = True,
    debug: bool = False,
):
    """Applies `variables` to the `template` and writes to `file`."""
    with open(file, "w") as f:
        ...
```

Similar tools: yapf

https://github.com/psf/black/blob/master/docs/the_black_code_style.md

GNU Make and *makefile*

Overview

- GNU Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files
- Store and automate shell commands
- Can be used with almost any technology or language

```
helloword: helloword.c
    gcc -o hellword hellword.c -I.
```

Advantages

- makefile and make can make your life much easier
- Automate most of your building and testing
- Clean up your repo, run test with coverage, use autoformatting
- Run application in docker container
- Manage your environment

https://krzysztofzuraw.com/blog/2016/makefiles-in-python-projects

https://www.gnu.org/software/make/manual/make.html

https://www.gnu.org/software/make/

GNU Make and *makefile* - example

```
1   .DEFAULT: env
2
3
4   init:
5       pip install -r requirements.txt
6       python -m spacy download en_core_web_sm
7
8   freeze:
9       pip freeze > requirements.txt
10
11  black:
12      python -m black src/model
13
14  black_diff:
15      black src/model --color --diff
16
17  type_checking:
18      mypy src/model/ModelClass.py
19      mypy src/model/LoggerClass.py
20
21
22  setup:
23      conda env create -f environment.yml
24
25  env:
26      conda activate explanation_env
27
28  export_env:
29      conda env export > environment.yml
30
```

```
1   HOST=127.0.0.1
2   TEST_PATH=./
3
4   clean-build:
5       rm --force --recursive build/
6       rm --force --recursive dist/
7       rm --force --recursive *.egg-info
8
9
10  test: clean-build
11      py.test --verbose --color=yes $(TEST_PATH)
12
13  run:
14      python manage.py runserver
15
16  docker-run:
17      docker build \
18        --file=./Dockerfile \
19        --tag=my_project ./
20      docker run \
21        --detach=false \
22        --name=my_project \
23        --publish=$(HOST):8080 \
24        my_project
```

```
C:\Users\maurol>make init
```

Conclusion

- Python has a huge ecosystem of tools to help you write code
- Embrace best practices from other developers, e.g. testing, type checking, documentation
- Practice consistency
- Favor readability
- Keep it simple
- Don't Repeat Yourself

Thank you!

Discussion

- Are you publishing your code with your research?
- Are you publishing your datasets?
- Do you create a common and shared code based to reuse existing code written for research?
- Do you share your developped tools as libraries?
- Is code quality ever a concern for you?
- Do you colleboratively work on code?

Cookiecutter - example

cookiecutter-data-science: https://github.com/drivendata/cookiecutter-data-science

```
cookiecutter -c v1 https://github.com/drivendata/cookiecutter-data-science
```

```
├── LICENSE
├── Makefile           <- Makefile with commands like `make data` or `make train`
├── README.md          <- The top-level README for developers using this project.
├── data
│   ├── external       <- Data from third party sources.
│   ├── interim        <- Intermediate data that has been transformed.
│   ├── processed      <- The final, canonical data sets for modeling.
│   └── raw            <- The original, immutable data dump.
│
├── docs               <- A default Sphinx project; see sphinx-doc.org for details
│
├── models             <- Trained and serialized models, model predictions, or model summaries
│
├── notebooks          <- Jupyter notebooks. Naming convention is a number (for ordering),
│                         the creator's initials, and a short `-` delimited description, e.g.
│                         `1.0-jqp-initial-data-exploration`.
│
├── references         <- Data dictionaries, manuals, and all other explanatory materials.
│
├── reports            <- Generated analysis as HTML, PDF, LaTeX, etc.
│   └── figures        <- Generated graphics and figures to be used in reporting
│
├── requirements.txt   <- The requirements file for reproducing the analysis environment, e.g.
│                         generated with `pip freeze > requirements.txt`
│
├── setup.py           <- makes project pip installable (pip install -e .) so src can be imported
├── src                <- Source code for use in this project.
│   ├── __init__.py    <- Makes src a Python module
│   │
│   ├── data           <- Scripts to download or generate data
│   │   └── make_dataset.py
│   │
│   ├── features       <- Scripts to turn raw data into features for modeling
│   │   └── build_features.py
│   │
│   ├── models         <- Scripts to train models and then use trained models to make
│   │   │                 predictions
│   │   ├── predict_model.py
│   │   └── train_model.py
│   │
│   └── visualization  <- Scripts to create exploratory and results oriented visualizations
│       └── visualize.py
│
└── tox.ini            <- tox file with settings for running tox; see tox.readthedocs.io
```

Black - Python code formatter

```
Usage: black [OPTIONS] [SRC]...

  The uncompromising code formatter.

Options:
  -c, --code TEXT                 Format the code passed in as a string.
  -l, --line-length INTEGER       How many characters per line to allow.
                                  [default: 88]

  -S, --skip-string-normalization Don't normalize string quotes or prefixes.
  -C, --skip-magic-trailing-comma Don't use trailing commas as a reason to
                                  split lines.

  --diff                          Don't write the files back, just output a
                                  diff for each file on stdout.

  --color / --no-color            Show colored diff. Only applies when
                                  `--diff` is given.

  --fast / --safe                 If --fast given, skip temporary sanity
                                  checks. [default: --safe]

  --config FILE                   Read configuration from FILE path.
```