Good practices for Python development

What you should know before doing anything

ZWE Software Workshop

Max-Planck-Institut für Intelligente Systeme

MPRIS Workshop, 28 October, 2021

Outline

Outline

Where do I start?

You start a new project. What do you need to do and what should be there at the bare minimum.

Minimal structure

```
project
├── doc
├── package
│   ├── __init__.py
│   ├── version.py
│   └── ...
├── tests
├── LICENSE.md
├── README.md
├── requirements.txt
└── setup.py
```

README.md

This is the first thing people read. `github`, `gitlab`, etc. display this on the project page. It should contain:

- project name
- project description
- requirements (but not python dependencies)
- installation instructions
- maybe usage examples

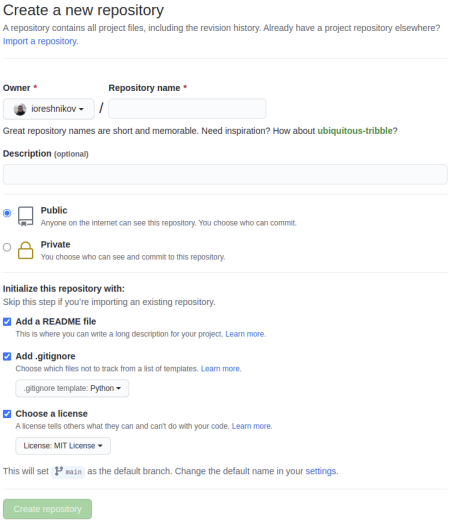
LICENCE

This license explains what people can do with your code. Pick the one you see fit (MIT, BSD, etc) and just copy it in your repo.

.gitignore

(not pictured in the previous slide)
This file lists all the file *patterns* `git` should ignore. It's a good idea to put in there all the `__pycache__` and `*.pyc` and `*.so` and `.idea` and so on.
If you don't know what to put in there, `github` offers you a truly all-encompasing file when creating a repo.

Github offers those three for you



Requirements file

The `requirements.txt` file contains the list of the dependencies of your project that can be installed via `pip`:

Contents of requirements file

```
numpy
scipy
matplotlib==3.1.1
mock>=3.0
pywin32; sys_platform == 'win32'
```

Usage:

Commands

```
$ pip install -r requirements.txt
```

setup.py

The `setup.py` file describe your project and allows you to build/distribute it:

Contents of setup file

```
from setuptools import setup
setup(name=package_name, version=1.0, \
install_requires=['numpy'], zip_safe=False)
```

Usage:

Commands

```
$ pip install .  # install as a package
$ pip install -e .  # for development
$ python setup.py sdist # build a source distribution
$ python setup.py build_ext bdist_wheel # build a wheel
```

Example project

We have set up a minimal (more or less) real-world example you can use as a reference

github.com/MPI-IS/py-project-example

What about the data?

- If it's small and you have no licensing problems you might as well put it in the repository, for example in a separate directory named `data/`.

- If you don't want to commit it, it's still a good idea to put it in the `data/` directory. How not to commit by accident: create an empty file `data/.keep`, commit *this file only*, put the entire content of `data/*` in `.gitignore` and then copy the data in that directory.

What about the scripts?

Where to put the scripts, for example to train and evaluate the model?

- create a `scripts/` directory and put your scripts there
- add them to a `scripts=[]` section in `setup.py`
- after `pip install -e .` they should be visible in the environment *globally*, i.e. independent from the working directory. The imports will also work.

Questions so far?

Outline

The problem

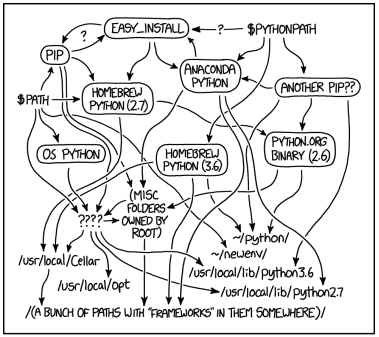
- By default, you are using a single Python interpreter for all your projects.
- If you install your packages with the corresponding `pip`, there is a high chance you will **create conflicts** between your projects

Solutions: Edit your `PYTHONPATH`, `.bashrc`, `.profile`, brew python...

The problem

- By default, you are using a single Python interpreter for all your projects.
- If you install your packages with the corresponding `pip`, there is a high chance you will **create conflicts** between your projects

Solutions: -Edit your `PYTHONPATH`, `.bashrc`, `.profile`, brew python...



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

Virtual environments

The **correct solution** is to use virtual environments.
A **virtual environment** is a **local** Python environment that is **isolated** from other virtual environments and (by default) from the system Python. Advantages:

- no sudo rights needed
- it has its own `python`, `pip`...
- whatever you install stays in the virtual environment
- if something goes wrong, delete the folder and create a new one

Usage:

Commands

```
~/Code/my-project$ python3 -m venv --copies my_venv
~/Code/my-project$ .  my_venv\bin\activate
(my_venv) ~/Code/my-project $ which python
(my_venv) ~/Code/my-project $ deactivate
```

A convinience: pipenv

If this one seems clunky, use Pipenv! It allows you to create and activate the environment based on your working directory.

Commands

```
~/Code/my-project$ pipenv install
~/Code/my-project$ pipenv shell
(my_venv) ~/Code/my-project $ which python
(my_venv) ~/Code/my-project $ deactivate
```

If you must: conda

`Conda` is a distribution and a package manager and an environment manager wrapped in one. It's really popular among data scientists and machine learning researchers and egineers.

We personally do not use it. Sometimes with `conda` things break in a strange way. But if you have to (need a very tricky set of exactly versioned dependencies, difficult installation with many third-parties, you are on Windows, etc.), then please do.

Questions so far?

Outline

Code can look differently

The same problem can be solved in many different ways. The algorithms might be different. But even the way the same algorithm *looks* might be also different.

Examples

- How do we name variables? And how do we spell the names? `like_so` or `MaybeLikeSo`?
- How do we name functions? Is it `is_even(number)` or `even(number)`? And again, why not `IsEven(number)`?
- Where do we put braces? How do we break long lines? What's a long line anyway? Tabs or whitespaces?
- Sometimes it's technical, e.g. do we throw exceptions?

Ideally we want the code to look the same

If there are several people working on the project it is good idea to make the code at least look the same way.

- consistency — there should be few surprises when switching from one part of the project to another, because
- readability — once you have a consistent style you can get used to it and the code will be easier to read and as a result
- comprehension — you will understand more

Code style

The way engineers do it is by introducing a *code style*. Agree on the common way to write the code and try to keep it this way. The scope can be different

- almost always there is a code style specific to the project. See Linux Kernel Style Guide (for C).
- sometimes there is a company-wide style. See Google Style Guide (for C++).
- sometimes there is an official style guide for a specific language. This usually also means that one can write tools that help to keep the style.
- the most radical: there is a *very opinionated* official style guide and there are powerful out-of-the-box tools doing the job for you. See go fmt and rustfmt.

PEP8

Python has an official style guide known as PEP8.

(interlude) What's a PEP

PEP is a python enhancement proposal. A few examples:

- PEP0: Index of PEPs
- PEP20: The Zen of Python
- PEP257: Docstring Conventions
- PEP465: A dedicated infix operator for matrix multiplication

PEP8

Python has an official style guide known as PEP8. It is 10 to 20 pages long and it illustrates how to write and how not to write python code.

Tools to help you

There are some tools to help you with code style:

- standalone style checkers such as flake8
- formatters such black as autopep8
- if you want to be strict/annoying, then such checker/formatters can be used as a githook
- most IDEs have plugins that use those tools automatically. PyCharm does this out-of-the-box. VSCode/Atom do that when you install a python extension. There are ways to integrate this into VIM/Emacs.

This is only about the style.

There are code analyzers that are a bit more advanced, e.g. pylint or radon.

There is no silver bullet

This is the bare minimum, but it does not guarantee your code any good.

- tools do not help with the overall sanity of your naming scheme.
- tools do not help you to structure your code into functions/classes/modules either.
- tools cannot guarantee that the code is straightforward enough.
- in the end you can write a good code but it doesn't look *idiomatic*!

It is difficult to define what a good code is.

It is a skill

Writing good code is a skill. It can be learned and it can be taught. Mostly by example.

- there are books. Python-specific — a (free online, a bit chaotic) Chapter on writing great python code in The Hitchhiker's Guide to Python. In general — Clean Code: A Handbook of Agile Software Craftsmanship.
- read good code
- do *code reviews*, i.e. write bad code, show it to other people and ask for their opinion

We do code reviews as a service. If you have a project you want to publish and you are not sure whether it is good or how to improve it then we are happy to help. Caveat: we might be slow, so write us in advance.

Questions so far?