GUIDELINES FOR PROJECT DEVELOPMENT IN PYTHON

Jacques Bourg

December 14, 2024

ABSTRACT

In this document, I outline some best practices for Python project development, such as file structure, unit tests, code quality, and package creation.

Keywords Python, project development, modules

1 Project structure

1.1 Tree structure

```
data_science_project
├── README.md
├── .gitignore
├── environment.yml
├── main.py
├── src
│   ├── __init__.py
│   ├── utils
│   └── models
├── tests
│   ├── test_utils.py
│   └── __init__.py
├── notebooks
│   └── algo_v1.ipynb
├── data
│   ├── raw
│   │   └── dataset_1.csv
│   └── processed
│       └── output.csv
├── models
│   └── mini_vgg16.pth
└── docs
```

In bash, in order to see the tree like structure do in the root folder:

```
> tree
```

To install the command tree:

```
> sudo apt install tree
```

1.2 Readme

The readme is an essential part of a project. In order to be well written write a prominent heading with the name, followed by a concise description of the project. Write a table of contents. Explain step by step how to install and set up the project including code snippets: include dependencies, virtual environments. Then provide instructions on how to run the project, basic commands to start, and a reference for documentation and jupyter notebooks of usage cases. Finally, specify the license under which your code is distributed. A nice way to format the code, is to write it in markdown.

1.3 .gitignore file

A good practice when developing a project is to exclude temporary files, generated data, and large datasets from git tracking. Here is a non exhaustive list of files and folders that one usually puts into the .gitignore file .

.*, *.pyc, *.lib, __pycache__/, .pytest_cache/, .ipynb_checkpoints/, *.h5, *.hdf5, *.pth, *.pt, *.tf, *.keras, *.onnx, *.pb, *.pb.gz, *.json, *.yaml, *.cfg, *.log, data/, docs/,

1.4 Conda environment

To create a virtual environment, first install miniconda, and then run in bash:

```
> conda create -n env_project python=3.10
```

To activate your environment:

```
> conda activate env_project
```

To install numpy in the newly created environment:

```
(env_project)> conda install anaconda::numpy
```

Once you installed all the dependencies you need, you can generate the environment.yml file using the command

```
conda env_project export > environment.yml
```

To recreate the environment env_project:

```
conda env create -f environment.yml
```

1.5 Modules

The src file is the core of the application. It has to be organized in modules. To create a module 'algorithms', create a folder of the same name, create a __init__.py file (can be left empty). These files are used to indicate to python that a directory should be treated as a package, and then allow the module importation using the dot notation. For instance, to import a function plot_pairs in main.py:

```python
from utils.visualisation import plot_pairs
```

The init file is run only once, at the importation of the module, so that you can insert in this file imports (for instance importing a class that you are going to use later).

```python
from .utils import PlotTool
```

After having done this last importation in the init file, I can now import directly in the main:

```python
from src import PlotTool
```

1.6 Unitary tests

In VSCode, in your environment, choose a testing framework, with VIEW /COMMAND PALETTE/ PYTEST, and choose the folder named 'tests'. It will install pytest in your environment. Be sure to have an init file in your test folder. The tests should be non trivial, and organized according to the functionality they cover, always the syntax $test\_X$, in which X is the function you want to test.

```python
from src.models.operations import Operations
def test_calculate_average():
   assert Operations.calculate_average([2, 2, 6, 6]) == 4
```

The tests should aim at covering all the code. One good practice is to build data from scratch and plug it to the functionality we want to test, in order to ensure independence of the tests. Before a merge request one runs all the tests and is sure that all the tests pass and that no functionality was broken.

1.7 Jupyter notebooks

1.7.1 Adding a kernel to jupyter notebook

In order to develop your algorithms in jupyter notebook, install ipykernel in your conda environment called env_project.

```
> conda activate env_project
> conda install ipykernel
```

Then register the kernel with jupyter.

```
> python -m ipykernel install --user --name=env_project
```

1.7.2 Function development

I use to develop my code at the end of a jupyter notebook, in cells to check that all the variables do what I want them to do. Whem I am satisfied about the result, I encapsulate them in functions, that I transfer to the modules in the src, and we can now call them in the pipeline developed in the jupyter notebook. To avoid having to restart the kernel, and having to reload all the cells again (which can be very tedious), add the following command before the imports (be sure ipython is installed in your environment). In this way, the changes in the code in the src will be reflected seamlessly in the notebook.

```
%load_ext autoreload
%autoreload 2
```

2 Debugging

The debugging can be configured in VScode by changing the hidden file .vscode/launch.json. For instance:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Python: Current File",
            "type": "debugpy",
            "request": "launch",
            "program": "${file}"
        }
    ]
}
```

To start debugging in VScode, put a breakpoint in your main code for instance, select the right environment (bottom right of VScode) and then choose the top right play button choosing the option "Python Debugger file: Debug Python File".

3 Code quality

In order to assure a good code quality, one should learn to write in a way that is standard and that can be read by any developer. Python has a series of guidelines, called Pep-8, with the objective of having clean and maintainable code.

Linters are tools that analyse code to identify and correct stylistic issues. Ruff is one of them. Once installed in your environment, format your documents using ctrl + shift + i. Also a good practice is to put docstrings. The following is a NumPy-style docstring:

```
def calculate_average(number:list):
  """Calculates the average of a list of numbers.

  Args:
    numbers: A list of numbers.

  Returns:
    av: float, average of the numbers.
  """
```

In reStructuredText, the docstring is like:

```
def calculate_average(number_list:list):
        """
        Calculates the average of a list of numbers.

        :param number_list: A list of numbers.
        :type number_list: list of float
        :return: The average of the numbers.
        :rtype: float
        """
```

One can also check the type of the inputs. This can be done trough methods such as check_array (available in scikit-learn and scikit-image).

```
check_array(
        image,
        ndim=3,
        dtype=[np.uint8, np.uint16, np.int32, np.int64,
               np.float32, np.float64])
```

When the input parameter can only take a given set of values, we can also verify all the cases:

```
if method == "median":
        projected_image = np.nanmedian(in_focus_image, axis=0)
    elif method == "max":
        projected_image = np.nanmax(in_focus_image, axis=0)
    else:
        raise ValueError("Parameter 'method' should be 'median' or 'max', not "
                         "'{0}'.".format(method))
```

There are important organizational principles: a good practice is to write one class (in PascalCase) per python file (in lower case with underscores). Functions also have to be simple: one function does only one thing. Functions with similar functionalities should be grouped together. Helper functions should go to the utils module. Another important aspect to have a clear code is to organize the imports: third party libraries first, built imports (imports that are specific to python such as os, sys...) and then import local files and relative imports.

4 Documentation

There are several tools like Mkdocs and Sphinx to generate automatic documentation of your code based on the docstrings. For instance, Sphinx imposes to use certain synthax like ReStructuredText or markdown format for writing documentation, so that headers, lists, and code blocks appear in a nice looking way.

Some quick guidelines for generating the documentation in Sphinx. In your environment, install sphinx.

```
> conda activate env_project
> conda install conda-forge::sphinx
```

Place yourself in the doc folder.

```
(env_project)> cd docs
```

Run sphinx:

```
(env_project)> sphinx-quickstart
```

Install the sphinx_rtd_theme:

```
(env_project)> pip install sphinx_rtd_theme
```

Configure the config file:

```
import os
import sys
sys.path.insert(0, os.path.abspath('../src'))

import sphinx_rtd_theme
extensions = ['sphinx.ext.autodoc', 'sphinx.ext.napoleon', 'sphinx_rtd_theme',]
html_theme = 'sphinx_rtd_theme
```

Configure the index file:

```
.. toctree::
   :maxdepth: 2
   :caption: Contents:


Modules
=======

.. automodule:: models.operations
    :members:
    :undoc-members:
    :show-inheritance:

.. automodule:: utils
    :members:
    :undoc-members:
    :show-inheritance:

Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

Finally, generate the documentation:

```
(env_project)> make html
```

The documentation will be located in docs/_build/html. This file usually is not tracked in git.

5 Package creation

Once the code is ready, install setuptools in your environment, and then create a setup.py file:

```
from setuptools import setup, find_packages


setup(
    name="package_data_science_project",
    version="0.1.0",
    description="Test package",
    author="JB",
    author_email="your_email@example.com",
    packages=find_packages(),
    install_requires=["numpy"]
)
```

Then install your package:

```
> python setup.py develop
```

and finally create a distribution:

```
> python setup.py sdist bdist_wheel
```

This will generate additional folders: build, dist, package_data_science.