fakeproj

Release 1.0rc

Anurag Saha Roy

Aug 31, 2021

CONTENTS:

This documentation is intended to serve as a self-paced tutorial to familiarise Python developers with various tools that can help them develop, maintain and publish better code while making the experience a lot more enjoyable. The repository contains sample files that demonstrate all the functionality that we discuss in this tutorial. Ideally, one should start from an empty repository and follow along the various sections below to write code, documentation & tests and then configure the build and deployment systems, referring to the sample code whenever necessary. Originally intended for scientific software developers coming from a non Software Engineering background, the tutorial mainly focuses on developing python libraries (as opposed to webapps). However, much of the information here is applicable and transferable to other domains of software development as well.

What this tutorial isn't, is an in-depth treatment of the various tools we describe. Almost always, we will provide a brief introduction and give a taste of what all can be done using a particular tool and then include links to documentation and tutorials that do a better job in giving you a step-by-step guide to using said tools.

CHAPTER

ONE

STRUCTURING YOUR PROJECT

Structuring the project repository into useful directories is an essential first step in ensuring that your code stays organised and new contributors or users of your repository can intuitively know where to look for what.

1.1 Code - Test - Docs

Most projects have 4 major parts:

1. Source Code
2. Tests and Benchmarks
3. Documentation
4. Configuration Files

The typical repository structuring involves keeping all your source code in a `src/` or `project_name/` and all your documentation in a `docs/` directory. The tests can either reside along side the code or be in a separate `test/` directory in the repository root. For the tutorial repository fakeproj, we choose the latter, as below:

```
fakeproj/
├── docs
│   └── source
├── fakeproj
│   ├── fakedir
│   └── gooddir
└── test
```

The Qiskit repository follows a similar structure:

```
qiskit-terra/
├── docs
.
.
.
│   └── source_images
├── examples
│   ├── python
│   └── qasm
├── qiskit
│   ├── assembler
.
.
.
```

(continues on next page)

(continued from previous page)

```
│   └── visualization
├── releasenotes
│   └── notes
├── test
│   ├── ipynb
│   ├── python
│   └── randomized
└── tools
```

However, tests in Numpy reside closer to the code being tested:

```
numpy/numpy
├── compat
│   └── tests
├── core
│   ├── code_generators
│   ├── include
│   ├── src
│   └── tests
.
.
.
├── fft
│   └── tests
├── lib
│   └── tests
.
.
.
└── typing
    └── tests
```

This is typically a matter of style and policy. The general practice is if you wish your tests to be installed as a part of your library, you keep them closer to your code while if the tests are only meant to be used for development, you keep them in a separate directory.

1.2 Packages, Modules and Imports

When working on a large project, you typically want to convert your python scripts to modules and packages that can be imported and reused elsewhere in the code. For the tutorial repository this looks as below:

```
fakeproj/fakeproj/
├── __init__.py
├── fakedir
│   ├── __init__.py
│   └── fakemodule.py
└── gooddir
    ├── __init__.py
    ├── goodmodule.py
    └── sysmodule.py
```

The `__init__.py` is the magic ingredient that allows you to convert your Python modules into packages that can now be imported and used from other directories. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable. The Python Packages Documentation describes this in further details.

A related topic that often confuses new developers working with packages and modules in Python is how the interpretor deals with Imports - both relative and absolute. We list below some useful resources that address the usual confusions:

- Relative imports for the billionth time - Stackoverflow
- Python PEP 328: import and build package

1.3 Configuration Files

There will always be various configuration files in your repository that interact with the services you use to maintain your code. These are typically stored in the root of the directory because that is where most services will look for these files (unless otherwise specified). Some of these files can also be combined in the form of `.toml` files, but it is usually advisable to have separate configuration files for the different services used. We list some common ones below:

- `.gitignore` - Files you don't want `git` to track (Useful gitignores)
- `.pre-commit-config.yaml` - *Pre Commit Hooks*
- `pytest.ini` or `conftest.py` - *Tests and Code Coverage*
- `.github/` - *Automating Workflows and CI/CD*

1.4 README

The `README.md` is essentially the front page to your repository and it is imperative that you provide a concise and useful introduction to your project while providing instructions for using and/or contributing to the code base. Below are some of the useful features that are nice to have in an Awesome README :

- Clear description of what the project does
- Table of Contents for easy navigation
- Step-by-Step installation and setup sections
- Overview of features
- Instructions for Contribution
- Demo screenshot or GIF
- Code snippets demonstrating common features/functionality
- API Overview where applicable
- Badges for stats
- FAQ for common usage and troubleshooting points
- Instructions on filing bugs/feature requests and getting support
- List of Alternatives
- Link to Documentation
- Link to Project Website
- Bibtext for citing the project
- References for further reading
- Contact details - Email or Mailing List or Gitter/Slack

CHAPTER

TWO

DEPENDENCY MANAGEMENT

CHAPTER

THREE

CODE DOCUMENTATION

3.1 Docstrings

3.2 Docstring coverage

3.3 Sphinx

3.4 ReadTheDocs

CHAPTER

FOUR

TYPE HINTING IN PYTHON

4.1 Type Hints for Primitive Type

4.2 Typing Module in Python

4.3 Type Hints for 3rd Party Libraries

4.4 MonkeyType and MyPy

4.4.1 Related

CHAPTER

FIVE

TESTS AND CODE COVERAGE

5.1 Testing with pytest

5.2 Code Coverage with pytest-cov and Codecov.io

CHAPTER

SIX

CODE COMPLEXITY

6.1 Measuring Complexity with radon

6.2 Monitoring Complexity with xenon

CHAPTER

SEVEN

CODE FORMATTING AND LINTING

7.1 Code Formatting with black

7.2 Code Linting with flake8

7.3 Best Practices

7.3.1 Related

CHAPTER

EIGHT

COLLABORATION WITH GIT

CHAPTER

NINE

PRE COMMIT HOOKS

9.1 Pre-Commit Config files

9.2 Black & Flake8 in precommit

9.3 MyPy in precommit

CHAPTER

TEN

AUTOMATING WORKFLOWS AND CI/CD

10.1 Continuous Integration

10.2 Continuous Deployment

10.3 Automating with Github Actions

CHAPTER

ELEVEN

PACKAGING AND PUBLISHING

11.1 setup.py for Packaging

11.2 Publishing to PyPi

11.3 Publishing to Conda Forge

CHAPTER

TWELVE

MAKING DEVELOPMENT ENJOYABLE

12.1 Why do you need an IDE?

12.2 An Opinionated Take on IDEs

12.3 Useful Plugins

CHAPTER

THIRTEEN

INDICES AND SEARCH

- genindex
- modindex
- search