Python-Project-Skeleton

Release 0.3.1

Joao MC Teixeira

Jan 22, 2021

CONTENTS

CHAPTER

ONE

PYTHON PACKAGE SKELETON TEMPLATE

1.1 Summary

This is a **project skeleton template** for a **Python project/library**. This repository implements and explains the latest practices in team software development and deployment within a continuous integration framework. **Note** that is impossible for me to cover all strategies available in the wild. This repository covers the needs of my Python projects, which include:

- **a robust Python library/application file hierarchy with packages, modules, clients**
  - detailed, yet simple, `setup.py`
  - the special use of the `src` directory
  - examples of Python command-line interfaces
- **unique testing framework for developers with tox and pytest**
  - assures tests are reproducible across developers platforms
  - assures same lint rules are always applied
  - assures all desired Python versions are covered
- **continuous integration with GitHub Actions**
  - automatic testing on Linux, MacOS, and Windows
  - automatic testing upon deployment with tox
  - test coverage report to Codecov
  - automatic version bump with bump2version
  - automatic git tagging and Python packaging to PyPI

1.2 Motivation

To understand and implement in the best practices in software development and deployment for scientific software. Actually, I believe the strategy reviewed here can be applied to most Python library projects.

This repository does **not** intent to be a cookiecutter-like repository. Though there are many and very well documented cookiecutter templates out there, even for scientific software, when I initiated my adventure in developing Python libraries I decided that using a cookiecutter would lead me to nowhere because I would miss what was actually being automatized. Hence, assembling this *template* repository from scratch was the only and best approach to achieve a minimum understanding of the best practices and protocols on the matter. Now, this repository serves as a reference guide for all my projects and I try to keep it up to date to my needs and changes in the CI ecosystem.

1.3 Acknowledgments

The Python library organization itself was strongly influenced by ionel discussions in his blog post about *Packaging a python library*. I really recommend reading through that post and the related posts in his blog.

I setup the CI pipeline with bits from many places. Kudos to python-nameless and cookiecutter-pylibrary two repositories that served as main source of information for the *python-project-skeleton* repository, specially in the first versions with Travis and Appveyor.

When migrating to GitHub Actions, I want to thank @JoaoRodrigues for the workflows in pdb-tools, `ymyzk` for the tox-gh-actions package, and structlog, which was also a repository I used as a reference to build test latest version here.

I reference other important sources of information as comments in the specific files. Thanks everyone for keeping discussions out there open.

1.4 How to use this repository

The repository simulates the implementation of a `sampleproject`. Here, `sampleproject` is the Python name of your project, that which will be *import sampleproject*. So everywhere you find `sampleproject` just replace with the name of your project.

In `setup.py` the project has the name `jmct-sampleproject` because `sampleprojet` was already in use in test.pypi.org, as expected. Substitute that by the name of you package. Normally, it as the same name as `sampleproject`.

You will find in the project's documentation all references that motivated the current configuration as well as detailed explanation on the different configuration files.

I intent to keep this repository up to date to my knowledge and needs. Your feedback and suggestions are highly appreciated, please raise an issue and share your thoughts.

1.5 Version

v0.3.1

CHAPTER

TWO

REPOSITORY CONFIGURATION

This page explains how this template repository is organized by detailing the building blocks of the project skeleton.

2.1 Branch organization

Two main branches set the development workflow: the master branch and the latest branch. The latest branch is thought to evolve according to continuous integration practices, and is referred as the *latest* build or version; while, on the other hand, the *master branch* is considered the *stable* or *production* version. Under this configuration the *master* branch receives updates from the *latest* build periodically or when a new version/patch is ready for deployment. Read further about *Branch workflow*.

2.2 Project Layout

The project layout follows the `src`, `tests`, `docs` and `devtools` folders layout.

2.2.1 The src layout

I have discovered that storing the source library folder under a `src` directory instead of directly in the project's root is by far the most controversial point out there on the wild Internet. Here I adopted the `src`-based layout discussed by ionel in his blog post. When reading through the discussed arguments, I found this strategy to have many advantages over the common root directory layout and no added disadvantage.

In detail, though encapsulating the source in a `src` directory is an uncommon practice in Python projects, adopting this practice avoids unexpected and uncontrolled code imports that could lead to wrong testing operations, as well stated by ionel, see his src-nosrc example. On the same hand, encapsulating the source under a `src` directory does not create any additional problems that would be avoided by the *standard* layout with source directly on a rootdir-based folder, usually named the same as the package name.

2.2.2 tests

Tests are nicely encapsulated in a `tests` folder. With this encapsulation, outside the library folder, it is easier to control that tests do not import from relative paths and can only access the library code after library installation (whatever the installation mode is). Also, having `tests` in a separated folder facilitates the configuration files layout on excluding tests from deployment (`MANIFEST.in`) and code quality (`.codacy.yaml`) or coverage (`.coveragerc`).

2.2.3 docs

All documentation related files are stored in a `docs` folder. These include files related to the library documentation but also with the development process, such as: `AUTHORS`, `CONTRIBUTING`, `CHANGELOG`, etc.

2.2.4 devtools

The `devtools` folder hosts the files related to development. Here I used the idea explained by Chodera Lab in their structuring your project guidelines, though for other issues described previously, I do not follow their guides, as explained in context.

2.3 CI Platforms

Here we provide an overview of the implementation strategies for the different continuous integration and quality report platforms. We have adopted a total of seven platforms, two for building and testing, two for code quality control, two for test coverage and one for documentation deployment:

1. **Building and testing**
    - Travis-CI (Linux and OSX)
    - Appveyor (Windows)
2. **Quality Control**
    - Codacy
    - Code Climate
3. **Test Coverage**
    - Codecov
    - Coveralls
4. **Documentation**
    - Read the Docs

We acknowledge the existence of many other platforms for the same purposes. Though, we have chosen these because they fit the size and scope of the projects to which this template aims at and are those platforms most used within our field of development.

2.3.1 Choosing the CIs

Please note that you do not need to use all these platforms when adapting this template for your project, we do suggest you use at least one for each topic. For example, you do not need to activate Appveyor if you do not intent to deploy/distribute your code for Windows machines. Also, for quality control and test coverage one of the two provided options may suffice, however, having both is free and you can benefit from the different analysis reports the platforms provide.

Note: To **NOT** use a specific CI platform simply do not activate it in their website, remove the configuration file from the root directory of the project, and remove the badge image link from the `README.rst` file. Continue reading to understand better these concepts.

2.3.2 Activate CI

To activate the different CI platforms for you repository just navigate to their website, login with your GitHub account and activate the repository. The configurations provided in this template should to the rest automatically `:-)`, just start pushing your commits to the server.

2.3.3 Travis-CI

The configuration for Travis-CI is defined in the `.travis.yml` file.

Overall, the Travis configuration defines how to execute the different *tox environments* defined in the `tox.ini` file.

Find in the .travis.yml file the complete explanation for the implementation proposed, here we mirror the file:

Todo: Configure Travis to run OSX tests.

2.3.4 Appveyor

The configuration for AppVeyor-CI is defined in the `.appveyor.yml` file.

Contrary to our configuration for *Travis-CI*, with Appveyor, the configuration file simply attempts to build the package and run the unittests battery in the different Python versions.

Find in the *.appveyor.yml_* file the complete explanation for the implementation proposed, here we mirror the file:

2.3.5 Codacy

There is not much to configure for *Codacy* in the version we propose in this template. The only setup provided is to exclude the analysis of test scripts, this configuration is provided by the `.codacy.yaml` file at the root director of the repository. If you wish Codacy to perform quality analysis on your test scripts just remove the file or comment the line. Here we mirror the .codacy.yaml file:

```
---
exclude_paths:
 - 'tests/**'
```

2.3.6 Code Climate

There is not much to configure for Code Climate in the version we propose in this template. The only setup provided is to exclude the analysis of test scripts and other *dev* files Code Climate by default analysis, this configuration is provided by the `.codeclimate.yml` file at the root director of the repository. If you wish Code Climate to perform quality analysis on your test scripts just remove the file or comment the line.

Code Climate provides a **technical debt** percentage that can be retrieved nicely with a *badge<Badges>*

Here we mirror the .codeclimate.yml file:

```
version: "2"         # required to adjust maintainability checks
checks:
  argument-count:
      enable: false
  complex-logic:
    config:
      threshold: 4
  file-lines:
    config:
      threshold: 2000
  method-complexity:
    config:
      threshold: 5
  method-count:
    config:
      threshold: 20
  method-lines:
    config:
      threshold: 25
  nested-control-flow:
    config:
      threshold: 4
  return-statements:
    config:
      threshold: 4
  similar-code:
    config:
      threshold: # language-specific defaults. an override will affect all languages.
  identical-code:
    config:
      threshold: # language-specific defaults. an override will affect all languages.
plugins:
  radon:
    enabled: true
    config:
      threshold: "C"
      python_version: 3
  bandit:
    enabled: true
  sonar-python:
    enabled: true
    config:
      tests_patterns:
        - tests/**
      minimum_critial: major
  editorconfig:
    enabled: false
```

(continues on next page)

(continued from previous page)

```
    config:
      editorconfig: .editorconfig
exclude_patterns:
  - "tests/"
  - ".ci/"
  - "alphas/"
```

2.3.7 Code coverage

Codecov

Codecov is used very frequently to report test coverage rates the software under development. Activate your repository under Codecov as done for any other CI platform. Additional configurations:

- In general settings change the default branch to the `latest` branch, if that is your preferred settings.

Coveralls

Coveralls is also included in this template skeleton. Again, activate the coveralls profile by linking your repository to the server (same as with other CI platforms).

The configuration to Coveralls, `.coveragerc` is the same as of *Codecov*.

Sending coverage reports

Coverage reports are sent to both Codecov and Coveralls servers during the *Travis-tox* `-cover` environment. .travis.yml configuration handles this and you do not need to worry about nothing else.

The options specific to Codecov report (actually coverage package) are described in .coveragerc file, mirrored bellow, description of the configuration file is provided as comments.

```
[paths]
source =
    src
    */site-packages

[run]
branch = true
source =
    sampleproject
parallel = true

[report]
show_missing = true
precision = 2
omit = *migrations*
exclude_lines =
    if __name__ == .__main__.:
```

The `.coveragerc` can be expanded to further restraint coverage analysis, for example adding these lines to the `exclude` tag:

```
[report]
exclude_lines =
    if self.debug:
    pragma: no cover
    raise NotImplementedError
    if __name__ == .__main__.:
```

2.4 Read the Docs

Activate your project at Read the Docs platform (RdD), their web interface is easy enough to follow without further explanations. If your documentation is building under the *tox workflow* it will build in at Read the Docs.

2.4.1 Docs Requirements

Requirements to build the documentation page are listed in `docs/requirements.txt`:

```
sphinx>=2.2
sphinx-py3doc-enhanced-theme
sphinx-argparse
CommonMark
mock
```

In this repository we are using Sphinx as documentation builder and the sphinx-py3doc-enhanced-theme as theme, though you can use many different theme flavors, see Sphinx Themes.

2.4.2 Build version

By default, RdD has two main documentation versions (also called builds): the latest and the stable. The *latest* points to the master branch while the *stable* points to the latest GitHub tag. Therefore, every time changes are pushed to the *master branch* a new build in the latest version of the documentation is created, while the stable version is built only when new tags are created.

However, for many projects it is desirable a different setup where the master branch holds the stable version, that is, the code referent to the latest tag, while another branch (usually named *latest* or *develop*) set to the repositories' default, holds the latest development code that has not yet been merged to the master and considered stable. This is the setup of this template repository. Under this setup, it is desirable that the documentation build referent to the *latest* version points to the latest branch, the *stable* doc build will always point to the latest tag. This can be edited in `Admin` -> `Advanced Settings` and `Default version` and `Default branch`.

2.4.3 Google Analytics

Read the Docs allows straight forward implementation of Google Analytics tracking in the project documentation, just follow their instructions.

2.5 Badges

Badges point to the current status of the different Continuous Integration tools, for example, Travis-CI or Appveyor, but also documentation and code quality reports.

This project has two badge groups, one for the master (stable) branch and other for the latest (develop) branch. By showing information for these two groups the development team can keep track on the improvements (or losses) on code quality or the success of the different building processes.

Each platform provide their own badges, yet you can further tune the badges style by creating your own personalized badge with Shields.io.

You will notice that the badge for Code Climate is missing in the master branch. I could not find yet a straightforward and easy implementation for several branches at Code Climate, so, the badge reports on the main branch set for the repository, in this case the *latest* branch. Also at *Shields.io* there is no shortcut to *branch* for this platform as there is for others.

I observed this same issue for COVERALLS, but then I realize that after the first commit to the master, COVERALLS actually displays nicely the information for both branches.

CHAPTER

THREE

INSTALLATION

In this page you can describe the installation steps required for end-users, use the *Contribution page* to provide the guidelines for developers.

3.1 Installation Example

At the command line:

```
pip install sampleproject
```

3.2 How to use this Template

To use this template for your projects use the green button at the main repository page.

CHAPTER

FOUR

USAGE

Describe here examples on how to use your software!

To use SampleProject in a project:

```
import sampleproject
```

CHAPTER

FIVE

CONTRIBUTING

Here we explain how to contribute to a project that adopted this template. Actually, you can use this same scheme when contributing to this template.

5.1 Fork this repository

Fork this repository before contributing. It is a better practice, possibly even enforced, that only Pull Request from forks are accepted. In my opinion this creates a cleaner representation of the whole contributions to the project.

5.2 Install for developers

First, clone the repository as described in the *section above*.

Create a dedicated Python environment where to develop the project.

If you are using `pip` follow the official instructions on Installing packages using pip and virtual environments, most likely what you want is:

```
python3 -m venv pyprojskel
source pyprojskel/bin/activate
```

If you are using Anaconda go for:

```
conda create --name pyprojskel python=3.7
conda activate pyprojskel
```

Where `pyprojskel` is the name you wish to give to the environment dedicated to this project.

Either under *pip* or *conda*, install the package in `develop` mode, and also *tox*.

```
python setup.py develop
# for pip
pip install tox bumpversion
# for conda
conda install tox bumpversion -c conda-forge
```

Under this configuration the source you edit in the repository git folder is automatically reflected in the development installation.

Continue your implementation following the development guidelines described bellow.

5.3 Branch workflow

The following applies to external contributors, yet main developers can also follow these guidelines.

Branch workflow for development and contribution should follow the Gitflow Workflow guidelines. Please read careful through that guide. Here we highlight the general approach with some tasteful additions such as the `--no-ff` flag.

5.3.1 Clone your fork

Indeed the first thing to do is to clone your fork, and keep it up to date with the upstream:

```
git clone https://github.com/YOUR-USERNAME/python-project-skeleton.git
cd into/cloned/fork-repo
git remote add upstream git://github.com/joaomcteixeira/python-project-skeleton.git
git fetch upstream
git pull upstream latest
```

5.3.2 New feature

To work on a new feature, branch out from the `latest` branch:

```
git checkout latest
git checkout -b feature_branch
```

Develop the feature and keep regular pushes to your fork with comprehensible commit messages.

5.3.3 Push to latest

To see your development accepted in the main project, you should create a Pull Request to the `latest` branch following the PULLREQUEST.rst guidelines.

Before submitting a Pull Request, verify your development branch passes all tests as *described bellow* . If you are developing new code you should also implement new test cases.

If you are an official contributor to this repository, have write permissions, and are sure the new feature branch passes tests, directly merge to the `latest` branch.

You should *bump a patch* beforehand.

```
# on your feature_branch
bumpversion patch --no-tag
git push origin feature_branch
git checkout latest
git merge --no-ff feature_branch
git push origin latest
```

The `--no-ff` option avoids `Fastforward` merging (1, 2), keeping a record of the branching out/in history.

5.3.4 To official contributors

Release Branches

Create a short lived branch to prepare for the release candidate, in this example `release/0.1.0`.

```
git checkout latest
git checkout -b release/0.1.0
```

Fix the final bugs, docs and minor corrections, and finally *bump the version*.

```
# first commit and push your changes
# then bump
bumpversion patch|minor|major
git push origin release/0.1.0
```

Finally, merge to `master` AND from `master` to `latest`.

```
git checkout master
git merge --no-ff release/0.1.0
git push origin master --tags
git checkout latest
git merge --no-ff master
```

Hotfixes from master

The hotfix strategy is applied when a bug is identified in the production version that can be easily fixed.

```
git checkout master
git checkout -b hotfix_branch
```

Work on the fix...

```
# push yours commits to GitHub beforehand
git push origin hotfix_branch
bumpversion patch
git push origin hotfix_branch --tags
git checkout master
git merge --no-ff hotfix_branch
git push origin master
git checkout latest
git merge --no-ff master
git push origin latest
```

5.4 Uniformed Tests

Thanks to Tox we can have a uniform testing platform where all developers are forced to follow the same rules and, above all, all tests occur in a controlled Python environment.

With *Tox*, the testing setup can be defined in a configuration file, the tox.ini, which contains all the operations that are performed during the test phase. Therefore, to run the unified test suite, developers just need to execute `tox`, provided tox is installed in the Python environment in use.

```
pip install tox
# or
conda install tox -c conda-forge
```

Todo: Review and consider integrating using tox to setup development envs -> https://tox.readthedocs.io/en/latest/example/devenv.html

One of the greatest advantages of using Tox together with the *src layout*, aside from uniforming the testing routines across developers, is that tests scripts actually perform against an installed source (our package) inside an isolated deployment environment. In order words, tests are performed in an environment simulating a post-deployment state instead of a pre-deploy/development environment. Under this setup, there is no need, in general cases, to deploy test scripts along with the actual source, in my honest opinion - see MANIFEST.in.

Todo: Discuss the need to deploy test scripts.

Before creating a Pull Request from your branch, certify that all the tests pass correctly by running:

```
tox
```

These are exactly the same tests that will be performed in the *CI Platforms*.

Also, you can run individual environments if you wish to test only specific functionalities, for example:

```
tox -e check  # code style and file compatibility
tox -e spell  # spell checks documentation
tox -e docs  # only builds the documentation
```

5.5 Bumpversion

I found two main version string handlers around: bumpversion and versioneer. I chose *bumpversion* for this repository template. Why? I have no argument against *versioneer* or others, simply I found bumpversion to be so simple, effective and configurable that I could only adopt it. Congratulations to both projects nonetheless.

CHAPTER

SIX

SOURCE DOCUMENTATION

6.1 sampleproject

Initial documentation of SampleProject.

6.2 AModule

Main DOCSTRING for amodule.

With several lines.

amodule.**hello**()
Print 'hello module'.

6.3 Libs

General Libraries for the project.

samplemodule that performs sample operations.

Contains:

- sampleclass

class sampleproject.libs.samplemodule.**SampleClass**
Documentation of the SampleClass.

classmethod false()
Docstrings should not start with Returns...

Nonetheless, returns False

static true()
Return True my friend.

CHAPTER

SEVEN

CHANGELOG

7.1 v0.3.1 (2021-01-22)

- Synchronized CHANGELOG with `.bumpversion`

7.2 v0.3.0 (2021-01-22)

- simplifies `setup.py`
- defines rules for `CHANGELOG.rst`
- adds `check` `tox` env to py37 machine

7.3 v0.2.2 (2021-01-22)

- Updates CI framework to GitHub Actions
- adds action to automate version bump and package build to PyPI
- completes CI for Linux, Windows, and MacOS
- reports test coverage to Codecov
- updated/enhanced bump2version configuration
- bump2version also changes CHANGELOG

7.4 v0.2.1 (2020-05-31)

- updated tox to accepts posargs in *pytest* and *flake8*

7.5 v0.2.0 (2020-01-31)

- Implemented Travis-CI for Windows, MacOSX and Linux * for Python: 3.6, 3.7 and 3.8 * all previous without using anaconda expect for MacOSX 3.8 * I have nothing against Anaconda ;-), on the contrary, I use it everyday.
- Improved `tox.ini` workflow to my current favorite standards.
- Implemented mock strategy to avoid installing dependencies for documentation generation. * TOXENV docs

7.6 v0.1.0 (2019-10-03)

- First release on PyPI.

CHAPTER

EIGHT

AUTHORS

- Joao M. C. Teixeira (webpage, github)

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX