project$template$

Release 0.1.0

May 13, 2019

Contents

This is a sample project generated by generator-python for Yeoman.[1] It has a number of features, each with its own chapter in the documentation:

- ISC license (a shorter MIT license)
- All package metadata in `pyproject.toml` (reaching standardization in PEP 518)
- Cross-platform scripts for common development tasks (linting, testing, building documentation)
- Testing with pytest, doctests, and coverage
- Continuous integration on Linux and OSX with Travis CI and Windows with AppVeyor
- Documentation with Sphinx and Read the Docs

[1] With the exception of a few additions. Most notably, the content of this documentation is not generated (but its boilerplate is).

CHAPTER 1

Why?

Imagine you're working on a Python project, and you take a detour to write a subpackage or submodule with some useful functions or a clever abstraction. This package depends on nothing else in your project; it can actually be a dependency of your project. It might be useful in some of your other projects, or it might be useful to other people in their projects. You would like to extract that package into its own project (while following best practices for directory structure, code style, tests, documentation, and continuous integration) that you can quickly and easily package and share through the Python Package Index (PyPI).

I consider this use case representative of the vast majority: an all-Python library that you want to share with yourself and others on PyPI. Common development tasks should be easy:

- running tests (across multiple versions of Python TBD);
- running a suite of state-of-the-art static analyses (including style checkers);
- building and publishing documentation (using the most common extensions);
- continuous integration on the big three platforms (Linux, OSX, and Windows); and
- publishing to PyPI (even without knowing the intricacies of Python packaging).

Tangentially, I spent a bunch of time on the documentation walking through each feature and explaining it from the ground up so that a newcomer can understand. I want the documentation to leave users with no unanswered questions. That means if you have a question, then the documentation is incomplete! Please let me know so that I can fill any gaps.

CHAPTER 2

Why did you choose X?

This project makes choices for each of its features, and there are bound to be people who do not like them or understand them. The rest of the documentation tries to explain each individual choice, but I will outline here the general philosophy.

For dependency management and packaging, there is one tool emerging that both

1. offers a good user experience to the point that you might never need to manually edit your package metadata file or learn the history and pain of Python packaging, and
2. uses a single, *standard* (PEP 518) package metadata file (`pyproject.toml`).

That tool is Poetry. Shout to Pipenv for leading the way on the first point, but in my opinion it has been overtaken.

For the rest of the tools (style checkers, static analyzers, test, docs, CI), I have tried to choose the most popular, battle-tested solutions. As the landscape changes, this project will change with it.

CHAPTER 3

Why didn't you choose Y?

If you think I've made an error, please let me know in the issues. Please remember that each choice is the result of a significant investment of my time in researching alternatives. I have not yet documented why I chose *against* the ones that I did, but I am open to having that discussion (and I would like to link back to those discussions from a "graveyard" chapter in the documentation).

3.1 License

The license for this project is the Internet Systems Consortium (ISC) license. It is functionally equivalent to the simplified BSD and MIT licenses, but without language deemed unnecessary following the Berne Convention.

ISC is the default just because it is my favorite permissive license. I would like to offer a choice of popular licenses if it is easy. I tried composing the most popular generator for licenses, but it is impossible to get the value of the choice returned to `generator-python` for writing the package metadata file. I don't plan on expending the effort to add a license prompt unless it becomes clear it is holding back many new users, but I welcome a pull request to add it.

If you want a different license, just drop it into the file `LICENSE` and edit the `license` setting in `pyproject.toml` . If you don't know what license you want, the wizard at choosealicense.com can help you pick one.

3.2 Dependencies

This project manages dependencies through `pyproject.toml` , a Python package metadata file working its way to standardization through PEP 518. `pyproject.toml` stands to replace a growing set of redundant, confusing, non-standard configuration files: `setup.py`, `requirements.txt`, `setup.cfg`, `MANIFEST.in`, and `Pipfile`.

3.2.1 Poetry

Right now, the premier tool for managing `pyproject.toml` is Poetry. The generator requires you to have Poetry installed and has instructions in its documentation for installing it.

If you're familiar with npm or Yarn, Poetry works much the same way. Poetry will create a virtual environment for your project and install your project's dependencies there, isolated from the virtual environments of other projects.

By default, Poetry will create your virtual environment underneath a cache directory in your home directory, `$HOME/.cache/pypoetry/virtualenvs`. You can find this directory by checking your Poetry configuration:

```
$ poetry config settings.virtualenvs.path
```

You can configure Poetry to create the virtual environment in your project directory, if you want:

```
$ poetry config settings.virtualenvs.in-project true
```

The in-project virtual environment directory will be named `.venv`.

3.2.2 Managing dependencies

When you run the generator, it will install the starting dependencies, but if you clone this project, you must install them yourself:

```
$ poetry install
```

Dependencies are grouped. The two most common groups are **required** (dependencies your code uses at runtime) and **development** (dependencies your project uses for development, e.g. `mypy` or `pytest`). You can add or remove dependencies easily:

```
$ poetry add requests
$ poetry remove --dev mypy
```

By default, Poetry will search the Python Package Index (PyPI) for the latest versions of the dependencies you name. To find out how to search other package repositories or how to search for specific versions, consult the Poetry documentation.

3.2.3 Default dependencies

By default, the generator does not install any required dependencies, but it does install a set of development dependencies, explained here.

Package	Reason
mypy	Type-checking.
pylint	Static analysis and PEP 8 (code style) conformance.
pydocstyle	PEP 257 (docstring style) conformance.
yapf	Formatting code.
pytest	*Testing*.
pytest-cov	Code coverage.
sphinx	*Documentation*.
sphinx-autobuild	Fast iterations on documentation.
sphinx_rtd_theme	Read the Docs theme for documentation.
toml	Reading the version from `pyproject.toml`.

3.3 Scripts

This project includes an "Invokefile" to approximate the convenience of npm scripts, at least until such scripts make their way into `pyproject.toml`. By Invokefile, I'm talking about a `tasks.py` script for the Invoke tool. Using Invoke for common tasks like running tests, formatting code, or building the documentation relieves us from having to memorize and recite the command lines or from keeping around a bunch of small shell scripts. Invoke tasks give us short, easily-remembered names for these functions.

Previously, this project used a Makefile for scripts. Make has the advantage that it is included by default on most Linuxes and OSX, and that it is more well known than Invoke, but I switched to Invoke because it is cross-platform, just like Python. The *continuous integration* scripts use Invoke to run the tests to ensure that (1) the tests are run the same way on every platform and that (2) the Invokefile is written correctly.

The default Invokefile has a few tasks:

Target	Task
lint	Run the *linters*: style checkers, type checker, and static analyzers.
test	Run the tests (including doctests) with coverage.
html	Build the documentation in HTML.
serve	Launch a server for the HTML documentation that, whenever a change is detected, rebuilds it and refreshes your browser.

Note: The scripts assume they are running in the virtual environment of the project. You should invoke them like this:

```
$ poetry run invoke <task>
```

Alternatively, if you want every command to conveniently execute in the virtual environment, then you can start a shell in that environment:

```
$ poetry shell
```

3.4 Code

The generator does not write any code for you, but it will generate an empty package or module based on your choice. The tests directory will be created for you, but you'll need to add the first test.

This sample project has a single-file module (`project_template.py`) and one test (`test_greeting.py`).

3.4.1 Quality

There are a few tools installed to help you maintain high code quality. All of these are executed with the `lint` *script*:

```
$ poetry run invoke lint
```

Tool	Reason
mypy	Type-checking.
pylint	Static analysis and PEP 8 (code style) conformance.
pydocstyle	PEP 257 (docstring style) conformance.

There is a Pylint configuration in `.pylintrc` . Its settings are documented there. To learn how to enable, disable, or configure Pylint's diagnostics, consult the documentation on message control or take a look at Pylint's own configuration file.

3.5 Tests

The generator chooses pytest for the testing framework. All you have to do to get started is add test modules in the `tests` package. Separately, you can write small tests in function docstrings using doctests. All of these tests are executed as part of the `test` *script*. The report includes coverage:

```
$ poetry run invoke test
pytest --cov=project_template --doctest-modules --ignore=docs
============================= test session starts ==============================
platform linux -- Python 3.6.8, pytest-4.4.1, py-1.8.0, pluggy-0.9.0
rootdir: /home/jfreeman/code/project-template-python
plugins: cov-2.7.1
collected 2 items

project_template.py .                                                    [ 50%]
tests/test_greeting.py .                                                 [100%]

----------- coverage: platform linux, python 3.6.8-final-0 -----------
Name                  Stmts   Miss  Cover
---------------------------------------
project_template.py       2      0   100%


=========================== 2 passed in 0.07 seconds ===========================
```

3.6 Continuous Integration

Continuous integration is the name for automatically executing your tests when you push changes to your software. The generator generates configuration files for the Travis CI and AppVeyor continuous integration platforms, and includes their status badges for the project in the README.

3.6.1 Travis CI

Travis CI will test your software on Linux and OSX across Python versions 3.6, 3.7, and 3.8, excluding those less than your minimum supported.

Note: As of 2019 May 6, Travis CI does not have an image for OSX that includes Python 3.8. While Homebrew is available, trying to update the package list seems to fail (or return a spurious non-zero status), which I have not yet investigated. For the same reason, it is impossible to install nproc (used by the lint *script*) on the OSX image that has Python 3.6.

To set up Travis CI for your project, you'll need to:

- Grant access to the Travis CI application for your account or organization. This will let Travis CI add hooks and mark commits with the status of your builds.
- Log in to Travis CI with your GitHub credentials.

- Enable your repository on Travis CI. This will create a hook on GitHub to notify Travis CI whenever you push to your repository.

3.6.2 AppVeyor

AppVeyor will test your software on Windows for Python versions 3.6 and 3.7, excluding those less than your minimum supported.

To set up AppVeyor for your project, you'll need to:

- Log in to AppVeyor with your GitHub credentials.
- Grant access to AppVeyor for the project you want to test. This will let AppVeyor add hooks and mark commits with the status of your builds.
- Add the project on AppVeyor. This will create a hook on GitHub to notify AppVeyor whenever you push to your repository.

3.7 Documentation

The most popular tool for writing Python documentation is Sphinx, and the most popular host for it is Read the Docs (RTD). Sphinx uses **reStructuredText_** for its markup language. The generator gives a skeleton for your documentation in the `docs` directory .

3.7.1 Defaults

- There are many themes for Sphinx, but the generator chooses the RTD theme. It is designed to look good on desktop and mobile devices.
- The latest stable version of Sphinx is 2.0, but it is not yet compatible with the RTD theme. For now, the generator chooses the latest 1.x version.
- The documentation is versioned with your code. The generated configuration reads the version from `pyproject.toml`, the single source of truth for package metadata.
- The generator includes a style sheet that will wrap table text for you. (Thanks to Rackspace!)
- The landing page is generated from `index.rst`. By default, it includes a section of content from the project README (so that you don't have to write it twice). Initially, that content is the project name and the project badges.

3.7.2 Editing

You can edit the landing page by editing `index.rst`. If you want to add more pages ("chapters"), then add new reStructuredText files to the `docs` directory and link them from the `toctree` in `index.rst`.

While you're working on documentation, you can use the `serve` *script* to launch a server for the HTML build of your documentation. Whenever you change a file, the server will rebuild your documentation and refresh your browser:

```
$ poetry run invoke serve
```

3.7.3 Publishing

When you're ready to publish your documentation on Read the Docs, follow these steps:

- Log in to Read the Docs with your GitHub credentials.
- Import your project repository. (You may likely need to refresh the list.)
- Click "Build version" for the first build. Subsequent builds are automatically triggered when you push to GitHub.

3.8 Package

Once you're done testing and documenting your project, you may want to package and distribute it so that other people can install, import, and use it. Poetry can help you upload a package to the Python Package Index (PyPI). You only get one shot to publish a specific version on PyPI, however. If you make a mistake, you have to publish a new version. That's why they made a TestPyPI where you can overwrite a version until you get it right.

3.8.1 Getting started

Create accounts on both PYPI and TestPyPI. Remember your username and password for both.

Check that Poetry already knows about the TestPyPI repository:

```
$ poetry config repositories.test.url
repositories.test.url = "https://test.pypi.org/legacy/"
```

Give Poetry your credentials for both of your accounts:

```
$ poetry config http-basic.pypi <username> <password>
$ poetry config http-basic.test <username> <password>
```

3.8.2 Publishing

If you've already published the version that is named in your `pyproject.toml`, then you'll need to pick the next version. Poetry can help with that:

```
$ poetry version
$ poetry version minor
$ poetry version major
```

Build your project:

```
$ poetry build
```

Publish to TestPyPI:

```
$ poetry publish --repository test
```

At this point, you should be able to create another project, install your package as a dependency, import it, and test it. If you use Poetry to manage that project, you'll need to add TestPyPI as a source in its `pyproject.toml`:

```
[[tool.poetry.source]]
name = "testpypi"
url = "https://test.pypi.org/simple"
```

Once you feel confident your package is in good working order, publish it to PyPI:

```
$ poetry publish
```