# Python Project Documentation

*Release 1.0*

**Tim Diels**

**Jan 10, 2018**

# Contents


**1 Simple project structure** **3**
- 1.1 Code repository usage . . . 3
- 1.2 Versioning . . . 3
- 1.3 Testing . . . 4
- 1.4 Documenting . . . 4
- 1.5 Developing . . . 5
- 1.6 Release checklist . . . 6
- 1.7 Deploying . . . 8
- 1.8 Creating a new project . . . 8

**2 Changelog** **9**
- 2.1 1.2.0 . . . 9
- 2.2 1.1.1 . . . 9
- 2.3 1.0.0 . . . 10

**3 Indices and tables** **11**

Python Project is a collection of Python project templates, checklists and documentation of their associated workflow. Currently, the collection contains just one project structure.

Contents:

CHAPTER 1

# Simple project structure

The simple project structure and workflow are suitable for projects where no more than 1 developer works on the project at the same time. The workflow assures high quality by automated tests and checklists.

The following are guidelines. Feel free to deviate from them when it makes sense to do so.

## 1.1 Code repository usage

There is one repository on GitLab or GitHub. Simply clone it and push your changes directly.

Consider using a branch named 'next' to push your changes to before merging them onto master. This way you can rebase and force push to the next branch while never having to force push to the master branch. As such the master branch's history never changes and can be relied on.

We'll refer to the directory that is created when cloning the repository as the project root.

Each commit should change/achieve one thing (not too fine grained though). If you've made many changes without committing, try separating them with `git add -i` using the patch command, into multiple commits.

Tip: better commit too often than too little, it's easier to squash than to split commits in a rebase.

## 1.2 Versioning

Use semantic versioning.

Version tags needn't begin with `v`, just be consistent. Historically the v prefix was used for version tags because tags had to be valid identifiers, i.e. had to start with alphabetic character. (Source)

## 1.3 Testing

Features that need to have assured quality should be covered by automated tests. *pytest* is used as testing framework. To run tests, run `py.test` in the project root.

To generate test coverage info, use `py.test --cov`. For this to work, all tests should be run and the should be ran sequentially. E.g. when using `pytest-xdist`, add `-n0` to run without xdist.

Tests are located in `$pkg/tests`. For example, for the `subproject` project below, which only provides packages starting with `project.subproject`, its file tree would look like:

```
project
    __init__.py
    subproject
        __init__.py
        algorithms
            __init__.py
            algorithm.py
            algorithm2.py
        util.py
        tests
            __init__.py
            conftest.py
            test_algorithms.py
            test_util.py
```

There is only one `tests` package, under which all tests can be found. They can be organised as you see fit, e.g. the following is fine too:

```
tests
    __init__.py
    conftest.py
    algorithms
        __init__.py
        test_algorithm1.py
        test_algorithm2.py
    test_util.py
```

## 1.4 Documenting

Documentation is written as reStructuredText files (.rst) located in `docs`. These get compiled by Sphinx into HTML, PDF and epub. When a commit is added to the master on GitLab/GitHub, Read the Docs (RTD) is notified and starts a build of the documentation with Sphinx, after which it hosts the result on its website, as the 'latest' version. The Sphinx configuration file is `docs/conf.py`.

Read the Docs can be configured per project on https://readthedocs.io or for most options (but not all) in `readthedocs.yml` in the project root.

To generate html documentation locally, run `cd docs; make clean html`. The output is in `build/html`. The makefile also has targets for latex and other formats.

## 1.5 Developing

Work test-driven for the parts that will need testing eventually, this saves the work of manually testing the code. You should run the tests occasionally and avoid pushing commits to master with failing tests. It's okay to commit with failing tests if you intend to rebase later to fix those commits.

Dependencies should at least be constrained to their major version to avoid breakage in the future. After all, a project's major version is bumped iff there is a backwards-incompatible change, which may break the project. For example `install_requires=['dep==1.*']`.

Tip: Instead of `dep2>=1.1`, write `dep2~=1.1`, this is equivalent to `dep2>=1.1, dep2==1.*`.

To set up your local development environment:

1. `git clone $github_or_gitlab_url`
2. It's recommended to be in a Python virtual environment while developing. This hides all system/user-wide installed packages, amongst other things this helps test for missing dependencies in `setup.py`.

   To create one:

   ```
   cd $project_root
   python -m venv venv  # create venv. Be sure to use the python version you want to␣
   ↪test with, e.g. python3.5
   ```

   To enter the venv use `. venv/bin/activate`, to exit `deactivate`. When using Eclipse PyDev, you can configure a new interpreter on the project which points to the venv instead of the system-wide interpreter (which makes for better hinting of missing imports).

   It's a good idea to install `wheel` (speeds up installation by using binaries when available) and upgrade `setuptools` and `pip`:

   ```
   pip install -U pip setuptools wheel
   ```

3. Install the project editably:

   ```
   pip install -e '.[all]'
   ```

   This also installs the dependencies. The -e option installs it in such a way that changes to the code will be reflected in the installed package, so you don't have to reinstall each time you want to use a script provided by the package. `[all]` tells pip to install the `all` extras, which in case of our `setup.py` contains the union of all extra dependencies.

   However, when changing `install_requires`, `extras_require` or `entry_points` in `setup.py`, you should reinstall (by rerunning the above command)

Code guidelines:

- Docstrings must follow NumPy style.
- Be precise when specifying expected types in docstrings, e.g. in the `Parameters` section. E.g. specify types according to PEP484 in docstrings. Do not use type hints as of the time of writing Sphinx does not display these properly. The typing module is especially helpful for specifying types.
- When using someone else's code or idea, give credit in a comment in the source file, not in the documentation, unless on a separate acknowledgements page.

Commit guidelines:

- Ensure `py.test` passes. If not, fix or mention tests are failing in the commit message (e.g. `[failing]`, `[broken]`).

- Remove trailing whitespace, unless it is significant. E.g. you cannot remove the trailing whitespace on the empty line in the string given to dedent:

```
dedent('''\
    First line

    Last'''
)
```

  By removing trailing whitespace one cause of git merge conflicts where semantically nothing has changed, is prevented. Command line git will highlight trailing whitespace. Most editors/IDEs can be configured to show trailing whitespace (Google `$editor show trailing whitespace`).

## 1.6 Release checklist

To release a new version to PyPI:

1. Ensure your working tree is clean and you are on the master branch:

```
git checkout master && git status
```

   Otherwise you risk including files/changes in the package uploaded to PyPI that are not in the git repo.

2. Ensure tests pass (`py.test`) in a fresh venv which does not rely on unreleased dependencies (i.e. no `pip -e ...`).

3. Ensure `docs/conf.py` is in sync with `setup.py`. E.g. if you reference `pytil` with intersphinx and have pinned it in `setup.py` to `pytil==5.*`, your `intersphinx_mapping` should contain:

```
'pytil': ('http://pytil.readthedocs.io/en/5.0.0/', None),
```

4. Update changelog in `docs/changelog.rst` by reviewing commit messages since last release.

   The changelog is mainly intended to answer:

   - Which version do I need given that I use feature X?
   - Something is broken, could this be due to a recent update?
   - What's new? What features got added that I may want to use?

   Suggested sections:

   - Backwards incompatible changes:
     - Removal of anything that's part of the public interface, e.g. function of an API
     - Any other breaking changes to the public interface
   - Enhancements/additions:
     - Addition of function, parameter, ... to public interface
     - Change that improves a function, ... of the public interface
     - Anything non-breaking that is part of the public interface
   - Fixes:
     - Bug fix to anything that affects the public interface. Briefly describe the symptoms of the bug and the fix.

- Internal / implementation details:
  - switching to different libraries
  - non-trivial changes to project structure
  - anything related to tests
  - fixes that do not affect the public interface
  - anything not part of the public interface. This section is more so intended for developers. It's debatable whether this section is necessary, you might indeed just use the commit log.

5. Set the version for the release (semantic versioning: bump major iff backwards incompatible change, else bump minor if enhancement/addition, else bump patch; refer to the changelog)
   (a) in `setup.py`
   (b) in the `__init__.py` of the package, e.g. for a subproject in `project/subproject/__init__.py`
6. Generate documentation locally and review all changed areas (git diff of docs since last tag):

```
cd docs
make clean html
firefox build/html/index.html
```

   Common errors:
   - Formatting errors?
   - Dead links? (feel free to use a tool for finding dead links instead)
   - Unfinished parts?
7. Point documentation link to tagged RTD release by changing `latest` to `$version`.
8. Push tagged commit with the changes:

```
git commit -am $version
git push
git tag $version
git push origin $version
```

9. Activate the version just released at RTD, see the Versions tab.
10. Release to PyPI:

```
python setup.py sdist bdist_wheel
twine upload dist/*
```

    - At first glance, looks correct
    - Homepage link is not dead
    - Documentation link links to tagged version at RTD

    Tip: `~/.pypirc` can be used to avoid retyping your password each time, chmod it to 0600
11. Bump patch and dev version (e.g. 4.0.0 -> 4.0.1.dev1, because 4.0.0.dev < 4.0.0), and point documentation link back to latest RTD. Then `git commit -am $dev_version && git push`.

## 1.7 Deploying

In your development virtual environment, run:

```
pip freeze > requirements.txt
```

Copy this to the production environment, create a new venv and then run:

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

This reduces the odds of bugs due to having installed different versions of dependencies. Alternatively, if you have high test coverage, you can just install and run the tests. Also note that a venv cannot be moved (it uses absolute paths), you have to recreate it on the new location.

## 1.8 Creating a new project

To create a new project with this structure:

1. Install cookiecutter: `pip install cookiecutter`.
2. Create the project from the cookiecutter template:

   ```
   git clone https://github.com/timdiels/python-project
   cookiecutter python-project/cookiecutters/simple
   rm -rf python-project  # when satisfied with the generated project
   ```

   and cd into the created directory (`$pypi_name`).
3. Initialize git and push first commit to GitHub/GitLab:

   ```
   git init
   git add .
   git commit -m 'Initial commit'
   git remote add origin $repo_url
   git push -u origin master
   ```

4. Enable documentation builds for the project on Read the Docs.
   (a) Import the project from GitHub/GitLab. Make an account and link it to your GitHub/GitLab if you haven't already.
   (b) Go to `Admin > Advanced settings` and check "Install your project inside a virtualenv using setup.py install". This allows access to the project's code from within conf.py and the documentation (e.g. for using autodoc directives).

   As requirements file, enter `rtd_requirements.txt`. RTD will `pip install -r rtd_requirements.txt`. The `rtd_requirements.txt` of the template references the optional dev dependencies, so you can forget about this file and add doc dependencies to `extras_require['dev']` in `setup.py`.

   If only supporting Python 3, change the interpreter to CPython3 (or similar implementation).
5. To start development, *set up your local development environment*.

CHAPTER 2

# Changelog

Semantic versioning is used.

## 2.1 1.2.0

- Enhancements:
  - Test with fresh venv on release
- Fixes:
  - Fix setup.py package_data
  - Fix broken semver link
  - Do ignore SIGPIPE in tests
  - Add missing semver reference
  - RTD installed the developed package from PyPI. Now, it installs the latest on the current branch.

## 2.2 1.1.1

- Backwards incompatible changes: none
- Enhancements/additions:
  - Add code guidelines
  - Add commit guidelines
- Fixes:
  - docs/developer_guide.rst: link to simple project was broken
- Internal / implementation details: none

## 2.3 1.0.0

Initial release.

CHAPTER 3

# Indices and tables

- genindex
- search