# pyscaffold-namespace-example Documentation

***Release 0.0.post1.dev5+g9d53b70***

**Sterling G. Baird**

**Apr 16, 2024**

# CONTENTS

This is the documentation of **pyscaffold-namespace-example**.

---

**Note:** This is the main page of your project's Sphinx documentation. It is formatted in reStructuredText. Add additional pages by creating rst-files in `docs` and adding them to the toctree below. Use then references in order to link them from this page, e.g. *Contributors* and *Changelog*.

It is also possible to refer to the documentation of other Python packages with the Python domain syntax. By default you can reference the documentation of Sphinx, Python, NumPy, SciPy, matplotlib, Pandas, Scikit-Learn. You can add more by extending the `intersphinx_mapping` in your Sphinx's `conf.py`.

The pretty useful extension autodoc is activated by default and lets you include documentation from docstrings. Docstrings can be written in Google style (recommended!), NumPy style and classical style.

---

CHAPTER

# ONE

# CONTENTS

## 1.1 pyscaffold-namespace-example

Add a short description here!

A longer description of your project goes here...

### 1.1.1 Note

This project has been set up using PyScaffold 4.5. For details and usage information on PyScaffold see https://pyscaffold.org/.

## 1.2 Contributing

Welcome to `pyscaffold-namespace-example` contributor's guide.

This document focuses on getting any potential contributor familiarized with the development processes, but other kinds of contributions are also appreciated.

If you are new to using git or have never collaborated in a project previously, please have a look at contribution-guide.org. Other resources are also listed in the excellent guide created by FreeCodeCamp[1].

Please notice, all users and contributors are expected to be **open, considerate, reasonable, and respectful**. When in doubt, Python Software Foundation's Code of Conduct is a good reference in terms of behavior guidelines.

[1] Even though, these resources focus on open source projects and communities, the general ideas behind collaborating with other developers to collectively create software are general and can be applied to all sorts of environments, including private companies and proprietary code bases.

### 1.2.1 Issue Reports

If you experience bugs or general issues with `pyscaffold-namespace-example`, please have a look on the issue tracker. If you don't see anything useful there, please feel free to fire an issue report.

---

**Tip:** Please don't forget to include the closed issues in your search. Sometimes a solution was already reported, and the problem is considered **solved**.

---

New issue reports should include information about your programming environment (e.g., operating system, Python version) and steps to reproduce the problem. Please try also to simplify the reproduction steps to a very minimal example that still illustrates the problem you are facing. By removing other factors, you help us to identify the root cause of the issue.

### 1.2.2 Documentation Improvements

You can help improve `pyscaffold-namespace-example` docs by making them more readable and coherent, or by adding missing information and correcting mistakes.

`pyscaffold-namespace-example` documentation uses Sphinx as its main documentation compiler. This means that the docs are kept in the same repository as the project code, and that any documentation update is done in the same way was a code contribution.

When working on documentation changes in your local machine, you can compile them using `tox`:

```
tox -e docs
```

and use Python's built-in web server for a preview in your web browser (`http://localhost:8000`):

```
python3 -m http.server --directory 'docs/_build/html'
```

### 1.2.3 Code Contributions

#### Submit an issue

Before you work on any non-trivial code contribution it's best to first create a report in the issue tracker to start a discussion on the subject. This often provides additional considerations and avoids unnecessary work.

#### Create an environment

Before you start coding, we recommend creating an isolated virtual environment to avoid any problems with your installed Python packages. This can easily be done via either `virtualenv`:

```
virtualenv <PATH TO VENV>
source <PATH TO VENV>/bin/activate
```

or Miniconda:

```
conda create -n pyscaffold-namespace-example python=3 six virtualenv pytest pytest-cov
conda activate pyscaffold-namespace-example
```

### Clone the repository

1. Create an user account on GitHub if you do not already have one.
2. Fork the project repository: click on the *Fork* button near the top of the page. This creates a copy of the code under your account on GitHub.
3. Clone this copy to your local disk:

```
git clone git@github.com:YourLogin/pyscaffold-namespace-example.git
cd pyscaffold-namespace-example
```

4. You should run:

```
pip install -U pip setuptools -e .
```

   to be able to import the package under development in the Python REPL.

5. Install `pre-commit`:

```
pip install pre-commit
pre-commit install
```

   `pyscaffold-namespace-example` comes with a lot of hooks configured to automatically help the developer to check the code being written.

### Implement your changes

1. Create a branch to hold your changes:

```
git checkout -b my-feature
```

   and start making changes. Never work on the main branch!

2. Start your work on this branch. Don't forget to add docstrings to new functions, modules and classes, especially if they are part of public APIs.
3. Add yourself to the list of contributors in `AUTHORS.rst`.
4. When you're done editing, do:

```
git add <MODIFIED FILES>
git commit
```

   to record your changes in git.

   Please make sure to see the validation messages from `pre-commit` and fix any eventual issues. This should automatically use flake8/black to check/fix the code style in a way that is compatible with the project.

---

**Important:** Don't forget to add unit tests and documentation in case your contribution adds an additional feature and is not just a bugfix.

Moreover, writing a descriptive commit message is highly recommended. In case of doubt, you can check the commit history with:

```
git log --graph --decorate --pretty=oneline --abbrev-commit --all
```

to look for recurring communication patterns.

5. Please check that your changes don't break any unit tests with:

```
tox
```

(after having installed `tox` with `pip install tox` or `pipx`).

You can also use `tox` to run several other pre-configured tasks in the repository. Try `tox -av` to see a list of the available checks.

### Submit your contribution

1. If everything works fine, push your local branch to GitHub with:

```
git push -u origin my-feature
```

2. Go to the web page of your fork and click "Create pull request" to send your changes for review.

### Troubleshooting

The following tips can be used when facing problems to build or test the package:

1. Make sure to fetch all the tags from the upstream repository. The command `git describe --abbrev=0 --tags` should return the version you are expecting. If you are trying to run CI scripts in a fork repository, make sure to push all the tags. You can also try to remove all the egg files or the complete egg folder, i.e., `.eggs`, as well as the `*.egg-info` folders in the `src` folder or potentially in the root of your project.

2. Sometimes `tox` misses out when new dependencies are added, especially to `setup.cfg` and `docs/requirements.txt`. If you find any problems with missing dependencies when running a command with `tox`, try to recreate the `tox` environment using the `-r` flag. For example, instead of:

```
tox -e docs
```

Try running:

```
tox -r -e docs
```

3. Make sure to have a reliable `tox` installation that uses the correct Python version (e.g., 3.7+). When in doubt you can run:

```
tox --version
# OR
which tox
```

If you have trouble and are seeing weird errors upon running `tox`, you can also try to create a dedicated virtual environment with a `tox` binary freshly installed. For example:

```
virtualenv .venv
source .venv/bin/activate
.venv/bin/pip install tox
.venv/bin/tox -e all
```

4. Pytest can drop you in an interactive session in the case an error occurs. In order to do that you need to pass a `--pdb` option (for example by running `tox -- -k <NAME OF THE FALLING TEST> --pdb`). You can also setup breakpoints manually instead of using the `--pdb` option.

### 1.2.4 Maintainer tasks

#### Releases

If you are part of the group of maintainers and have correct user permissions on PyPI, the following steps can be used to release a new version for `pyscaffold-namespace-example`:

1. Make sure all unit tests are successful.
2. Tag the current commit on the main branch with a release tag, e.g., `v1.2.3`.
3. Push the new tag to the upstream repository, e.g., `git push upstream v1.2.3`
4. Clean up the `dist` and `build` folders with `tox -e clean` (or `rm -rf dist build`) to avoid confusion with old builds and Sphinx docs.
5. Run `tox -e build` and check that the files in `dist` have the correct version (no `.dirty` or git hash) according to the git tag. Also check the sizes of the distributions, if they are too big (e.g., > 500KB), unwanted clutter may have been accidentally included.
6. Run `tox -e publish -- --repository pypi` and check that everything was uploaded to PyPI correctly.

## 1.3 License

## 1.4 Contributors

- Sterling G. Baird <sterling.baird@utoronto.ca>

## 1.5 Changelog

### 1.5.1 Version 0.1

- Feature A added
- FIX: nasty bug #1729 fixed
- add your changes here!

## 1.6 my_namespace

### 1.6.1 my_namespace namespace

#### Subpackages

#### my_namespace.my_package_1 package

#### Submodules

#### my_namespace.my_package_1.skeleton module

This is a skeleton file that can serve as a starting point for a Python console script. To run this script uncomment the following lines in the `[options.entry_points]` section in `setup.cfg`:

```
console_scripts =
     fibonacci = my_package_1.skeleton:run
```

Then run `pip install .` (or `pip install -e .` for editable mode) which will install the command `fibonacci` inside your current environment.

Besides console scripts, the header (i.e. until `_logger...`) of this file can also be used as template for Python modules.

**Note:** This file can be renamed depending on your needs or safely removed if not needed.

### References

- https://setuptools.pypa.io/en/latest/userguide/entry_point.html
- https://pip.pypa.io/en/stable/reference/pip_install

my_namespace.my_package_1.skeleton.**fib**(*n*)

Fibonacci example function

**Parameters**
**n** (*int*) – integer

**Returns**
n-th Fibonacci number

**Return type**
int

my_namespace.my_package_1.skeleton.**main**(*args*)

Wrapper allowing *fib()* to be called with string arguments in a CLI fashion

Instead of returning the value from *fib()*, it prints the result to the `stdout` in a nicely formatted message.

**Parameters**
**args** (*List[str]*) – command line parameters as list of strings (for example `["--verbose", "42"]`).

my_namespace.my_package_1.skeleton.**parse_args**(*args*)

Parse command line parameters

**Parameters**
**args** (*List[str]*) – command line parameters as list of strings (for example `["--help"]`).

**Returns**
command line parameters namespace

**Return type**
`argparse.Namespace`

my_namespace.my_package_1.skeleton.**run**()

Calls *main()* passing the CLI arguments extracted from `sys.argv`

This function can be used as entry point to create console scripts with setuptools.

my_namespace.my_package_1.skeleton.**setup_logging**(*loglevel*)

Setup basic logging

**Parameters**
**loglevel** (*int*) – minimum loglevel for emitting messages

### Module contents

### my_namespace.my_package_2 package

### Submodules

### my_namespace.my_package_2.skeleton_2 module

This is a skeleton file that can serve as a starting point for a Python console script. To run this script uncomment the following lines in the `[options.entry_points]` section in `setup.cfg`:

```
console_scripts =
     fibonacci = my_package_1.skeleton:run
```

Then run `pip install .` (or `pip install -e .` for editable mode) which will install the command `fibonacci` inside your current environment.

Besides console scripts, the header (i.e. until `_logger`...) of this file can also be used as template for Python modules.

**Note:** This file can be renamed depending on your needs or safely removed if not needed.

### References

- https://setuptools.pypa.io/en/latest/userguide/entry_point.html
- https://pip.pypa.io/en/stable/reference/pip_install

my_namespace.my_package_2.skeleton_2.**fib**(*n*)

Fibonacci example function

**Parameters**
**n** (*int*) – integer

**Returns**
n-th Fibonacci number

**Return type**
int

my_namespace.my_package_2.skeleton_2.**main**(*args*)

Wrapper allowing *fib()* to be called with string arguments in a CLI fashion

Instead of returning the value from *fib()*, it prints the result to the `stdout` in a nicely formatted message.

**Parameters**
**args** (*List[str]*) – command line parameters as list of strings (for example `["--verbose", "42"]`).

my_namespace.my_package_2.skeleton_2.**parse_args**(*args*)

Parse command line parameters

**Parameters**
**args** (*List[str]*) – command line parameters as list of strings (for example `["--help"]`).

**Returns**
command line parameters namespace

**Return type**
`argparse.Namespace`

my_namespace.my_package_2.skeleton_2.**run**()

Calls *main()* passing the CLI arguments extracted from `sys.argv`

This function can be used as entry point to create console scripts with setuptools.

my_namespace.my_package_2.skeleton_2.**setup_logging**(*loglevel*)

Setup basic logging

**Parameters**

**loglevel** (*int*) – minimum loglevel for emitting messages

### Module contents

CHAPTER

TWO

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## m

# INDEX