cookiecutter-research-template Documentation

Tobias Raabe

Apr 10, 2019

Getting Started

This repository lays out the structure for a reproducible research project based on the Waf framework.

It is derived from https://github.com/hmgaudecker/econ-project-templates and the authors of this project deserve all the credit for the implementation of Waf as a framework for reproducible research. My contribution is to add several helpers around the project which are common in software engineering and should help researchers to write better code, therefore more comprehensible and reproducible research.

CHAPTER 1

Installation

This is a Cookiecutter template. To use it, you need to install `cookiecutter` by running

```
$ pip install cookiecutter
```

After that, you can quickly set up a new research project with this template by typing

```
$ cookiecutter https://github.com/tobiasraabe/cookiecutter-research-template.git
```

Answer all the prompts and a folder `cookiecutter-research-template` is created in your current directory. Rename the folder initialize a repository.

One of the last prompts is about whether the template should create a conda environment from the pre-configured *environment.yml*. If that is not what you want, stick to the default answer. You can fetch it later by running

```
$ conda env create -f environment.yml -n <env-name>
```

Happy research!

CHAPTER 2

Usage

After the installation, here is the manual for running the project.

2.1 Waf

The general research project must be configured in advance with

```
$ python waf.py configure
```

Make sure that all steps pass successfully. Otherwise determine what is missing, fix it and rerun the command. After that, run

```
$ python waf.py (build)
```

where `build` is optional, but it executes the same action. If you want to delete everything created in `bld` and restart the project from the source files, run

```
$ python waf.py distclean configure
```

2.2 Quality Checks

The quality of the code base is ensured by pre-commit-hooks which are automatically executed before changes are committed. If a check fails, the commit is aborted. To install the hooks, type

```
$ pre-commit install
```

After that, run

```
$ pre-commit run --all-files
```

to execute the checks without making a commit. Currently, the following hooks are installed:

- black - The Uncomprimising Python Formatter
- blacken-docs - Black for Documentation
- flake8 - Linting
- reorder-python-imports
- doc8
- check-yaml - Validating .yaml files

To update all hooks, type

```
$ pre-commit autoupdate
```

which will change the versions in `.pre-commit-config.yaml`. Do that from time to time.

2.3 Testing

Before committing changes, make sure that everything works fine. Run

```
$ tox
```

and the whole test suite will be run.

CHAPTER 3

pyup.io

pyup.io is a service which helps you to keep your dependencies up-to-date. When connected to your Github repository, a pyup-bot will automatically create pull-requests in pre-defined intervals to update `requirements.txt` or `environment.yml` (only the packages under `pip:`) to the latest package versions.

3.1 Caveat

This tool is not a no-brainer. Newer does not mean better as it might introduce bugs or break your project pipeline, leads to incompatibilities among installed packages. But, this tool keeps you notified when libraries change and provides easy access to the changelogs. After reading them carefully, you can decide to update or not.

3.2 Installation

Just go to https://pyup.io and choose login via Github in the upper right corner of the website. You will be redirected to Github to allow the service to read your repositories, etc..

After that, when logged in on https://pyup.io, choose add repository in the upper right corner. You are asked about how you would like to schedule updates, but that does not matter as the service is already configured by `.pyup.yml` in your project folder. If you want to change the settings, have a look at https://pyup.io/docs/bot/config/ to see all possible options.

CHAPTER 4

Travis-CI

Travis-CI is a service which allows you to test and deploy your projects. We are only interested in the first aspect, testing, as Travis-CI allows us to run a complete battery of tests every time a new commit is made on the master branch or every time a pull-request is updated. This ensures that we are gradually improving the project and do not introduce bugs or style issues in areas where we already have tests.

Broadly speaking, there are two categories of tests we are implementing in a research project. The first category is about testing our data to ensure that the source files are the same, intermediate results did not change, etc.. Most of the time, researchers are bounded by confidentiality agreements to keep their data private. In this case you cannot use Travis-CI to test your data and you need to skip this part of the testing battery.

The second category of tests concerns the code which does not normally fall under the former constraint and can be given into the hands of private company.

4.1 Installation

To enable testing, go to https://travis-ci.com and choose log in with Github in the upper right corner. Then, you have to agree that Travis-CI is allowed to have access to your repositories. There is nothing to do afterwards as Travis-CI will automatically check your repositories for a `.travis.yml` which includes all the build information.

To get an impression of a configuration file, take the following example of the Travis-CI configuration of this template.

```
notifications:
  email:
    on_success: never
    on_failure: never

language: python

python: {{ cookiecutter.python_version }}

before_install:
  - sudo apt-get -qq update
```

(continues on next page)

(continued from previous page)

```
  - wget https://github.com/jgm/pandoc/releases/download/2.1.3/pandoc-2.1.3-1-amd64.
↪deb -O $HOME/pandoc.deb
  - sudo dpkg -i $HOME/pandoc.deb

install:
  - pip install -U tox

script:
  - tox -e flake8
  - tox -e black
  - tox -e docs
  - tox -e sphinx

# We only want to build with Travis when pushing to master or PR:
# https://stackoverflow.com/questions/31882306/how-to-configure-travis-ci-to-
# build-pull-requests-merges-to-master-w-o-redunda/31882307#31882307
branches:
  only:
    - master
```

- `notifications` lets you define whether you want to be notified if a tests fails for some commit. My personal opinion is to disable notifications as they will flood your email inbox. You can also send notifications to Slack or other services.
- `language` defines the main language of your project. Depending on this choice, there are several other tools pre-installed.
- `python` sets the Python version of your test environment. The default is 3.6, but you are free to change it. Furthermore, you can test your project against multiple python versions by using:

  ```
  python:
    - 3.5
    - 3.6
  ```

  Note that for each Python version a different build is created meaning the same tests would run in a Python 3.5 and 3.6 environment in parallel.

The next three steps define the build lifecycle. There are several ways to differentiate between different stages of the build process. Here is one.

- In `before_install` we download pandoc to be able to use the latex builder for our documentation.
- In `install` we make sure that tox is installed and has the latest version to run our tests.
- In `script` we actually run the tests. As mentioned before, if you only want to run a subset of tests defined in tox, change the config to

  ```
  install:
    - tox -e flake8
    - tox -e black
  ```

  or something similar. Maybe you want to exclude `pytest` as some tests depend on data. In this case, I would recommend that you still include `pytest` and mark tests which cannot succeed for different reasons.
- `branches` includes a current fix so that commits on PRs are not built twice.

There are a lot of things you can do. See this document if you are looking for a different configuration.

CHAPTER 5

Code Conventions

Code conventions are rules to follow during coding to prevent mistakes and ensure readability. The last point is crucial and cannot be stressed enough.

Code is far more often read than written.

Therefore, one has a huge incentive to write as readable as possible to reduce one's own mental effort, to reduce it for others who then are more willing to contribute and use readability as a mean to replicability.

There are two different kind of programs included in the template to achieve this goal.

5.1 Formatters

Formatters are tools which take the code and transform it to something else without influencing the way the code works.

5.1.1 Black

Black is relatively new in the Python ecosystem and calls itself the uncompromising Python code formatter. The slogan is true. There are almost no options to choose a different style except line length and whether you want to use single quotes, `'`, or double quotes, `"`, for strings. But, it definitely produces more readable code and helps you to learn what good code looks like.

The default line length is set to 88. Other common values are 79, 80, 88, 120. 80 characters can be displayed on most devices and most of the time it is possible to have to files side-by-side to cross-read. An 88 character limit seems to produce much shorter files while only increasing the width by 10%.

There is a heated debate about single versus double quotes. Single quotes seem to produce less visual noise for readers. Double quotes anticipate apostrophes in English text. Some others use single quotes for data and double quotes for real language. Black settles on double quotes only and despite that I am more inclined to use single quotes myself, I think standardization is a good thing. Therefore, the formatter will recode all strings to double quotes. If you want to keep it your way, insert `skip-string-normalization = true` in the `pyproject.toml`.

Note: After using double quotes for some months, I do not understand the visual noise argument anymore. It simply does not matter. So why not go with the mass?

5.1.2 reorder-python-imports

This tool reorders import statements for better readability.

5.2 Linters

To lint a file means to check the file for errors. The errors can be stylistic errors, warning if you do not follow code conventions, etc.. One example is that unused variables like loop counters are referenced with an _ like this:

```
# Content of temp.py
for _ in range(3):
    print("Hey")
```

Rules like this can be hard to remember while linters help to conform. E.g. flake8 in its implemented version will show the following error messages if you replace _ with `a`:

```
.\temp.py:1:5: B007 Loop control variable 'a' not used within the loop
               body. If this is intended, start the name with an
               underscore.
.\temp.py:2:5: T001 print found.
```

You are encouraged to not use print in your code, but rely on a real logger like Loguru to get information on the running program.

5.2.1 flake8

flake8 is common tool to lint Python files. It does not only recognize stylistic issues which should be fixed with Black anyway, but it also makes comments on the naming of variables, suggestions for rewriting code segments and more.

5.2.2 doc8

doc8 helps you to avoid errors in restructured-text files which are hard to debug using the sphinx error log.

5.2.3 restructuredtext_lint

This tool validates your `README.rst` in case you want to publish your project as a package on PyPI.

CHAPTER 6

Documentation

The problem with guides on how to write documentations are more or less cheat sheets for restructured-text which is not what I was looking for. Therefore, here are some projects which what I consider as beautiful and informative documentations.

6.1 List of example projects

- Click is a very cool command-line interface written in Python. The documentation is convincing and presenting the capabilities of the project very well. Use the raw files also as a formative style guide.

6.2 List of Tips

- The level of headings in this documentation does not have to be very deep. Therefore, only underline headings in the following order are used: =, −, ^ and ~.

CHAPTER 7

How to debug the project

To debug our research project, we often want to run a single file within the Waf framework repeatedly or we even want to dive into the debugger if an error occurs. Normally, this is not possible as Waf controls the execution and places the `bld` or `src` on the `PYTHONPATH`. Thus, if we execute a single file, an `ImportError` is raised since `bld.project_paths` cannot be imported. Adding the paths manually seems a little bit hacky and can be circumvented elegantly. In addition to that, even if we insert a debug statement in the file and the code reaches this line, Waf hides the prompt of the debugger from the user. Then, it will silently run forever as the debugger is never closed.

7.1 Make `bld` and `src` importable

To place `bld` and `src` on `PYTHONPATH` we turn the project into a python package. This can be accomplished by placing a file called `setup.py` in the root directory of the project. This file is the entry point for every other Python package you have ever installed with `pip`. For our project, this file contains only necessary information as we will never upload our research project on PyPi. Here is what the file looks like:

```python
from setuptools import setup


setup(
    name="project_name",
    packages=["bld", "src"]
)
```

That is all. `name` is the name of the package which we can use to install or remove the package. `packages` lists directories which will be added to `PYTHONPATH`.

To install the package, we do **not** use `pip install .` as this will install the package in its current form. Instead, we would like that the installed package changes with our changes to the project. This can be done by making an editable install of the package which registers our project as a moving target. For the editable install, go into the root folder of the project where the `setup.py` lies and type

```
$ pip install -e .
```

That is all. Now, you can run every single file withing the project.

7.2 Debugging

As an example, let's say we have a file called `src/data_management/create_dataset.py` with the following content:

```python
from bld.project_paths import project_paths_join as ppj


def main():
    df = pd.read_stata(ppj("IN_DATA", "example.dta"))

    df.AGE = df.AGE.astype(int)

    df.to_pickle(ppj("OUT_DATA", "example.pkl"))


if __name__ == "__main__":
    main()
```

The file loads `example.dta`, turns variable `AGE` into an integer and saves the file as a pickle object. Assume that running the program raises an error as the variable `AGE` is not defined in the data and is instead called `ALTER` (german word for age). Then, Waf aborts the execution and returns a more or less readable report of the error which is in this case quite clear. How can we jump into the debugger to inspect the state of the program?

The first method is to insert a debug statement before the error occurs. Starting with Python 3.7 this is even more simple.

```python
...

def main():
    df = pd.read_stata(ppj("IN_DATA", "example.dta"))

    import pdb; pdb.set_trace()   # For Python < 3.7

    breakpoint()   # For Python >= 3.7

    df.AGE = df.AGE.astype(int)

...
```

Then, you can start to debug your program. For more information on how to use the Python debugger `pdb` visit this tutorial.

The second method to start the debugger is directly from the command line. Type

```
$ python -m pdb -c continue src/data_management/create_dataset.py
```

to enter the debugger if an exception occurs. If you leave out `-c continue` you will jump into the debugger right at the start.

7.3 Using a different debugger

The default debugger is not really visually appealing. Instead we can use ipdb which is the IPython debugger with tab-completion, syntax highlighting, etc.. Install it with

```
$ pip install ipdb
```

Then, use it with `import ipdb; ipdb.set_trace()` or register it as the default debugger for `breakpoint()` by setting the environment variable

```
$ export PYTHONBREAKPOINT=ipdb.set_trace  # Unix

$ $env:PYTHONBREAKPOINT="ipdb.set_trace" # Windows
```

Or just run the file with `ipdb` by running

```
python -m ipdb -c continue src/data_management/create_dataset.py
```

CHAPTER 8

Tips and Tricks for Waf

Here is a list of tips and tricks you might want to use for your research project. Some of the things are only suggestions to solve some problems, but are not further explained.

8.1 Compiling the reports with LaTeX

1. Compiling pollutes the command line interface. To shut it off, change `prompt` to 0 in `src/paper/wscript`. Unfortunately, if an error happens, you have to switch back to find the source.
2. (Windows) Sometimes changes in the dependencies of the report are not recognized and the document is not compiled. Maybe this is related to a similar issue with LatexTools. In this case, type `rm bld/src/paper` to delete all built artifacts.

8.2 Copying files

Do not use the `rule` argument with `cp` or `copy` as those operations tend out to be extremely slow. Use one of the following instead.

- To copy a file from the source to the build directory, use

```
ctx(
    features="subst",
    source=ctx.path_to(ctx, "IN_DATA", "file.pkl"),
    target=ctx.path_to(ctx, "OUT_DATA", "file.pkl"),
    is_copy=True,
)
```

- To copy a directory use `buildcopy` (Link)

8.3 Running interactive commands

Apparently, this should be possible with this. Should test it with the debug script.

8.4 Type annotations with Monkeytype

MonkeyType allows you to collect information on variable types at runtime and store the results in a SQLite database. After that, you are able apply the type annotations to your code.

For that, we need to replace plain Python as the executor of the scripts with MonkeyType in `run_py_script.py`. Of course, it is possible to create another runner named `run_monkeytype_script.py`, but it would be more tiresome. Instead locate the variable `run_str` in `class run_py_script()` and replace `${PYCMD}` with `monkeytype run`.

Running Waf will execute all scripts with MonkeyType and store the results in `bld/monkeytype.sqlite3`. Use `monkeytype apply some.module` to apply the type annotations to the module.

Maybe the procedure messes with your prepended arguments, but I have not used them so far. Feedback welcome!

CHAPTER 9

Anaconda on Windows

This cookiecutter is designed to work with Anaconda, a scientific Python distribution including its own package manager conda. Anaconda simplifies the usage and maintenance of python in particular for Windows user. However, the programming community is still extremely focused on use cases with Linux or MacOS and neglect issues on Windows as these users should switch the OS anyway :). Therefore, the following is a step-by-step installation and user guide for Anaconda on Windows.

9.1 Installation

1. Download the latest graphical installer from anaconda.org. If you do not know whether you need the 32-bit or 64-bit installer, look at this FAQ.
2. Start the installer. Choose whether to install Anaconda for all users which requires administrator privileges or for a single user. I prefer to install Python system-wide so it is available to all users. But then, you have to be careful as every time you interact with the base environment you have to use an elevated shell (a shell with administrator privileges as described in this article).

   Warning: If you install Anaconda for a single user you may run into problems when executing python from the Windows Powershell. In particular, it may happen, that you can run your Python files only from the Anaconda prompt.

3. Tick "Add Anaconda to my PATH environment variable" and also "Register Anaconda as my default Python 3.x". Finish installation.
4. Often times you have to manually add Anaconda to your PATH environment. You can find an instruction on how to do that here. If you installed Anaconda for all users, the `PATH` should read something like `C:\ProgramData\Anaconda3`.

9.2 Which console?

The Powershell is the preferred way on Windows as it provides a better interface and better tab-completion.

9.2.1 Conda Version < 4.6

Unlike CMD and "Anaconda Prompt", it is not fully supported by Anaconda. In particular, the activation and deactivation of environments is broken in older versions of conda (<4.6). To solve this issue, you have to install an additional package . If you installed Anaconda with administrator privileges, start an elevated shell. Then, type

```
$ conda install pscondaenvs -c pscondaenvs
```

Since Windows is extremely cautious, it does not want to execute the new `activate.ps1`. Thus, we need to change the execution policy in administrator mode with

```
$ Set-ExecutionPolicy RemoteSigned
```

and answer the following prompt with "Yes, for all" or `a`.

Now, if you go back to your normal Powershell, you can activate an existing environment with

```
$ activate <env-name>
```

and deactivate it with

```
$ deactivate
```

9.2.2 Conda Versions >= 4.6

Starting from version 4.6 Anaconda officially supports Powershell (see Conda 4.6 Release). Note, however, that this is still in an experimental state and issues may still occur. To initialize the use of Anaconda environments from Powershell, you have to open your Powershell and execute:

```
$ conda init
```

You then have to restart your Powershell. Now, you can activate existing environments with

```
$ conda activate <env-name>
```

and deactivate with

```
$ conda deactivate
```

9.3 How to interact with the base environment?

The base environment is activated by default. If you start a Powershell and type `python`, you are using the Python interpreter and the packages from the base environment.

My personal advice is to touch the base environment only if you want to do some small programming or prototyping. In all other cases, create a separate environment.

9.3.1 Updating conda and the package manager

Start a Powershell (with administrator privileges if you installed Anaconda for all users). Type

```
$ conda update conda
```

to update the package manager.

Warning: Be aware that sometimes the developers of conda distribute buggy versions which usually forces you to reinstall Anaconda completely. Still, I recommend to upgrade from time to time. If you are extremely cautious, check the latest versions and update only if the latest version is a week old.

Then, update Anaconda with

```
$ conda update anaconda
```

9.4 How to interact with environments?

9.4.1 Create environments

As I said before, I recommend to create a new environment for each of your projects. If you do not know which packages you need later, start with a plain Python environment and install packages along the way. Create a plain Python environment with

```
$ conda create python=3.7 -c anaconda
```

or you can create an environment from a file with

```
$ conda env create -n <env-name> -f <path-to-yml>
```

The environments are usually placed in your user folder under `C:\Users\<user-name>\.conda/envs/`, but I would not be surprised to find them elsewhere :).

9.4.2 Manage packages

If you leave out the name, conda takes the name from the `environment.yml`. If you leave out the file, conda looks for a `environment.yml` in the current folder.

To install a package type

```
$ conda install statsmodels=0.9.0
```

and to update

```
$ conda update statsmodels
```

9.4.3 Export an environment

To make your projects reproducible, you have to define an `environment.yml`.

```
$ conda env export -f environment.yml
```

Exporting the environment is one but maybe not the best way to create the environment file. I would recommend that you do it yourself and add only packages you are importing directly. The reason is that you only want to ensure that the results hold for the specific versions of the main packages and you do not care about how they are using their dependencies. An example looks like this:

```
# content of environment.yml
name: cc
channels:
    - defaults
    - pscondaenvs
dependencies:
    - pscondaenvs=1.2.4
    - python=3.7
    - pip:
      - pandas==0.24.1
```

`name` is the shortcut used to activate the environment later. `channels` contains different sources for installing packages in order. During installation conda iterates through the channels from top to bottom and looks for the specific package. In `dependencies` one can see first packages installed via conda. Notice the single equality sign to pin a specific version. Under `pip` you can see a list of packages which should be installed with pip. Here, you pin a package with two equality signs. I would recommend to install as many packages with pip as possible, e.g. pandas, but not Numpy, statsmodels, scikit-learn. First, every package is always up-to-date on PyPi, but sometimes distributing to Anaconda takes longer. Second, pyup can only inform you about updates under pip.

If you export the environment, there is a second entry after each package installed with conda.

```
dependencies:
  - vs2015_runtime=14.15.26706=h3a45250_0
```

The hash, `h3a45250_0`, makes sure that packages have the same build instructions, but they are not only compiler but also OS-specific. Thus, you cannot install a hashed package on Windows and Linux.

9.4.4 Update an environment

What if you want to update the environment because you altered the `environment.yml`?

```
$ conda env update -n <env-name> -f <path-to-yml>
```

Again, you can leave out `-n` and `-f` if the name is specified in the file or if the file is in the current directory.

9.4.5 Remove an environment

```
$ conda env remove -n <env-name>
```

The rest of the commands can be found in the official conda documentation.

9.5 Advanced tips

- At some point, you might decide that you do not need a base environment or that you would like to force yourself to create environments for each project. Then, Miniconda might be an option for you. This setup only installs the package manager, but no base environment. This also saves space on your disk.

- Often times you want to start a Jupyter notebook server while continuing to use the shell. Just type

  ```
  $ start jupyter notebook
  ```

  or

  ```
  $ start jupyter lab
  ```

  which will open a new Powershell within the same conda environment running the server. Then, you are able to install new packages from the unblocked shell, but you have to restart running kernels before packages become available.

CHAPTER 10

Todo

Currently, no features are planned, but I am open to suggestions :).

CHAPTER 11

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

11.1 Types of Contributions

11.1.1 Report Bugs

Report bugs at https://github.com/tobiasraabe/cookiecutter-research-template/issues

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

11.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement a fix for it.

11.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

11.1.4 Write Documentation

The cookiecutter could always use more documentation, whether as part of the official docs, in docstrings, or even on the web in blog posts, articles, and such.

11.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/tobiasraabe/cookiecutter-research-template/issues.

If you are proposing a new feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

11.2 Get Started!

Ready to contribute? Here's how to set up *cookiecutter-research-template* for local development. Please note this documentation assumes you already have *virtualenv* and *Git* installed and ready to go.

1. Fork the *cookiecutter-research-template* repo on GitHub.
2. Clone your fork locally:

   ```
   $ cd path_for_the_repo
   $ git clone git@github.com:YOUR_NAME/cookiecutter-research-template.git
   ```

3. Assuming you have conda installed, you can create a new environment for your local development by typing:

   ```
   $ conda env create -n cc -f environment.yml
   ```

4. Install automatic quality checks with pre-commit by running

   ```
   pre-commit install
   ```

5. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

6. When you're done making changes, check that your changes pass the defined tests.

   ```
   $ tox
   ```

7. If your contribution is a bug fix or new feature, you may want to add a test to the existing test suite. See section Add a New Test below for details.
8. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

9. Submit a pull request through the GitHub website.

11.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in `README.rst`.
3. The pull request should work for Python 3.5, 3.6 and 3.7 on Windows, Linux and MacOS. Check the test panel in the pull request and the logs of the test if the tests are failing.

11.4 Add a New Test

When fixing a bug or adding features, it's good practice to add a test to demonstrate your fix or new feature behaves as expected. These tests should focus on one tiny bit of functionality and prove changes are correct.

To write and run your new test, follow these steps:

1. Add the new test to *tests/test_cookie.py*. Focus your test on the specific bug or a small part of the new feature.
2. If you have already made changes to the code, stash your changes and confirm all your changes were stashed:

   ```
   $ git stash
   $ git stash list
   ```

3. Run your test and confirm that your test fails. If your test does not fail, rewrite the test until it fails on the original code:

   ```
   $ pytest
   ```

4. (Optional) Run the tests with tox to ensure that the code changes work with different Python versions:

   ```
   $ tox
   ```

5. Proceed work on your bug fix or new feature or restore your changes. To restore your stashed changes and confirm their restoration:

   ```
   $ git stash pop
   $ git stash list
   ```

6. Rerun your test and confirm that your test passes. If it passes, congratulations!

CHAPTER 12

Credits

12.1 Development Lead

- Tobias Raabe

12.2 Contributors

- Radost Holler (documentation)