Code development with Python

Marc van der Sluys and Paul van Kan
HAN University of Applied Sciences
Arnhem, The Netherlands
http://han.vandersluys.nl

August 14, 2020

Contents

1 Introduction **3**

2 Creating Python modules **4**
2.1 Python module consisting of a single file 4
2.2 Python module consisting of multiple files in a directory 4

3 Creating and publishing Python packages **5**
3.1 PyPi/pip package 5
3.1.1 Summary 5
3.1.2 Setuptools / setup.py 5
3.1.3 setuptools_scm 7
3.1.4 Name of the package 7
3.1.5 Test installation 8
3.1.6 Importing the package 8
3.1.7 Upload the package 8
3.2 Installing a local PyPi package with pip (i.e., not downloaded from cloud) 9
3.3 __init__.py 9
3.3.1 Module consisting of multiple files 9

4 Python documentation **11**
4.1 Information on existing Python packages 11
4.2 Docstrings 11
4.2.1 Description of docstrings 11
4.2.2 Language 12
4.2.3 Docstring styles 12
4.2.4 Example 12
4.3 Generating docstrings from Python code 13
4.3.1 Pyment 13
4.3.2 Emacs 13
4.4 Creating documentation from Python docstrings 13
4.4.1 Pdoc3 13
4.4.2 Sphinx 14
4.4.3 Doxygen 15
4.4.4 Read the Docs 15
4.4.5 GitHub pages 16

5 Web apps with Python **17**
5.1 Embedding Python code in Azure web apps 17

6 Data files **18**
6.1 Comma-separated values 18

7 Workflow on GitHub **19**
7.1 Preparation 19
7.2 Planning/design 20
7.3 Work cycle on your PC 21
7.4 Finalising your work 21

Chapter 1

Introduction

This document describes the Python coding style used for the MMIP heat pump configurator project.

Chapter 2

Creating Python modules

The information in the Sections below was gathered from a short [1] and a long [2] answer to a question on Stack Overflow.

2.1 Python module consisting of a single file

The rules are:

1. A module is a file containing Python definitions and statements.
2. The file name is the module name with the suffix `.py`.
3. The file name consists of lower-case characters. Use underscores to improve readability.

Example: create `hello.py`:

```
def helloworld():
    print "hello"
```

In the same directory, you can now create `test_hello.py`:

```
import hello
hello.helloworld()
```

and run it.

2.2 Python module consisting of multiple files in a directory

1. To group many `.py` files, you can put them in a folder. Any folder with a file called **`init.py`** is considered a **package module** by Python. The `.py` files in the directory can then be called a **package**.
2. To `init.py`, add:

   ```
   name = "<pkg_name>"
   ```

3. Rules: use all lower-case characters, use underscores to improve readability.

Chapter 3

Creating and publishing Python packages

- http://devarea.com/deploying-a-new-python-package-to-pypi/

3.1 PyPi/pip package

- SO Summary
- Tutorial
- Pypi test repo

3.1.1 Summary

1. Create/edit `setup.py` Set/change version in `setup.py`!
2. Git commit
3. Create package:
   ```
   $ ./setup.py sdist bdist_wheel
   ```
4. Test:
   (a) List Python files:
   ```
   $ tar tfz `ls -rt dist/*.tar.gz |tail -1` | grep -E "\.py"
   ```
   (b) Check (README):
   ```
   $ twine check dist/*
   ```
   (c) Install locally:
   ```
   $ pip3 install --user `ls -rt dist/*.whl |tail -1`
   ```
5. Upload:
   ```
   $ twine upload -u <user> -p <pwd> --skip-existing --verbose dist/*
   ```

3.1.2 Setuptools / setup.py

1. Put a file setup.py in the root dir of the project, the parent dir of the subdir containing the .py files.
   - See file:///home/sluys/diverse/software/ELP-MPP02/setup.py for an example.

- Writing the Setup Script - Python documentation

```
#!/bin/env python3

version="0.0.1"

import os
#os.system('rm -rf *.egg-info/')        # Make 'really clean'

# Prevent the setuptools_scm plugin from adding (only) the contents of the
    git repo to the tarball:
os.system('mv -f .git .git_temp')

with open("README.md", "r") as fh:
    long_description = fh.read()


from setuptools import setup
setup(
    name='elp-mpp02',
    description='Accurate Moon positions using the Lunar solution
        ELP/MPP02 in Python',
    author='Marc van der Sluys',
    url='https://github.com/MarcvdSluys/ELP-MPP02',

    packages=['elp_mpp02'],
    install_requires=['numpy','fortranformat'],
    long_description=long_description,
    long_description_content_type='text/markdown',

    version=version,
    license='GPLv3+',
    keywords=['Moon','Astronomy','Ephemeris'],

    classifiers=[
        "Development Status :: 3 - Alpha",
        "Intended Audience :: Developers",
        "Intended Audience :: Education",
        "Intended Audience :: Science/Research",
        "License :: OSI Approved :: GNU General Public License v3 or later
            (GPLv3+)",
        "Natural Language :: English",
        "Operating System :: OS Independent",
        "Programming Language :: Python :: 3",
        "Topic :: Scientific/Engineering :: Astronomy",
    ]
)

# Put git repo back:
os.system('mv -f .git_temp .git')

# Do some basic checks:
print("\nPython source files included in tarball:")
os.system('tar tfz dist/elp-mpp02-'+version+'.tar.gz |grep -E "\.py"')
print()

os.system('twine check dist/elp-mpp02-'+version+'.tar.gz')
os.system('twine check dist/elp_mpp02-'+version+'-py3-none-any.whl')
print()
```

- Note:
  - author_email: leave out because of privacy. setup.py will warn, but upload suc-

ceeds. With a dummy address, upload fails

- install_requires=['pkg1','pkg2'], # Depencencies
- classifiers: https://pypi.org/pypi?:action=list_classifiers

2. Add LICENCE to root dir
3. Add README.md to root dir
   - run setuppy2READMEmd
   - Example README.md
   - M-x gh-md-render-buffer
   - see jotter markdown
     - use twine check dist/* after packaging to check README
     - Pypi-friendly readme - useful?
4. Create a source and wheel distribution:

   ```
   ./setup.py sdist bdist_wheel
   ```

   - needs: pip3 install wheel
5. Check package Setup.py can also upload to Pypi, but use Twine instead:

   ```
   twine check dist/*  # For a basic check (of the README only?)
   twine upload  # for usage info
   ```

3.1.3 setuptools_scm

- NOTE: setup.py ignores its file settings in setuptools.setup() if the setuptools_scm plugin is installed on your system, a git repo is present in the current dir. It simply adds ALL and ONLY THOSE files that are in the repo.
  - Work-around:

```
from setuptools_scm import integration
integration.find_files = lambda p: []
```

  - NOTE however, that setuptools_scm must be installed when installing the package using pilocate p!
  - Hence, instead use:

```
import os
os.system('mv -f .git .git_temp')
# ...
os.system('mv -f .git_temp .git')
```

3.1.4 Name of the package

- The name of the package appears in (at least) five different places:
  1. the name of the directory where the Python files/modules are located
  2. in name="¡name¿" in setup.py
  3. in the **init.py** in the package dir

4. in the packages="" entry in setup.py
5. in the dist file/tarball name

- Here:
  1. is used to **import** the package. It is also printed as .__name__ after import. It must be equal to 4)
  2. seems of little consequence for simple use (but may be important for dependencies?). It is printed as .name, not as .__name__!
  3. is used to **install** the package. It is equal to 5)
  4. must be equal to 1), since it specifies where the packager must look for Python/module files
  5. is set from 3)

3.1.5 Test installation

1. tar tfz 'ls -rt dist/*.tar.gz —tail -1' — grep -E "\.py" # List all Python source files included
2. twine check dist/* # For a basic check (of the README only?)
3. pip3 install –user 'ls -rt dist/*.whl —tail -1' # Install locally

3.1.6 Importing the package

- For a package with structure package/module.py:

```
from package import module
module.function()

import package.module as mod
mod.function()
```

- This does **not** work - why?:

```
import package
package.module.function()  # AttributeError: module 'package' has no
    attribute 'module'
```

- Leaving out the dir package/ from the example above and putting module.py in the root dir would put that file **in** the site-packages/ directory, rather than in a subdir!

3.1.7 Upload the package

To PyPI Testing

- twine upload –repository-url https://test.pypi.org/legacy/ -u ¡user¿ -p ¡pwd¿ –skip-existing –verbose dist/*
  - this returns an url where the package can be viewed

To PyPI Live

- twine upload -u ¡user¿ -p ¡pwd¿ –skip-existing –verbose dist/*

3.2 Installing a local PyPi package with pip (i.e., not downloaded from cloud)

- need setup.py
- from the package directory, do:
  - pip3 install -e . –user
  - **Note**: this installs the latest packaged version, rather than the current source code?
    - not always true...; when no packages are present, the subdir ¡package¿.egg-info is created in the dev dir
      - in ~/.local/lib64/python3.6/site-packages/ ¡package¿.egg-link is created, with a link to the dev dir
      - remove this file after installing the package
    - the package seems to be preferred if both the link and the package are present in ~/.local/lib64/python3.6/site-packages/
    - weird: after manually removing the package and the link, the dev dir is still found!

3.3 __init__.py

- Source
  - see also: SO
- a module is basically a Python file from which a function can be `import`ed.
  - the name of the module is that of the Python file, without the extension `.py`. *E.g.* for `module.py`:

    ```
    import module
    ```
- a package consists of a (number of) module(s) with a directory hierarchy and an `__init__.py`.
  - *e.g.* for `package/module.py`:

    ```
    import package.module
    ```

3.3.1 Module consisting of multiple files

```
someDir
|-- modName
|   |-- __init__.py
|   |-- srcFile1.py
|   |-- srcFile2.py
|   `-- srcFile3.py
`-- example.py
```

__init__.py

```
from .srcFile1 import fun1
from .srcFile2 import fun2
from .srcFile3 import fun3
```

example.py

```
import modName

y1 = modName.fun1(x)
y2 = modName.fun2(x)
y3 = modName.fun3(x)
```

Chapter 4

Python documentation

This chapter is about obtaining information on Python packages, but more importantly about adding documentation to the source code in such a way that it can be used in the code itself, as well as in a code generator that takes a Python source file as input and *e.g.* PDF, Markdown or HTML as output.

4.1 Information on existing Python packages

Online information on installed Python packages can be obtained using the `pydoc` command in the shell, similar to man pages. `pydoc` can also show the documentation contained in a Python file to screen (`pydoc ./file.py`) or generate html `pydoc -w <module or file>`. See `pydoc pydoc` for more information.

4.2 Docstrings

Introductory information on docstrings can be found in [3].

4.2.1 Description of docstrings

Python docstrings are the default way of documenting Python code. They associate documentation with Python modules, functions, classes, and methods (=function in class). The following rules apply:

- A docstring is enclosed by three double quotes `"""` and can contain hard returns.
- The contents of a docstring begins with a capital letter and ends with a period.
- The first line of a docstring should be a short description.
- If a docstring consists of multiple lines, the second line should be empty:

```
"""This module solves all issues related to heat pumps.

More explanation..."""
```

- An object's docstring is the first statement in the object's definition.
- A module's docstring is the first statement in the module's file.
- All functions should have a docstring.
- Docstrings can be used by an interactive help system, or as metadata:

```
import myModule
help(myModule)
help(myModule.myFunction)
```

- Docstrings can be accessed by the `doc` attribute on objects:

```
print(myFunction.__doc__)
```

4.2.2 Language

In order to make our code as portable as possible, we will write documentation in English.

4.2.3 Docstring styles

The following docstring styles exist:

javadoc:	`@param` style	Does not work well with `pdoc3`.
reST:	`:param x:` style.	Most widely used. Default for Sphinx. Hard to read in code. Not recognised by `pdoc3`.
numpydoc:	`---` and `:`	Readable in code. Does not work well with `pdoc3`.
google:	`var: ...` style	Indented. **Looks good** in source and `pdoc3`. Can be used by Sphinx with the Napoleon plugin.

We will use the **Google** style, since it is easy to read in the code and is supported by both Sphinx (and hence *Read the Docs*), `pdoc3` and Doxygen (see Section 4.4.3).

4.2.4 Example

Code listing 4.1 shows a docstring example with single and multiple return values.

Listing 4.1: Docstring example with single and multiple return values and two types of documentation for body statements. Adapted from [4].

```
def my_function(arg1):
    """
    Summary line.

    Extended description of function.

    Parameters:
      arg1 (int):    Description of arg1.
      arg2 (float):  Description of arg2.

    Returns:
      int:  Description of return value

    Returns:
      tuple (float,float):  Tuple containing (rv1, rv2):

        - rv1 (float):  Description of return value 1
        - rv2 (int):    Description of return value 2

    References:
      - Some paper

    """

    var1 = 1
    """Docstring for var1"""
```

```
27
    var2 = 2;   """Docstring for var2"""
29
    return var1 + var2
```

Note that:

1. This example is both for a single return value and for multiple return values. In reality, you should of course choose **one** of them.
2. All **empty lines** are necessary.
3. All **indentations** are intended (as is the pun) and necessary.
4. Only **one** space in "x (float)".
5. The **colons** define section headers.
6. There is no way to indicate **multiple return values** (since they form one tuple). The second solution above seems to work nicely (in Pdoc3 and Sphinx+Napoleon). The bullet list **needs** the empty line and indentation!
7. In the code body, put the docstring on the line after a variable declaration, or on same line after a semicolon.

4.3 Generating docstrings from Python code

4.3.1 Pyment

If you are borrowing existing code without documentation, `pyment` can help to generate a base version.

- In a source directory, do
  ```
  $ pyment -o google -w .
  ```
- This keeps existing documentation, adds new if possible. `pyment` can convert between styles.
- By default, `pyment` writes a patch; use `-w` to overwrite the file.
- It is recommended to run `pyment` on a separate file, before adding the code to out module.

4.3.2 Emacs

In Emacs, a Google-style docstring template can be set up using `yasnippet`. Ask MvdS if you're interested.

4.4 Creating documentation from Python docstrings

An overview of tools for Python documentation can be found in the Python Wiki [5].

4.4.1 Pdoc3

The program `pdoc3` [6] can be used to generate simple, nice, lightweight and small HTML pages from a Python module. Google-style docstrings are preferred for compatibility with `pdoc3`.

Note: ensure that you use `pip` to install `pdoc3`, not `pdoc`!

Example use:

```
$ cd ~/work/HistoricalAstronomy/HistAstro/Python/histastro
$ pdoc --html --output-dir docs histastro
```

- Use `--force` to overwrite existing output.

The following issue may occur:

```
ERROR: No module named ...
$ export PYTHONPATH="." # Was PYTHON_PATH Use both?
$ export PYTHONWARNINGS='error::UserWarning'
```

4.4.2 Sphinx

The program **Sphinx** [7] is widely used to generate documentation from docstrings. It can generate (bloated!) HTML, (pdf)LaTeX, and more, and is compatible with *Read the Docs*.

Its default docstring format is reST (reStructured Text), but it can handle Google-style docstrings using the Napoleon plugin [8].

The information below is adapted from [9]. A $ indicates a shell command. You can use your favourite text editor whenever `emacs` is used.

1. install Sphinx
2. `$ mkdir docs && cd docs/ # From your Python root directory`
3. `$ # sphinx-quickstart # If you want`
4. `$ sphinx-apidoc -ef -o .  ../<package>/`
   - `-o DESTDIR, --output-dir DESTDIR`: directory to place all output
   - `-e, --separate`: put documentation for each module on its own page
   - `-f, --force`: overwrite existing files
   - `-F, --full`: generate a full project with sphinx-quickstart
     - `-a, --append-syspath`: append `module_path` to `sys.path`, used when `--full` is given
     - see `--help` for more options with `--full`
5. `emacs Makefile`
   (a) replace `BUILDDIR = _build` with `BUILDDIR = build`
   (b) Note: `make clean` cleans BUILDDIR, and setting `BUILDDIR = .` removes all configuration!
6. `emacs conf.py`
   (a) `sys.path.insert(0, os.path.abspath('../'))`
   (b) `source_suffix = ['.rst', '.md']` ?
   (c) `html_theme = 'sphinx_rtd_theme'`
   (d) Use the Napoleon plugin for Google (or Numpy) style:

```
extensions = [
   \ldots{}
'sphinx.ext.napoleon',
```

```
]

napoleon\textsubscript{google}\textsubscript{docstring} = True
napoleon\textsubscript{use}\textsubscript{param} = False
```

7. `$ emacs index.rst`
   (a) readme (inside toctree)
   (b) <nameOfYourModule>
8. docs/readme.rst:

```
HistAstro ReadMe
================
.. include:: ../README.md
```

9. `$ rm modules.rst` ?

4.4.3 Doxygen

Doxygen is a widely used documentation system for many languages, including Python [10]. When using `doxypypy` as a filter, it is supposed to be capable of handling Google-style docstrings [11]. However, we have not tested this and we will not use it as a primary tool here.

4.4.4 Read the Docs

Read the Docs (RtD) is a free (albeit commercial) cloud service to publish technical documentation. It offers (semi)automatic building, versioning and hosting. Documentation can automatically be rebuilt after a GitHub push using a webhook as described below.

Add RtD support to the git repository

To use the GitHub-RtD webhook, the following files should be added to your git repository.

- `.readthedocs.yml`
- `docs/*.rst`
- `docs/conf.py` ?

The format of `.readthedocs.yml` is described by [12] (the URL provided and the next page linked from there).

Manually creating a GitHub webhook

While it is possible to give RtD (full) access to you GitHub repository, we find that too much power for any service is undesirable. It is easy to create the RtD webhook in GitHub manually [13].

1. In the RtD project: in Admin > Integrations > Add integration, select GitHub incoming webhook
   (a) Copy the URL (looks like `readthedocs.org/api/v2/webhook/histastro/100519/`)
2. In the GitHub project: Settings page for your project > Click Webhooks > Add webhook
   (a) Payload URL = RtD URL with `https://`
   (b) Content type = application/x-www-form-urlencoded

(c) Secrets: blank

(d) Let me select individual events

i. Branch or tag creation

ii. Branch or tag deletion

iii. Pushes

(e) Active

(f) Add webhook

After this, your RtD documentation should be rebuilt shortly (seconds?) after a git push.

4.4.5 GitHub pages

We should consider GitHub pages as an alternative[14] to Read the Docs.

Chapter 5

Web apps with Python

5.1 Embedding Python code in Azure web apps

Link for Python script to Azure Function: [15]

Chapter 6

Data files

To store and exchange data, we will try to use the comma-separated-values format as much as possible. One-off input data in different formats may be converted to csv by a separate Python tool.

6.1 Comma-separated values

The csv-format is the de facto standard in the open community. It is easy to read by eye and by a machine, where the former is useful in the development of a routine to do the latter.

Csv files are plain text (ASCII), in our case presumably mostly consisting of numbers, ordered in rows and columns. The international standard specifies the **comma** as a separator for columns [16]. The decimal **point** is used as a decimal separator.

Chapter 7

Workflow on GitHub

Note: this is a raw export from Org mode, based on GitHub guide: GitHub workflow

7.1 Preparation

1. Create a **Milestone**
   - this is a (long-term) goal
     - e.g. for a next software release
   - the Milestone can be reached in multiple steps, called **Issues**
   - a due date is optional
2. Create a **Project**
   - this really means **Workflow**!
   - uses the Kanban method
     - should choose the **Automated Kanban with code review** template
   - this uses **columns** to represent the work flow (from left to right), e.g.
     (a) To do
     (b) In progress
     (c) Review in progress
       - if "with code review" was chosen

     (d) Reviewer approved
       - if "with code review" was chosen

     (e) Done
   - **Issues** are represented in the Project as **Cards**
     - they move from left to right through the Project/workflow/columns
   - not all Cards are Issues
     - there can also be Notes
     - special Note Cards are the Automation Rules Cards

    - they help to sync between Milestones/Issues and Projects/Cards
    - these should always be at the bottom (or top) of a column to work!
    - they are added once by the admin and stay forever

7.2 Planning/design

1. Create an **Issue**
    - to implement a feature or fix a bug:
        (a) in the **Issues** tab, click the green **New issue** button
        (b) provide a short, descriptive **name** (it will also be used for the branch name)
        (c) provide more details in the **comment** box
            – a **task list** (created with `* [ ]`) is very useful here for subissues; you can tick them off on the issue page, and the progress is visible on the issue overview page
        (d) add a **Milestone** and a **Project** on the right
        (e) if you want, **assign** the Issue to someone immediately (on the right)
            – in this case, a Branch will also be created (see below)
        (f) click the green button **Submit new issue**
    - a new **Card** will be created in the **To do** column of the Project
2. **Assign** an Issue / create a **Branch**
    - an Issue is assigned **to someone**, who will carry out or coordinate the work
    - a **Branch** is automatically created when assigning an Issue
        – when using the create-issue-branch app
        – the branch will get a descriptive name: issue-#-$\text{Name}_{\text{ofIssue}}$
    - this will also move the Issue **Card** from **To do** to **In progress**
    - this is a **development (Active) branch**, as opposed to the **master (Default) branch**
        – use it to implement a new feature or fix a bug
        – use a descriptive short name (automated here to use the name of the Issue)
        – you can experiment without affecting the master branch
        – each Branch should correspond to an Issue (automated here when assigning the Issue)
3. **Pull** your newly assigned Branch to your **PC**
    - the new Branch is created locally on your PC

7.3 Work cycle on your PC

1. **Pull** from GitHub to your **PC**
   - especially when collaborating with others on the same Branch!
2. **SWITCH TO THE CORRECT BRANCH!!!**
   - `git checkout <branch_name>`
   - if you forget this, you will be committing to the master branch!
3. Add **commits**
   - make your changes
   - before you commit, check again: are you on the correct branch?
   - write clear commit messages
     (a) short header line
     (b) empty line
     (c) more detailed explanation if desired
4. **Push** to GitHub
   - it is a good idea to **pull and merge** the master branch into your development branch from time to time. This will ensure that your development branch stays close to the master branch and minimise the possibility of conflicts when merging your branch back into the master branch in a pull request later on. Pulling and merging the master branch is in fact only necessary after a merge from another development branch (since those are the only instances when the master branch is changed), but it is a good habit to do a pull and merge after each push.
     (a) make sure you are in your **development branch**
     (b) **pull** the whole repository, including the master branch, from GitHub
     (c) **from the development branch**, **merge** the master branch into your development branch
       – it is **very important** to merge in **this direction**; you want to update your development branch, *not* the master branch

7.4 Finalising your work

1. Open a **Pull Request (PR)**
   - to:
     – share ideas
     – ask for help or advice
     – ask for code review
     – with the eventual goal of merging your Branch back into the master branch and resolving the Issue

- you can create a PR from the **Code** tab, the **Pull requests** tab by clicking on a green **Compare & pull request** button, or from the **Code / Branches** subtab by clicking on a **New pull request** button
- you end up on the page **Comparing changes** or **Open a pull request**
  - the top of the page will show **Able to merge** in green if automatic merging is possible
  - below, it will show what has changed since the Branch was created
  - to create the PR:
    (a) in the comment box, add a line saying **This should resolve #4**
      * very important: "resolve #N" - this links the PR to the Issue!
      * when you type **'#'**, a list pops up - the Issues are marked by an exclamation mark in a circle, the PRs don't
      * if you forget this, go back to the PR after creating it and select the corresponding Issue under Linked issues on the right
    (b) on the right, add the **Project** and **Milestone**
    (c) if you want the code to be reviewed, you can add (a) **Reviewer(s)** on the right
    (d) click the green **Create pull request** button to continue
      * you may be able to update any comments and may have to click the Create PR button again
- the PR now sits under the PR tab and is ready for review
- the Issue Card will move from **In progress** to **Review in progress**

2. Discuss and **review** the code
   - the **Pull requests** to be reviewed can be found under the PR tab
   - ask others to review your code by selecting an existing PR clicking on Reviewers on the right
     - check the functionality of the code
     - check the interface
     - check the coding style
     - give useful feedback
   - go back to the work cycle on your PC to apply changes if needed
   - once the changes have been approved, the Issue Card will move from **Review in progress** to **Reviewer approved**
3. **Merge** the PR
   - note: closing a PR means **rejecting** it, which is typically *not* what you want
   - check again whether the corresponding Issue was added to the PR
   - just above the green Merge PR button is an indication as to whether automatic merging is possible

- click **Merge PR** to approve the changes and merge them into the **master/default-/base branch**
- options (need to figure out the differences - use the default)
  - Create a merge commit (default)
    - all commits from this branch will be added to the base branch via a merge commit
    - this should now also automatically close the corresponding Issue *[2020-04-19 Sun]*
      - when using the default Branch name generated from the Issue name
      - even if "resolve #1" or similar isn't used
    - see: GitHub help: closing issues via commit messages
  - Squash and merge
    - the N commits from this branch fill be added to the base branch
  - Rebase and merge
    - the N commits from this branch fill be rebased and added to the base branch
- after clicking **Confirm merge**, and if the PR is successfully merged, you can safely **Delete** the Branch to clean up, or leave it to keep history
- this should also move the Issue Card from **Reviewer approved** to **Done**
  - although it seems to get stuck on **In Progress**?

4. **Tidy up** on your **PC**
   - go back to the **master** branch: `git checkout master`
   - note that the master branch on GitHub has been updated: `git pull`
   - you may want to remove your local copy of the branch you just finished: `git branch -d <branch_name>`

Bibliography

[1] **Anuj on Stack Overflow**. *How to write a Python module or package?*, 2013. URL https://stackoverflow.com/a/15747198/1386750. Visited 2020-02-12.

[2] **Arcseldon on Stack Overflow**. *How to write a Python module or package?* URL https://stackoverflow.com/a/33770042/1386750.

[3] **Python for beginners**. *Python Docstrings*, 2013. URL https://www.pythonforbeginners.com/basics/python-docstrings/. Visited 2020-02-12.

[4] **Mor22 on Stack Overflow**. *Can Sphinx Napoleon document function returning multiple arguments?*, 2015. URL https://stackoverflow.com/a/29343326/1386750. Visited 2020-02-12.

[5] **Python Wiki**. *DocumentationTools*, 2003. URL https://wiki.python.org/moin/DocumentationTools.

[6] **Kernc**. *Pdoc3*, 2013. URL https://pypi.org/project/pdoc3/.

[7] **Georg Brandl and the Sphinx team**. *Sphinx – Python Documentation Generator*, 2007. URL https://www.sphinx-doc.org.

[8] **Ruana, R.** *sphinx.ext.napoleon - Support for NumPy and Google style docstrings*, 2007. URL https://www.sphinx-doc.org/en/master/usage/extensions/napoleon.html.

[9] **Nicholls, S.** *An idiot's guide to Python documentation with Sphinx and ReadTheDocs*, 2016. URL https://samnicholls.net/2016/06/15/how-to-sphinx-readthedocs/. Visited 2020-02-12.

[10] **Heesch, van, D.** *Doxygen*, 1997. URL http://doxygen.nl/.

[11] **Brown, E.W.** *Doxypypy*, 2013. URL https://pypi.org/project/doxypypy/.

[12] **Read the Docs, Inc & contributors**. *Configuration File*, 2010. URL https://docs.readthedocs.io/en/stable/config-file/.

[13] ——. *Webhooks*, 2010. URL https://docs.readthedocs.io/en/stable/webhooks.html.

[14] **GitHub, Inc.** *GitHub Pages*, 2020. URL https://pages.github.com. Visited 2020-02-12.

[15] **Microsoft**. *Python Developer Reference for Azure Functions*, 2020. URL https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference-python. Visited 2020-05-28.

[16] **Wikipedia contributors**. *Comma-separated values — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Comma-separated_values, 2020. Visited 2020-03-04.