Applying Best Practices for Sustainable Code: Workshop Brief

Scenario

You've inherited code from a post-doctoral researcher who has since left your group. The PI of your research group wants to publish a paper along with the code used to generate the analyses, with the hope that other researchers may apply the analysis to their own datasets and extend the capabilities of the project to other analyses.

Your task is to prepare the code for publication, applying project management practices and coding design principles concerning documentation, formatting, refactoring, and publishing, to enhance the lifecycle of the code.

Your PI is unusually proactive and well-informed about best-practices in research software engineering and has **provided you with the following checklist** to prepare the project for publication. The checklist is divided into tasks concerning documentation, formatting and refactoring, and publication.

In your pre-assigned groups, you will work together to prepare the project for publication, **dividing the following tasks amongst the members of your group**. Feel free to pair up on tasks that interest you or have relevance for your own work, or to work solo. You do not have to complete all tasks in the time available and tasks beginning with the phrase "*Essential*" should be prioritised. The remaining tasks are nice to have in a research software project, but you may decide which of these tasks are the most important for the project at hand.

If you have any issues accessing or editing the project, please refer to the *Working on the Workshop Repository* guide or, if you're still having trouble, ask a workshop helper for assistance.

Tasks: Documentation

❑ *Essential:* Add descriptive comments

- **Description:** Comments should be useful and informative to future developers of the project. They can explain the overall outline of the code, describe specific intent of certain sections of the code, and explain specific algorithmic decisions. In Python, comments begin with a hash (#) symbol on each line of the comment.
- **Task:** Comments are provided throughout the project, but there are instances where comments are missing (indicated by the placeholder comment

"`Descriptive comment`"), the comments are not sufficiently descriptive, or the formatting of comments is inconsistent. Step through the notebook and add or edit comments throughout to explain specific lines and blocks.

- **More information:** https://realpython.com/python-comments-guide/

❑ *Essential:* Add docstrings

- **Description:** In Python, the initial comment in a function or script that describes the objectives and interface is referred to as a *docstring*. The docstring describes the purpose, parameters, and return values of the function or script. Python includes a built-in function `help()` that prints the docstring for the input to `help()` to the console, so docstrings should ideally contain all information that will help guide a user in using the function or script. Docstrings are denoted by three quotation marks (""") before and after the docstring and can span multiple lines.
- **Task:** Include a docstring at the beginning of the main script and at the beginning of each function. The docstrings should describe the objective, interface (the expected inputs and outputs), and specific implementation.
- **More information:** For more guidance on how to write docstrings and examples of docstrings, see this tutorial: https://www.dataquest.io/blog/documenting-in-python-with-docstrings/

❑ *Essential:* Add a README file

- **Description:** A README file describes the purpose and components of a software project and provides potential users with instructions on how to install and run the software. The file will also list the current contributors to the project, how others can contribute to the project, and where to find relevant resources. On GitHub, the README file also acts as the landing page for the repository project and will be the first thing that any visitors to the repository will see.
- **Task:** Edit the provided `README.md` file for the project to describe how the components of the project fit together. Include stepwise instructions on downloading and running the project and how to test the project output using the provided test data file `test_data.txt` located in the `data` directory. Also include a message encouraging others to contribute to the project and outlining how contributions can be made. Use the following template to organise the contents of the README: https://ha0ye.github.io/CW21-README-tips/template_README.html

- **More information:** https://book.the-turing-way.org/project-design/project-repo/project-repo-readme

❑ *Essential:* Add a requirements.txt file

- **Description:** The `requirements.txt` file lists the packages that the project depends on for proper execution and makes installation of these dependencies easy using the "`pip install requirements.txt`" command. A `requirements.txt` file reduces the likelihood of compatibility issues and ensures that a project is well-documented, maintainable, and reproducible.
- **Task:** Create a `requirements.txt` file in the top-most directory of the project. Populate this file with a list of Python packages that the program relies on and make sure that you include only packages that are actually used by the program.
- **More information:** https://www.geeksforgeeks.org/how-to-create-requirements-txt-file-in-python/

❑ Create a tutorial notebook

- **Description:** Code packages often come with a set of instructions on how to install and use the package on an example dataset. By providing an example of how to run the provided code, the project outputs will be easily replicable and future users will be more likely to use and cite the project.
- **Task:** Create a jupyter notebook file named "`example.ipynb`". In this file, include an example for how to use the code in its new script form, including what input and output arguments are expected and how the results are saved. Include details of how to run the tutorial notebook in the README file and have a different member of the team test the instructions to make sure they work and are easy to follow.
- **Task Dependencies:** This task relies on the following tasks to be completed prior to beginning this task:
    - Documentation: Add a README file
    - Formatting & Refactoring: Translate the notebook into an executable python script

❑ Go through a software quality checklist

- **Description:** Software quality checklists can help you write good quality software and align software quality standards across software projects. They also help others who are viewing or contributing to a project understand the

state of the code and what could still be improved. A software quality checklist is an assessment of the current state of the code, rather than a list of tasks that should be completed before reporting the results of the checklist.

- **Task:** Go through the following software quality checklist and evaluate the current state of the software project: https://fairsoftwarechecklist.net/v0.2/. When you are finished, include the checklist as part of the README, in its own section.
- **More information:** Other software quality checklists and an explanation for their use can be found here: https://fair-software.nl/recommendations/checklist
- **Task Dependencies:** This task relies on the following tasks to be completed prior to beginning this task:
- Documentation: Add a README file

Tasks: Formatting & Refactoring

❑ *Essential:* Improve formatting

- **Description:** Code can become considerably more readable with the addition of blank lines that group lines of code into logical sections. Although badly formatted code will still run, it is very difficult for others to read and interpret and thus limits the likelihood that others will use and extend the code.
- **Task:** First, review the code to determine the main functionality that is implemented. Then, use blank lines to group lines of code with common functionality and visually separate sections of the code that perform different tasks.

❑ *Essential:* Improve variable and function naming

- **Description:** Variable and function names should succinctly indicate what a function does or a variable means. When variable and function names are uninformative, code can be considerably harder to understand. As a rule of thumb, the length of the name should be proportional to the scope and complexity of the variable or function, and formatting conventions (such as using `snake_case` or `camelCase`) should be consistent throughout the project.
- **Task:** Locate the lines indicated by "TODO Naming" and modify the relevant variable and function names for clarity and consistency. Make sure variable and function names throughout the code use a consistent style and are adequately

descriptive. There are additional unmarked sections where variable names could be improved – challenge yourself to find them all!

- **More information:** The python style guide (PEP 8: https://peps.python.org/pep-0008/) provides rules for consistent formatting, including use of blank space, naming conventions, and comments, and is generally followed by production-level projects.

❑ *Essential:* Fix non-DRY code

- **Description:** DRY stands for Don't Repeat Yourself. DRY code is streamlined to remove code repetitions, for instance when multiple lines could be better implemented in a single line by efficiently using existing function calls, by using a loop, or by creating a new function, and considerably improves readability and clarity.
- **Task:** Locate the lines indicated by "TODO DRY" and modify these sections to remove repetition by taking advantage of existing code, adding in a function call, or using a loop, as appropriate.

❑ Translate the notebook into an executable python script

- **Description:** To facilitate reproduction and enable future extensions to the project, the components of the Jupyter notebook can be repackaged into an executable python script.
- **Task:** Convert the Jupyter notebook into an executable python script that contains a `main()` function. All code that is not already inside of a function should be placed inside the `main()` function. Be careful to include all variable definitions inside the functions where they are used.
- **More information:** https://realpython.com/python-main-function/

❑ Refactor the script into functions

- **Description:** Each function should accomplish one logical task, enabling the script to read like a series of instructions. Smaller functions are also easier to read and understand, providing greater clarity for future developers.
- **Task:** Identify the key components of the project and factor out main functionality into separate functions. As a hint, recall the main functionality that is expected in text processing applications.

❑ Input arguments: Allow for flexible input dataset

- **Description:** Executable scripts allow for flexible processing and code reuse through the use of input arguments. By changing the script to accept input arguments, the analysis could be easily applied to other collections of files.
- **Task:** Locate the lines indicated by "TODO Inputs" and change the script to accept different inputs, such as a single file, a list of file locations, or a directory containing multiple files. All lines indicated by the comment "TODO Inputs" are related to the use of input arguments, although not all of them will need to be changed. The input should include a complete path to the location of the input arguments or be able to create a complete path from the input arguments.
- **More information:** https://www.geeksforgeeks.org/command-line-arguments-in-python/
- **Task Dependencies:** This task relies on the following tasks to be completed prior to beginning this task:
    - Formatting and Refactoring: Translate the notebook into an executable python script

❑ Input arguments: Allow for a flexible location to save results

- **Task:** As in the preceding task, change the main script to accept a second input argument. This second input argument should be a string that indicates the location where the output histogram figure will be saved, including the complete path to that location. Change the code that saves the histogram figure to use the updated location.
- **Task Dependencies:** This task relies on the following tasks to be completed prior to beginning this task:
- Formatting and Refactoring: Translate the notebook into an executable python script

❑ Add automated tests

- **Description:** Testing code thoroughly and frequently ensures each component of the code functions as intended.
- **Task:** Create a script that performs automated testing of your python functions, for instance verifying that the flexible input arguments you implemented in the previous task are parsed and used as intended. For more guidance on creating tests, see this guide: https://realpython.com/python-testing/
- **More information:** https://book.the-turing-way.org/reproducible-research/testing

- **Task Dependencies:** This task relies on the following tasks to be completed prior to beginning this task:
- Formatting and Refactoring: Translate the notebook into an executable python script
- Formatting and Refactoring: Refactor the script into functions

Tasks: Publishing

❑ *Essential:* Create a DOI

- **Description:** A digital object identifier (DOI) is a unique and persistent identifier that enables proper attribution and reproduction. Zenodo is a data archiving tool that is commonly used to create DOIs for digital research objects.
- **Task:** In Zenodo (https://zenodo.org/), log in or create an account via the menu in the top right corner. Then, go to "new upload" and add details about the project. Click the "reserve" button to get the DOI. Include this DOI in the project README and in any other relevant documents such as the `CITATION.cff` file (created in the below task, Publishing: Add a `CITATION.cff` file). Download the repository from GitHub as a compressed .zip file and upload the compressed repository to Zenodo. Add details of all contributors to the project in the Zenodo entry and include a link to the GitHub repository.
- **More information:** To learn more about depositing records on Zenodo, visit the records documentation page here: https://help.zenodo.org/docs/deposit/about-records/; Zenodo is also directly integrated with GitHub and allows you to mint a DOI for public repositories which you own. A tutorial for minting DOIs directly for GitHub repositories can be found here: https://docs.github.com/en/repositories/archiving-a-github-repository/referencing-and-citing-content

❑ *Essential:* Add a LICENSE file

- **Description:** A software licence describes how a piece of software can legally be used. The licence is a legal agreement between the software developer(s) and the users. By default, any creative work (such as code) is under exclusive copyright, so the authors of open-source code must explicitly grant permission for others to use their work through a licence. Depending on the needs of your project, there are many open-source licenses that may be appropriate. You can find guidelines on choosing a licence here: https://choosealicense.com/, and the Open Source Initiative

(OSI) also maintains a list of open-source and accredit licences here: https://opensource.org/licenses

- **Task:** Select a license file appropriate for the given project and add it as a plain text file named `LICENSE.txt` in the top-most (root) project directory.
- **More information:** For more information on licensing, we recommend this guide provided by the Turing Institute: https://book.the-turing-way.org/reproducible-research/licensing
- https://www.data.cam.ac.uk/data-management-guide/choosing-software-licence

❑ *Essential:* Add a copyright statement

- **Description:** A copyright statement indicates who owns the intellectual property included in the research code. It is important to establish who owns the intellectual property and therefore who can licence the software. All contributors to the project are considered copyright holders but sometimes, if the contributors are not students and the work was completed using time or resources provided by an employer, the contributor's employer may hold the copyright. This differs from institution to institution.
- **Task:** Include a copyright statement at the beginning of your licence file, stating the copyright holders (in this case, yourself and the fictional post doc).
- **More information:** The Legal Side of Open Source https://opensource.guide/legal/

❑ *Essential:* Add a CITATION.cff file

- **Description:** Adding a citation file provides clear information on how to cite your work and ensures authors receive credit for their software development work while improving dissemination and software sustainability. The citation file format (cff) provides citation metadata for software in a human- and machine-readable format.
- **Task:** Include a `CITATION.cff` file in the top-most (root) directory of the project repository, using the `example_citation.cff` file in the repository as a template.
- **More information:** https://citation-file-format.github.io/, https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-citation-files

- **Task Dependencies:** This task relies on the following tasks to be completed prior to beginning this task:
    - Publishing: Create a DOI

❑ Prepare a GitHub release

- Description: Once a software project has reached a milestone in its development, either in the development of new features or integration of new packages, a "release" of the package is created
- Task: Create an initial release of your project, following these instructions: https://docs.github.com/en/repositories/releasing-projects-on-github/managing-releases-in-a-repository
- More information: https://docs.github.com/en/repositories/releasing-projects-on-github

❑ Package the project

- **Description:** Packaging a python project enables others to easily access it using the Python Package Index (PyPI) by using the command "`pip install mypackage`" where `mypackage` is the name of the python project. Packaging your Python projects enables others to easily implement your analyses, validating your findings and extending them to other datasets.
- **Task:** Package your python project by following this tutorial: https://packaging.python.org/en/latest/tutorials/packaging-projects/

❑ Prepare the work for publication in the Journal of Open Source Software

- **Description:** The Journal of Open Source Software (JOSS) is an open access journal for research software packages. JOSS enables the quality of software to be improved through a formal peer review process while giving researchers a citable DOI from an academic journal.
- **Task:** The JOSS review criteria (https://joss.readthedocs.io/en/latest/review_criteria.html) contain several of the recommended tasks already completed in this workshop. To prepare a submission for JOSS, you must prepare a short paper and a metadata file. See the JOSS guidelines for submission (https://joss.readthedocs.io/en/latest/submitting.html) for more guidance.