Clean up and package your code and results

Exercise for applying some good (Python) practices to your code.

- **Contact:** alexander.kanitz@unibas.ch

Part 1: Package and version control your code

Exercise 1.1: Reorganize your files (3 point)

Please create an empty directory with the following files and directories:

- `README.md`: A markdown-formatted file containing instructions on how to deploy (e.g., set up a Conda environment or build a Docker image; see below) and run your code and answering all questions from previous exercises.
- `LICENSE`: A file containing an Open Source License, such as the MIT or Apache 2.0 license; you can use this service to pick a license you like.
- `environment.yml` OR `Dockerfile`: A Conda enviroment file listing all your Python and third-party requirements. Alternatively, a `Dockerfile` that can be used to build an image that contains all dependencies.
- `src/`: A directory containing all your custom code (except for tests).
- `tests/`: A directory that will contain test code (see next exercise). You can also put all your test/input files in this directory, ideally in a subdirectoy (e.g., `test_files/`).
- `run_me.sh`: A single Bash script running all code from all exercises on the test input files as per the previous two exercises. Please also include the calls to run `flake8` and `pytest` on your core repository after you have implemented these. This is really a poor man's workflow, but it's still a lot better than having to guess how you ran your code. If you like, you can organize this file so that it executes other Bash scripts (in `src/`) to keep it more tidy.
- `.gitignore`: A file with patterns indicating files/artefacts that Git should not version control, e.g., test reports. Generate it from https://gitignore.io for Python and your code editor.

Exercise 1.2: Create a Git repository and push to a remote (1 point)

- Please create a Git repository based off your new directory, then stage and commit all contents of the directory.
- Create a blank/empty project on GitHub or GitLab.
- Add the remote URL of your GitHub/Lab project to your local Git repository and push your code.

Part 2: Testing and additional documentation

Pointers for all exercises in this part are available in the Jupyter notebook `good_practices.ipynb`.

Maintainable code includes documentation and testing. For each of these subtasks, **please do not forget to update dependencies (e.g., `flake8`, `pytest`) and commit your code changes and push them to the remote repository.**

Exercise 2.1: Add type hints to your code (1 point)

Add type hints to your custom Python function/method signatures. It will be enough to only add type hints for all input arguments and the return values, although you are of course welcome to add them for any local variables as well.

Exercise 2.2: Add docstrings to your code (1 point)

Add Google-style docstrings to your custom Python functions/methods. Please do not include argument and return value types in the docstrings, as these have been already added to the signatures themselves (which is a much better idea, because the actual code is always the source of truth, and thus the risk of code and documentation diverging over time is reduced).

Exercise 2.3: Make sure your code lints (1 point)

Please use the `flake8` linter to help you refactor your code such that it adheres to Python conventions.

Exercise 2.4: Test your code (3 points)

Write unit tests for all of your custom Python functions/methods, run tests with `pytest` and make sure all tests pass. Compute the code `coverage` and make sure it's at a 100%.

Part 3: Use a workflow language/engine to run your analysis (optional)

As mentioned above, specifiying a Bash file to run your analysis is not very good. It will be difficult to parallelize your code, scatter/gather multiple jobs of the same task, keep sufficient logging and provenance information, and it will be difficult to share your analysis in a way that it is easily reproducible/reusable. Workflow languages and corresponding management systems/engines take care of all of these things and more.

If you are (planning on) doing bioinformatics analyses more regularly, we strongly recommend you to pick up one of these languages, e.g., Nextflow (Groovy-based) or Snakemake (Python-based) are two popular choices that we frequently use in our lab.

If you are interested, you can follow a tutorial on either of these domain- specific languages and learn how you can package your code as a proper shareable workflow.