Code Review

1. Repository setup

- Log into your github account and accept the invitation from the course github organization
- Click the new repository button on the top-right
- Click the “Create repository” button
- Populate your repository from local directory via command line
- To generate new personal access tokens, go to github account Settings>Developer settings>Generate new token
- See this link for more details

Listing 1. Repository setup

```
# DO NOT directly copy paste into command line due to PDF formatting issues
$ echo "#[title]" >> README.md
$ git init
$ git add README.md
$ git commit -m "first commit"
$ git branch -M main
$ git remote add origin git@github.com:[username]/[repo_name].git
$ git push -u origin main
```

2. Code review workflow

The workflow of entire code review process should follow (each step is explained in subsections)

1. Set up new branch for development purposes and commit your code
2. Create pull request upon the new branch, assign reviewers
3. Address the reviews/comments posted by reviewers
4. Request final approval of all changes and merge the branch back to main

2.1. Development branch setup

- You need to work on a new branch other than the "main" branch. `git checkout -b dev` will create a new development branch called dev and switch to that branch. Now all newer commits will be added to the dev branch
- `git add [filename]` or `git add --all` to add new files or files changed, e.g. your python script
- `git commit -m "[Commit Message]"` to commit your changes. Type **meaningful** Commit Messages, e.g. "add tsne script"
- `git pull origin dev` to pull changes from specific branch to make sure your local directory is synchronized
- `git push -u origin dev` to push all local changes to your development branch

2.2. Create pull request

- Navigate to github.com and find your repository and the development branch
- Go to the tab `Pull requests`, then click `New pull request`
- Set your "base" to main branch and "compare" to your development branch, `Create pull request`
- Remember to assign the reviewer in your pull request. For more details, refer to this post

2.3. Code review & address reviewer comments

- Reviewer will receive email notification upon pull request and review your code according to the guidelines section
- Reviewer can go to the `Files changed` tab and click + to write comments inline. After finishing all comments, click `Review changes` to submit your review.
- Once reviewer's comments are posted, you should try to resolve and fix your code accordingly. Commit new changes to your development branch

2.4. Request final approval

- When all the pending comments/request changes are addressed, you can ask the reviewer for a final approval.
- With the approval, you are now ready to merge the development branch back to main (click `Merge pull request`) so that the main branch is synchronized and reflects all the changes you have made. DO NOT discard your development branch because you need to commit new code on this branch.

3. Code review guidelines

- Correctness: it is not the reviewer's responsibility to debug the code, but you should try to spot any obvious mistakes
- Readability: refer to standard python code style . You are recommended to run a formatter, for example black, before submitting your code. For additional style recommendations (adapted form Google's Python Style Guide), refer to section of **Python style guide**.
- Performance: e.g. loop vs vectorize
- A few other tips:
  - Comments make the code easier to understand, safer from bugs and ready for change.
  - Avoid identical or very similar code in multiple places. Use function calls instead
  - Give succinct/self-descriptive names to variables/parameters, and try not to recycle variables/parameters names
  - Use consistent indentation
  - Avoid global variables

4. Python style guide

- Imports:
  - Use `import x` for importing packages and modules.
  - Use `from x import y` where x is the package prefix and y is the module name with no prefix.
  - Use `from x import y as z` if two modules named y are to be imported, if y conflicts with a top-level name defined in the current module, or if y is an inconveniently long name.
  - Use `import y as z` only when z is a standard abbreviation (e.g., np for numpy).
- Comprehensions & Generator Expressions: Each portion must fit on one line: mapping expression, for clause, filter expression. Multiple for clauses or filter expressions are not permitted. Use loops instead when things get more complicated.

- Conditional Expressions: Each portion must fit on one line: true-expression, if-expression, else-expression. Use a complete if statement when things get more complicated.
- Default Argument Values: Do not use mutable objects as default values in the function or method definition. (e.g., [])
- True/False Evaluations:
  - Always use `if foo is None:` (or `is not None`) to check for a None value. E.g., when testing whether a variable or argument that defaults to None was set to some other value. The other value might be a value that's false in a boolean context!
  - Never compare a boolean variable to False using ==. Use `if not x:` instead. If you need to distinguish False from None then chain the expressions, such as `if not x` and `x is not None:`.
  - For sequences (strings, lists, tuples), use the fact that empty sequences are false, so `if seq:` and `if not seq:` are preferable to `if len(seq):` and `if not len(seq):` respectively.
- Semicolons: Do not terminate your lines with semicolons, and do not use semicolons to put two statements on the same line.
- Line length: Maximum line length is 80 characters. Make use of Python's implicit line joining inside parentheses, brackets and braces. Line length exceptions: Long import statements, URLs, pathnames, or long flags in comments
- Parentheses: Do not use them in return statements or conditional statements unless using parentheses for implied line continuation or to indicate a tuple.
- Indentation: Indent your code blocks with 4 spaces. Never use tabs or mix tabs and spaces. In cases of implied line continuation, you should align wrapped elements either vertically, as per the examples in the line length section; or using a hanging indent of 4 spaces
- Blank Lines: Two blank lines between top-level definitions, be they function or class definitions. One blank line between method definitions and between the `class` line and the first method. No blank line following a `def` line. Use single blank lines as you judge appropriate within functions or methods.
- Whitespace: No whitespace inside parentheses, brackets or braces. No whitespace before a comma, semicolon, or colon. Do use whitespace after a comma, semicolon, or colon, except at the end of the line. No whitespace before the open paren/bracket that starts an argument list, indexing or slicing.
- Strings: Use an f-string, the `%` operator, or the `format` method for formatting strings, even when the parameters are all strings. Use your best judgment to decide between + and string formatting.
- Follow the complete Google's Python Style Guide