Activities 3: Introduction to code quality

Table of contents

Instructions and pre-requisites

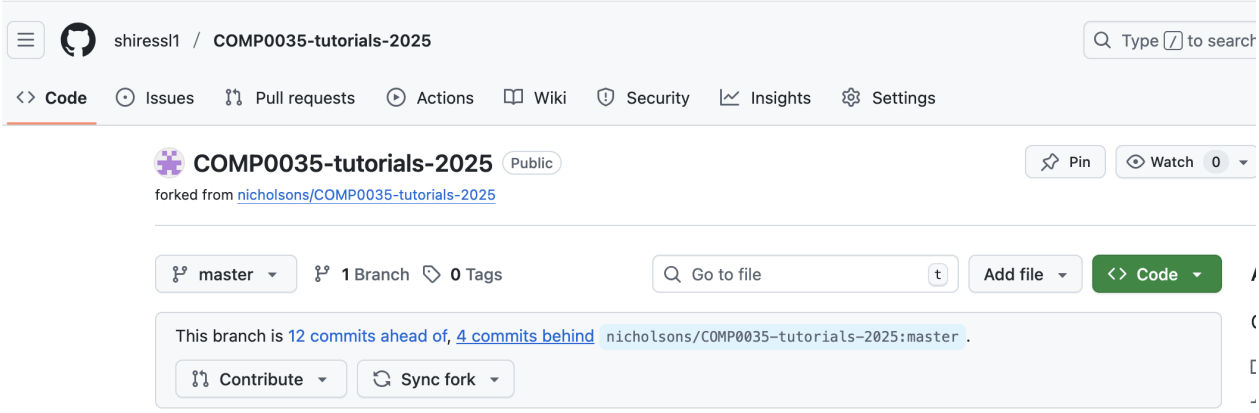
Theme: Working with code for applications

Pre-requisites

1. Update the forked tutorials repository

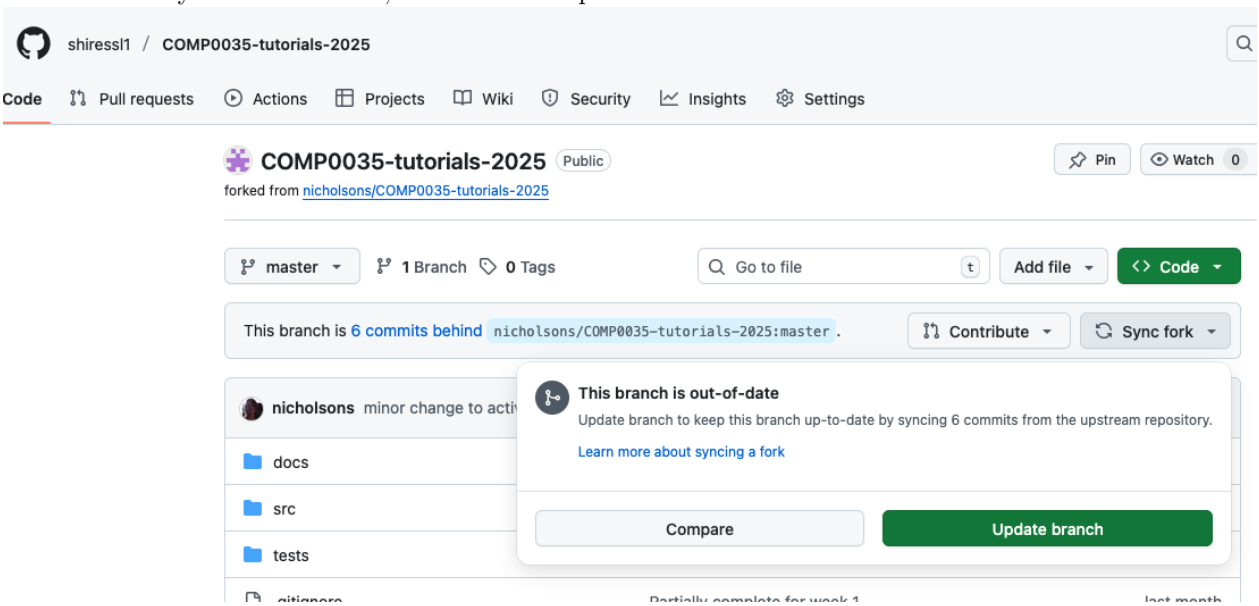
Login to GitHub and navigate to your forked copy of the COMP0035 tutorials 2025 repository (https://github.com/nicholsons/COMP0035-tutorials-2025).

Check whether any changes have been made. For example, the image below shows 4 new commits have been made to the original.



If changes have been made, you will need to update your forked repository.

Click on the "Synch fork" button; and then on "Update branch".



Now, open your IDE (VS Code, PyCharm) and update the local copy of the repository. This assumes you have integrated your IDE with your GitHub account in week 1. You may be prompted to log in to GitHub before you can carry out the following.

- In PyCharm try menu option Git > Pull
- In VS Code click on the source code control icon on the left side panel, then when the source code control pane opens, click on the three dots and select Pull.

There are other methods, look in the Help for either PyCharm or VSCode.

Activities

The activities this week focus on actions that can be taken to improve the quality of your code. That is, actions that do not affect the functionality of the code, only how it is written and structured.

The activities centre around the concept writing 'clean code'. Code is considered clean if it can be understood easily by yourself and other developers. Code that is easier to understand is easier to read, change, extend and maintain by anyone (not just the author).

Robert Martin's book Clean Code describes practices that help a programmer to write clean code, these include:

- Follow standard conventions (style guides, syntax)
- Create expressive variable and function names
- Keep it simple, avoid complexity
- Write modular code and practice DRY (Don't Repeat Yourself (DRY) in The Pragmatic Programmer by Andy Hunt and Dave Thomas)
- Document your code

The activities this week concentrate practical steps you can take in your work that help to achieve this.

Note that the quality of your code is considered throughout the coursework in this module and in COMP0034.

The activities this week are shorter. Use remaining time to complete any outstanding activities from week 2.

UPDATE: If you are using VSCode to open markdown files, please refer to their documentation (https://code.visualstudio.com/docs/languages/markdown#_markdown-preview) for how to view the rendered page rather than the raw markdown.

1. Docstring (3-01-docstrings.md)
2. Linting (3-02-linting.md)
3. Auto-formatting (3-03-formatter.md)
4. GitHub Actions lint report (3-04-github-actions.md)
5. (Optional) Static analysis: beyond linting (3-05-static-analysis.md)
6. Project structure (3-06-project-structure.md)
7. Imports (3-07-imports.md)
8. Error handling (3-08-error-handling.md)

Further reading

- How to write clean code (https://www.freecodecamp.org/news/how-to-write-clean-code/)

1. Docstrings

Python docstrings are special strings used to document your code.

A docstring appears as the first statement in a Python module, function, class, or method.

Using docstrings is considered good practice:

- Helps others, and yourself, understand what your code does without digging into the implementation.
- Supports documentation tools such as Sphinx or IDEs that can extract docstrings to generate user-friendly documentation.
- They can facilitate testing. Python's doctest module can run examples embedded in docstrings as tests.
- The can encourage good design as writing a docstring forces you to think clearly about the purpose and behavior of your code.

PEP 257 (https://peps.python.org/pep-0257/) is a Python standard that outlines conventions for writing docstrings such as:

- Use triple double quotes (“““), even for one-liners.
- Start with a short summary line.
- Optionally follow with a more detailed description.
- Include descriptions of parameters and return values.

PEP 257 does not mandate a particular style of docstring. There are several popular styles such as:

- Google-style docstring (https://google.github.io/styleguide/pyguide.html#38-comments-and-docstrings)
- Numpy-style docstring ()
- Sphinx/reStructuredText (https://sphinx-rtd-tutorial.readthedocs.io/en/latest/docstrings.html)

It does not matter for this course which style you adopt, however when you choose one then you should be consistent in its use.

Docstrings and genAI

Writing a clear docstring facilitates the use of gen-AI tools to write the corresponding code.

Conversely, gen-AI can also be used to generate docstrings from code in your IDE, e.g. `/doc Google-style docstring`.

Activity: Docstring

1. Open cs_docstring.py (../../src/activities/starter/cq_docstring.py) and consider the docstring style. Is there a style you prefer to adopt in your own code?
2. Go to the `generate_histogram()` function and follow the guidance in the comment to use copilot (or other) generate a docstring from the code.
3. Go to the `describe()` function and follow the guidance to copilot (or other) complete the code from the docstring.

Next activity (3-02-linting.md)

2. Using linters to check your code style

Linting in Python refers to the process of running a program that analyzes your code for potential errors, stylistic issues, and bugs. It's a little like having a spelling and grammar checker for your code.

A **Linter** is a Python tool that analyses Python code to detect potential errors, stylistic issues, and violations of coding standards

Python styles are documented in standards or style guides. The main style guide is PEP8 (https://peps.python.org/pep-0008/); you also saw PEP257 for docstrings (https://peps.python.org/pep-0257/) in the previous activity. Adhering to standards helps you and others to read your code.

The PyCharm IDE by default includes a linter. For the VS Code IDE you need to install a linter.

Popular Python linters include:

- ruff (https://pypi.org/project/ruff/)
- Pylint (https://pypi.org/project/pylint/)
- Flake8 (https://pypi.org/project/flake8/)

Each has its own set of rules and configurations.

Some of the things that linters can help with:

- **Code Style**: They help ensure your code adheres to a consistent style guide (e.g. PEP8), making it more readable and maintainable.
- **Best Practices**: Linters can suggest improvements based on best practices, such as avoiding certain patterns that might lead to bugs or performance issues.
- **Error Detection**: Linters can catch syntax errors, undefined variables, and other common mistakes.

Activity: Use a linter to find style issues

1. Install these linters in your virtual environment: `pip install flake8 pylint ruff`
2. Lint the file in cq_code_to_lint.py (../../src/activities/starter/cq_code_to_lint.py) with flake8: `flake8 src/activities/starter/cq_code_to_lint.py`
3. Repeat using pylint: `pylint src/activities/starter/cq_code_to_lint.py`
4. Repeat using ruff: `ruff check src/activities/starter/cq_code_to_lint.py`
5. The three linters report slightly different issues, for example:

```
(.venv) flake8 src/activities/starter/cq_code_to_lint.py
src/activities/starter/cq_code_to_lint.py:2:80: E501 line too long (81
> 79 characters)
src/activities/starter/cq_code_to_lint.py:7:1: E302 expected 2 blank
lines, found 1
src/activities/starter/cq_code_to_lint.py:17:1: E302 expected 2 blank
lines, found 1
src/activities/starter/cq_code_to_lint.py:18:80: E501 line too long (82
> 79 characters)
src/activities/starter/cq_code_to_lint.py:21:1: F811 redefinition of
unused 'incorrect_spacing_between_functions' from line 17
src/activities/starter/cq_code_to_lint.py:26:13: E222 multiple spaces
after operator
src/activities/starter/cq_code_to_lint.py:28:18: W292 no newline at end
of file

(.venv) pylint src/activities/starter/cq_code_to_lint.py
************* Module src.activities.starter.cq_code_to_lint
src/activities/starter/cq_code_to_lint.py:28:0: C0304: Final newline
missing (missing-final-newline)
src/activities/starter/cq_code_to_lint.py:5:0: C0103: Constant name
"globalTEST" doesn't conform to UPPER_CASE naming style (invalid-name)
src/activities/starter/cq_code_to_lint.py:7:0: C0116: Missing function
or method docstring (missing-function-docstring)
src/activities/starter/cq_code_to_lint.py:7:0: C0103: Function name
"inCorrect_functionName" doesn't conform to snake_case naming style
(invalid-name)
src/activities/starter/cq_code_to_lint.py:11:0: C0116: Missing function
or method docstring (missing-function-docstring)
src/activities/starter/cq_code_to_lint.py:17:0: C0116: Missing function
or method docstring (missing-function-docstring)
src/activities/starter/cq_code_to_lint.py:21:0: C0116: Missing function
or method docstring (missing-function-docstring)
src/activities/starter/cq_code_to_lint.py:21:0: E0102: function already
defined line 17 (function-redefined)
src/activities/starter/cq_code_to_lint.py:25:0: C0116: Missing function
or method docstring (missing-function-docstring)

-----------------------------------
Your code has been rated at 1.88/10

(.venv) ruff check src/activities/starter/cq_code_to_lint.py
F811 Redefinition of unused `incorrect_spacing_between_functions` from
line 17
  --> src/activities/starter/cq_code_to_lint.py:21:5
   |
21 | def incorrect_spacing_between_functions():
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
`incorrect_spacing_between_functions` redefined here
22 |     print("This is a duplicated function name")
   |
  ::: src/activities/starter/cq_code_to_lint.py:17:5
   |
15 |     return result
```

```
16 |
17 | def incorrect_spacing_between_functions():
   |     ---------------------------------- previous definition of
`incorrect_spacing_between_functions` here
18 |     print('This function has incorrect spacing between it and the
function above')
   |
help: Remove definition: `incorrect_spacing_between_functions`

Found 1 error.
```

Configuring the linter

You will need to refer to the documentation for the linter you are using for methods to configure the linter, for example the ruff documentation (https://docs.astral.sh/ruff/configuration/). You can typically configure the linter when you run it from the command line, or in a configuration file such as `pyproject.toml`.

For this course, consider adding linter configuration to `pyproject.toml`.

For example, if you are following the Google style guide, it suggests line length of 80 characters rather than the PEP8 specification of 79 characters so you might set the linter to flag warnings only if your line length is greater than 80. Some developers prefer a longer line length such as 88 or 100 characters.

Examples of how you can set this config in `pyproject.toml`:

```
# Pylint configuration
[tool.pylint]
max-line-length = 80 # Default is 79 (PEP8 standard length)
good-names = ["i", "j", "df", "x", "y"]  # Prevents variable name
        warnings for these common names that don't meet PEP8

# Flake8 configuration
[tool.flake8]
max-line-length = 100  # Default is 79 (PEP8 standard length)

# Ruff configuration
[tool.ruff]
line-length = 100 # Default is 88
```

As well as configuring what the reporter will lint or ignore, you could also install additional packages to format the output, for example flake8-html (https://pypi.org/project/flake8-html/) to output to html format report.

Next activity (3-03-formatter.md)

3. Fixing issues with a formatter

The purpose of using is a linter is to identify the issues. The benefit however is only achieved once you fix the identified issues. Running a linter is the easy part, the challenge is to then fix the identified issues.

Formatters are packages that will autoformat your code, e.g.

- ruff (https://pypi.org/project/ruff/) - ruff can be used to detect and correct
- Black (https://black.readthedocs.io/en/stable/)
- autopep8 (https://pypi.org/project/autopep8/) that you may wish to explore.

The tools use rules and correct your code to meet these. You may not always agree with the rules

PyCharm has options that will correct your code. If you are using PyCharm, try opening the code_to_lint.py and go to the menu option 'Code | Format Code' and it should correct the spacing, line break and any indentation issues.

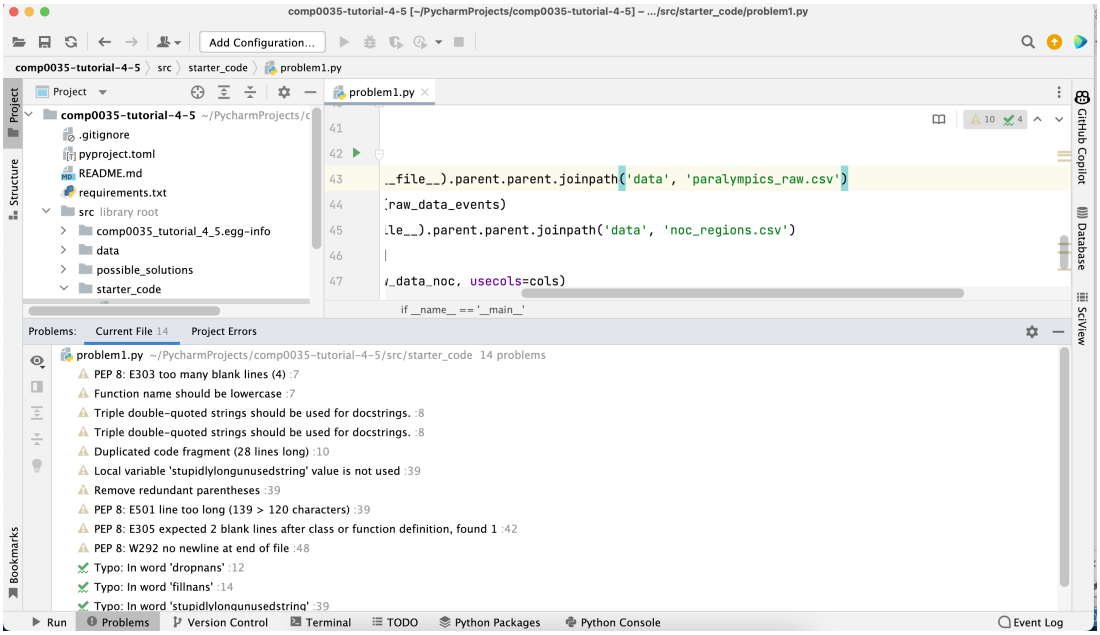
Both PyCharm and VS Code, particularly when you combine them with Copilot, will warn of style issues and offer solutions.

Typically, where you see an error (by default a red squiggly underline in VS Code, or highlighted code in PyCharm, or a lightbulb symbol); then the IDE will help you to fix it. Usually, clicking on the lightbulb will tell you what is wrong and if possible, offer to fix it for you.

You will need to refer to the documentation for your IDE to work out how to identify warnings and use linting:

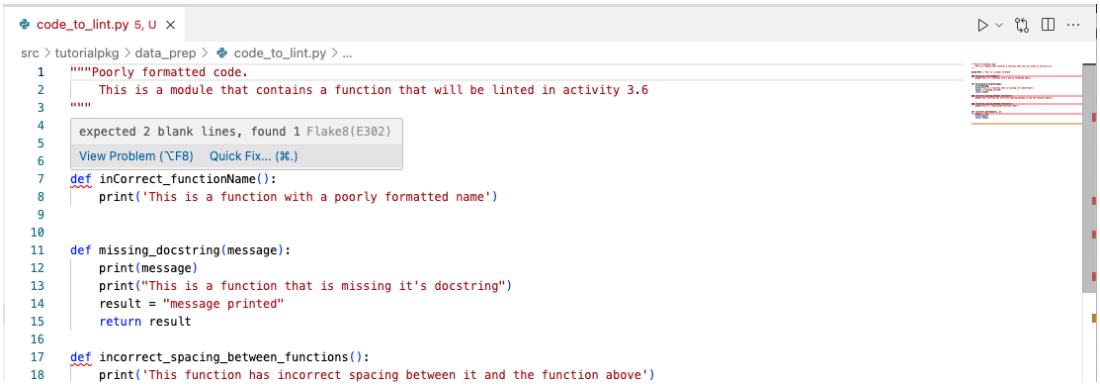
- VS Code linting (https://code.visualstudio.com/docs/python/linting)
- VS Code identify warnings (https://code.visualstudio.com/Docs/editor/editingevolved#_errors-warnings)
- PyCharm Python code inspections (https://www.jetbrains.com/help/pycharm/running-inspections.html)
- PyCharm fix problems (https://www.jetbrains.com/help/pycharm/resolving-problems.html)

Example of PyCharm style warnings, click on the warning triangle in the upper right of the code pane:



0-1. PyCharm lint example

Example of VS Code style warning, hover over the squiggle to see the options:



0-1. VS Code lint example

NB: In VS Code, the warnings will not disappear as soon as you correct the code, you need to save the changes. The linter appears to run only on save.

Activity: Use a formatter to correct issues

Run commands in the terminal virtual environment.

1. Make a copy of cq_code_to_reformat.py (../../src/activities/starter/cq_code_to_reformat.py) - this is just so you can repeat the activity later if you want. Open the file to see the issues but do not correct anything.
2. Run a linter: `flake8 src/activities/starter/cq_code_to_reformat.py`
3. Install autopep8: `pip install autopep8`
4. Auto-format the file: `autopep8 --in-place --aggressive --aggressive src/activities/starter/cq_code_to_reformat2.py`
5. Run a linter again: `flake8 src/activities/starter/cq_code_to_reformat.py`. Fewer issues should be reported.
6. Open the file again and see the changes that have been made.

Next activity (3-04-github-actions.md)

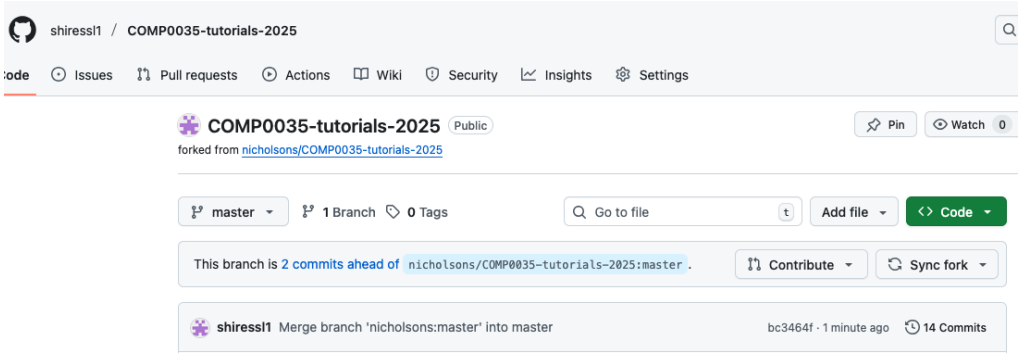
4. GitHub Actions workflow to automate linting

GitHub Actions (https://docs.github.com/en/actions) is a GitHub feature that allows you run automated workflows on your repository code using a virtual server within the GitHub ecosystem. For example, a workflow that runs a linter or static analyser and reports the results every time you push your code changes to GitHub.

GitHub provides templates for different starter templates, there is often one that is close to what you want.

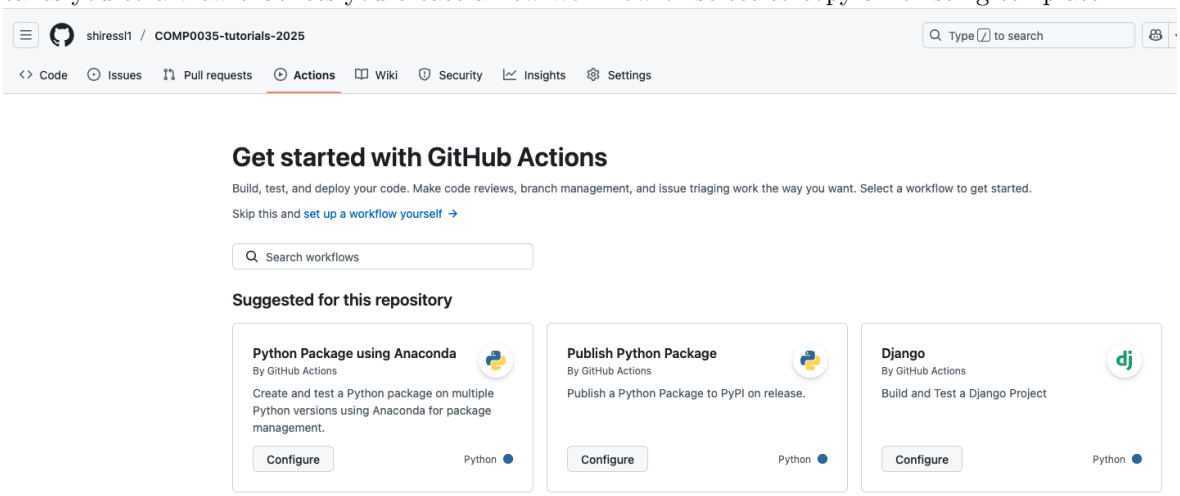
Create a workflow for the tutorial project

1. Navigate to your forked copy of the repository on GitHub. Mine looks like this:

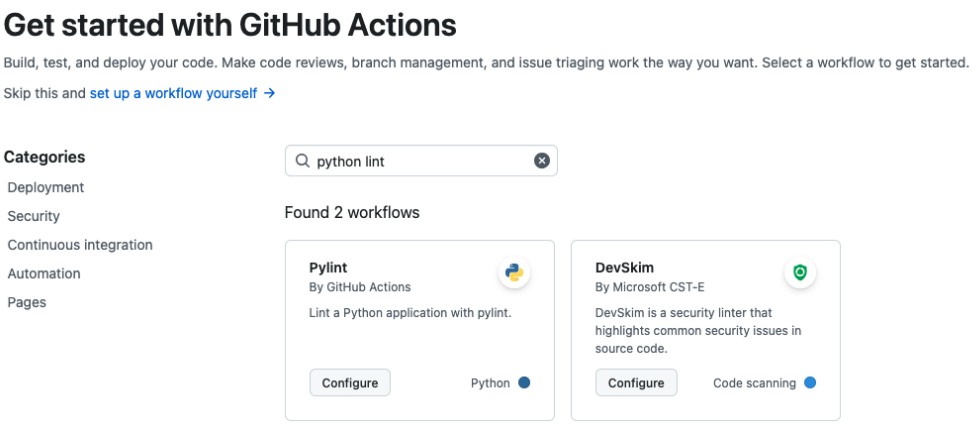


0-1. Tutor's forked copy of the tutorial project

2. Find the Actions menu link and click on it. You can see it in the image above near the top. This takes you to a view that lets you create a new workflow or select to copy an existing template.



3. Add `python lint` to the search bar and enter. Choose the 'Pylint' by clicking on it's `Configure` button.



0-1. gha-configure-workflow.png

4. This will generate a copy of the template which is in a format called YAML. You probably need to edit this. Below I have added comments to suggest what you should change:

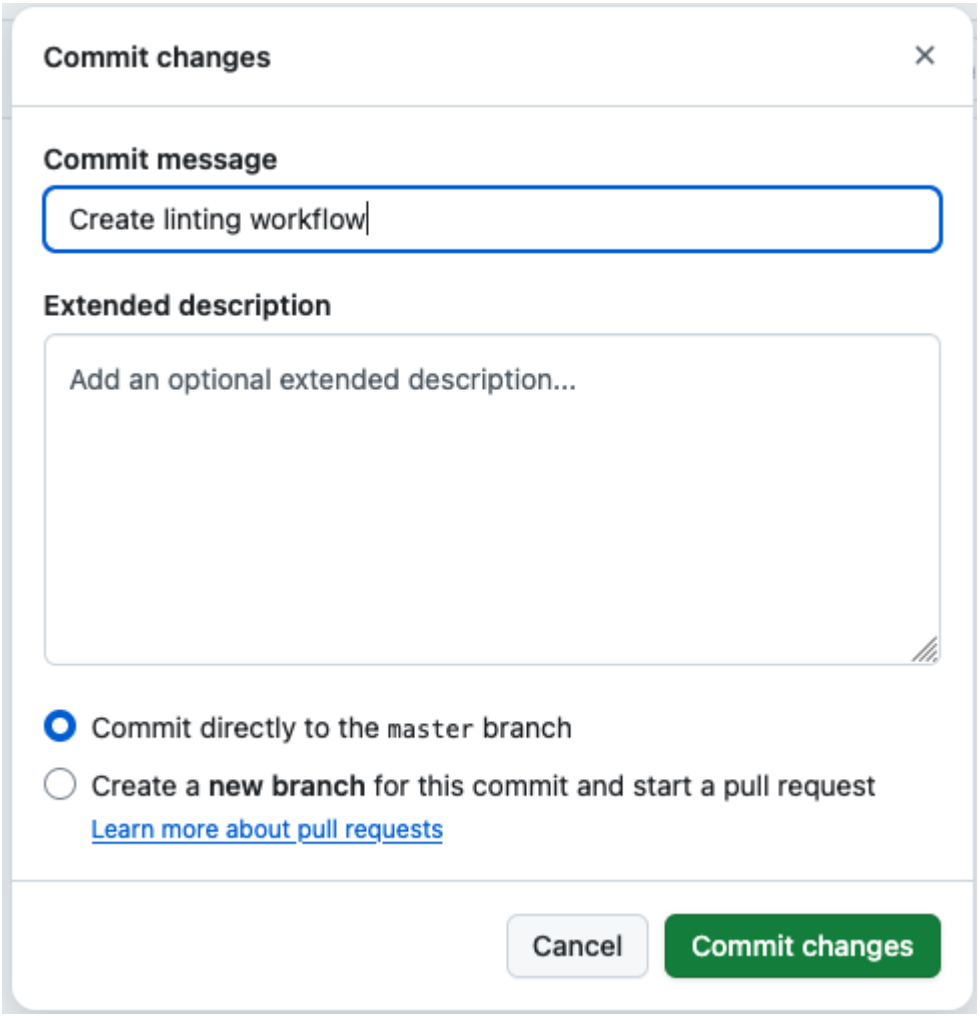
```
name: Pylint   # Optional change. Name of the workflow that will
        appear in your workflow list.

on: [push]   # Do not change. Tells GitHub to run this workflow
        when changes are pushed to repository.

jobs:
  build:
    runs-on: ubuntu-latest   # Optional change. Specifies the
        server to run the code on. You can change to windows or
        mac if you wanted. See https://github.com/actions/runner-
        images for the options.
    strategy:
      matrix:
        python-version: ["3.8", "3.9", "3.10"]  # Suggested
        change. List of python versions to run the workflow on.
        Set here to run 3 times on 3.8, 3.9, 3.12. Suggest you set
        this to match your venv, likely 3.12 or 3.13
    steps:  # Do not change. List of steps that the workflow
        carries out. Checks out your code, sets up the environment
        with the specified python version(s)
    - uses: actions/checkout@v4
    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v3
      with:
        python-version: ${{ matrix.python-version }}
    - name: Install dependencies  # Required change. Install
        dependencies. Add the last line "pip install -e ." as this
        is not in the template.
      run: |
        python -m pip install --upgrade pip
        pip install pylint
        pip install -e .
    - name: Analysing the code with pylint  # Optional change.
        Runs the pylint and checks all python files. You can
        change it to lint specific folders/files. You can also add
        steps to run other tools, try adding flake8 or ruff.
      run: |
        pylint $(git ls-files '*.py')
```

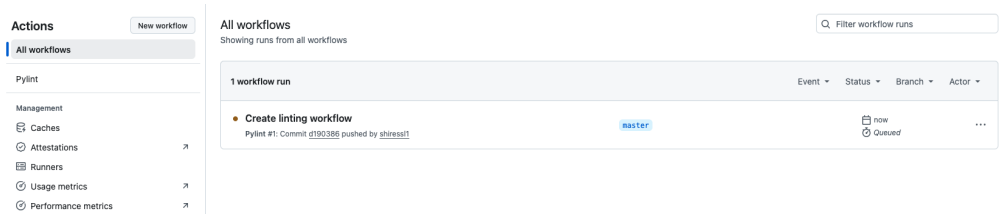
Once you have made the changes you need, select the green `Commit changes...` button in the top right.

5. Select ‘Commit changes’ again on the next window.



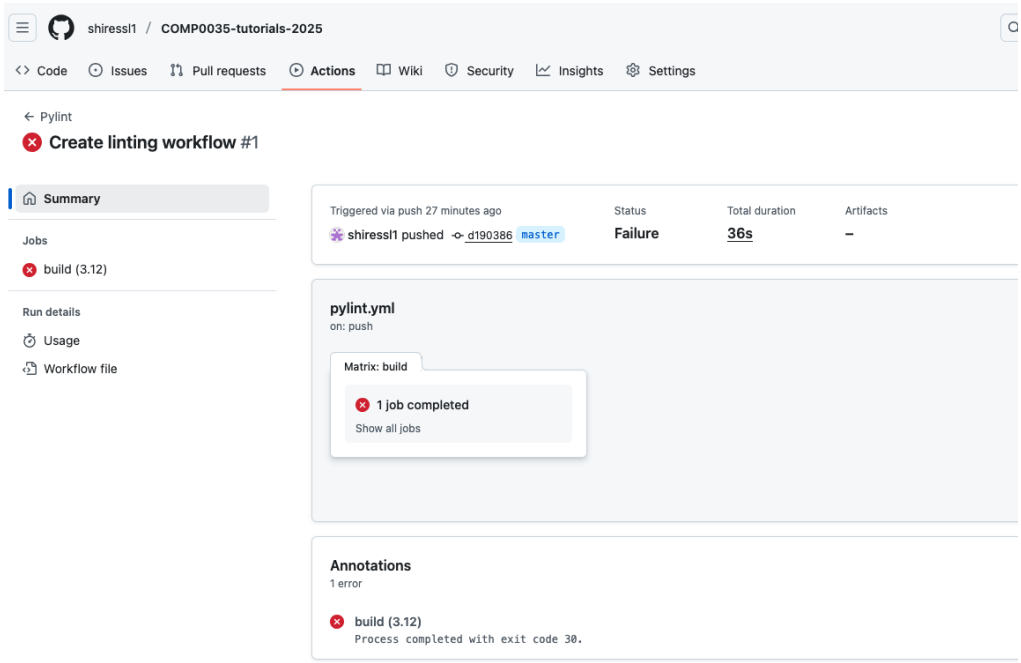
0-1. Select ‘Commit changes’ again

6. The workflow `.yml` is now added to a `.github` folder within your project files. Adding this file and commiting it in the last step caused a ‘push’ on the repo so it immediately runs the workflow. Select Actions from the menu at the top. The workflow is running, the status can be amber (in the screenshot) which means it is running, green it ran successfully, or red it ran but issues were found.



0-1. Workflow running

7. Click on the highlighted name of the workflow (in the image above it is “Create linting workflow”). This opens that run of the workflow and shows the status of each job, the following only has one job as I only configured the workflow to run on Python 3.12 and not multiple versions. Click on the job name, in the below image it would be ‘1 job created’.



0-1. Workflow job status

8. You can now see a report. The section headings map to the step descriptions that were in the .yml file. If you expand the red X button you can see what the issue is - it failed the linting (this is due the deliberate issues in one of the activity starter code files).

This activity gave you a basic linting workflow. There are many options you can configure and more steps that can be added. Refer to the GitHub Actions documentation (https://docs.github.com/en/actions/how-tos) for more information.

Next activity (3-05-static-analysis.md)

5. (Optional) Static analysis: beyond linting

This activity is optional, you can skip to the next activity (3-06-project-structure.md).

Linters belong to a category of tools referred to as 'static analysers', that is tools that check or analyse code but don't change it.

As well as linters, there are tools that check for:

- cyclomatic complexity mccabe (https://github.com/PyCQA/mccabe)
- security vulnerabilities bandit (https://bandit.readthedocs.io/en/latest/)
- duplicated code PMD CPD (https://pmd.github.io/pmd/pmd_userdocs_cpd)

Some tools combine several tools in one such as prospector (https://pypi.org/project/prospector/)

Cyclomatic complexity

Cyclomatic complexity is a metric used to measure the complexity of a program by quantifying the number of independent paths through its source code. It was introduced by Thomas McCabe in 1976 and is used in assessing how difficult a function or module might be to test or maintain.

The code in `paralympics_add_data.py` is more complex than other code in the examples so try the following:

1. `pip install mccabe`
2. `python -m mccabe src/activities/starter/paralympics_add_data.py`

While there are no specific rules as to what is an acceptable or desired level of complexity. 10 is often cited as a score that indicates excessively complex code that could be more efficiently broken down into smaller, more manageable parts.

NB you can also configure flake8 and ruff to report on complexity.

Combined tools

Try using prospector to see the combined output of several static analysis tools:

1. `pip install prospector`
2. `prospector src/activities/starter/paralympics_add_data.py`

Results in:

```
src/activities/starter/paralympics_add_data.py
  Line: 3
    pylint: line-too-long / Line too long (103/100)
  Line: 72
    pylint: f-string-without-interpolation / Using an f-string that
does not have any interpolated variables (col 19)
  Line: 80
    pylint: f-string-without-interpolation / Using an f-string that
does not have any interpolated variables (col 27)
  Line: 94
    pylint: line-too-long / Line too long (101/100)
  Line: 102
    pylint: line-too-long / Line too long (109/100)
    pylint: f-string-without-interpolation / Using an f-string that
does not have any interpolated variables (col 31)
  Line: 104
    pylint: line-too-long / Line too long (102/100)
  Line: 115
    pylint: line-too-long / Line too long (102/100)
  Line: 128
    pylint: line-too-long / Line too long (110/100)
  Line: 131
    pylint: line-too-long / Line too long (101/100)
  Line: 137
    pylint: line-too-long / Line too long (101/100)
  Line: 153
    pylint: line-too-long / Line too long (102/100)
  Line: 225
    pylint: line-too-long / Line too long (111/100)



Check Information
=================
         Started: 2025-09-20 19:54:22.789916
        Finished: 2025-09-20 19:54:24.494414
      Time Taken: 1.70 seconds
       Formatter: grouped
        Profiles: default, no_doc_warnings, no_test_warnings,
strictness_medium, strictness_high, strictness_veryhigh,
no_member_warnings
      Strictness: Noneprospector --no-django src/activities/starter/
paralympics_add_data.py
       Tools Run: dodgy, mccabe, profile-validator, pycodestyle,
pyflakes, pylint
  Messages Found: 14
 External Config: pylint: /PycharmProjects/COMP0035-tutorials-2025/
pyproject.toml
```

Next activity (3-06-project-structure.md)

6. Project structure

Recap from week 2

Activity 2-01-python-structure.md (../2_pandas/2-01-python-structure.md) introduced the concept of organising your project using packages, modules, functions and/or classes.

- A Python module (https://docs.python.org/3/tutorial/modules.html) is "a file containing Python definitions and statements. The file name is the module name with the suffix .py appended."
- A Python package (https://docs.python.org/3/tutorial/modules.html#packages) is "a way of structuring Python's module namespace by using 'dotted module names'. For example, the module name A.B designates a submodule named B in a package named A." There is a more advanced concept, 'namespace packages', in Python that is not covered in this minor.
- A Python function (https://docs.python.org/3/glossary.html#term-function) is "A series of statements which returns some value to a caller. It can also be passed zero or more arguments which may be used in the execution of the body."
- Directories (or folders) and files that are not Python code files, e.g. data files such as `.csv`, `.xlsx`; or database files such as `.sqlite` or `.db` files.
- Project configuration and information files e.g. README.md, .gitignore, pyproject.toml etc.

Typical Python application project structure

There is no single pattern for structuring the files associated with a Python application. However, there are common patterns. Adhering to these is useful, as often Python tools will auto discover or recognise files and folders saved in these patterns; and it will also help other developers to more easily understand your code.

There are examples of typical Python project structures on these sites:

- Python Packaging User Guide (https://packaging.python.org/en/latest/discussions/src-layout-vs-flat-layout/#src-layout-vs-flat-layout)
- The Hitchhiker's Guide to Python (https://docs.python-guide.org/writing/structure/#sample-repository)
- Real Python (https://realpython.com/python-application-layouts/)

The tutorial project is not a good example of how to structure an application project, it is not designed for an app, it is designed as a weekly series of activities and so there are repeated module names, repeated functions, etc. You will likely see lots of warnings in your IDE about duplicated code.

Many of the package and module names also break the Python convention in PEP8. This is usually as I made the names longer so it is clearer what the code is. PEP8 (https://peps.python.org/pep-0008/#package-and-module-names) says:

- Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability.
- Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

A more suitable structure for your coursework might be as follows. Use meaningful package and module names, not "my_python_project", "module1" or "app". You may also have sub-packages within your project package. You won't have tests until coursework 2.:

```
my_python_project/
    ├── README.md            # Information about your project for you
and other developers who use your code
    ├── requirements.txt     # Lists packages your code requires. Used
when creating a virtual environment.
    ├── pyproject.toml       # Configuration values for your project
and tools that might be used (e.g. test, lint, formatter)
    ├── .gitignore           # Lists files that git should ignore and
not add to github e.g., venv, IDE config, temporary files
    ├── .venv/               # Code for the virtual environment, '.'
hides the folder depending on your operating system settings
    ├── my_python_project/   # The main package for your project,
usually the app or project name
    │   ├── __init__.py
    │   ├── app.py           # The entry point to run the app code
    │   ├── module1.py
    │   └── module2.py
    ├── data/               # Data files and database.
    │   ├── __init__.py
    │   ├── processed_data.csv
    │   ├── my_database.db
    │   └── raw_data.csv
    ├── tests/               # Contains test code, "tests" is a name
recognised by most test runners.
    │   ├── __init__.py
    │   ├── test_app.py      # Test modules usually start with 'test_'
or end '_test'
    │   ├── test_module1.py
    │   └── test_module2.py
```

Project and tool config files

These files were explained in week 1.

- requirements.txt in 1-02-environments.md (../1_structure/1-02-environments.md#requirementstxt)
- pyproject.toml in 1-02-environments.md (../1_structure/1-02-environments.md#pyprojecttoml)
- README.md in 1-03-source-code-control.md (../1_structure/1-03-source-code-control.md#readmemd)
- .gitignore in 1-03-source-code-control.md (../1_structure/1-03-source-code-control.md#gitignore)

Activity: review project structure for the coursework

Review the structure for your coursework repository. Does your project structure look consistent with a typical Python application project structure?

Using the src layout, or a flat layout, is encouraged as tools often auto recognise this by default, so avoiding the need for additional configuration such as in `pyproject.toml` (see next activity).

There are examples of Python project structures on these sites:

- Python Packaging User Guide (https://packaging.python.org/en/latest/discussions/src-layout-vs-flat-layout/#src-layout-vs-flat-layout)

- The Hitchhiker's Guide to Python (https://docs.python-guide.org/writing/structure/#sample-repository)
- Real Python (https://realpython.com/python-application-layouts/)

Next activity (3-07-imports.md)

7. Importing Python packages/libraries

You are already aware that to use Python packages in your code you import them. This activity covers guidance on how to structure the imports, including how to import your own application packages and modules.

Structuring imports

PEP8 (https://peps.python.org/pep-0008/#imports) says that:

- Imports should be at the top of the Python file
- Group imports in the following order with a blank space between each group
    - Standard library imports
    - Related third-party imports
    - Local application/library specific imports
- Order imports alphabetically within the group
- Avoid wildcard imports like `from module import *`
- Prefer absolute imports over relative imports

For example:

```
# Standard library imports
import os
from math import pi

# Related third-party library imports
import pandas as pd
from plotly.graph_objs import Bar, Layout

# Local application specific imports
import my_module
from my_module import my_function
```

In PyCharm, the menu option Code > Reformat Code will automatically sort the imports in this order. Some formatters may also.

Importing your own code

To import your own packages and modules within your code depends on Python being able to locate them.

PEP8 states **prefer absolute imports to relative imports**.

Relative imports are a way to import modules based on their location relative to the current module (i.e. the file doing the importing), rather than from the root of the project or the Python path, e.g., `from . import module_a`

Absolute imports to importing modules using their full path from the project's root directory. This is the most common and recommended way to import modules, especially in larger projects. For example: `from my_package import my_module` or `from my_package.my_module import my_function_one, my_function_two`.

Importing modules using their full path does not mean you have to specify the full directory path in the import. By installing your own code in the Python environment allows it to determine the full path from the registered package names. This is why it is important to install your own code as a package in the Python environment you are using for the project.

`pip install -e .` with `pyproject.toml`

Executing `pip install -e .` in the Python venv installs your project and any dependencies that are specified in `pyproject.toml`.

`-e` stands for "editable". It tells pip to install your project in a way that any changes you make to the source code are immediately reflected when you run or import the package so there is no need to reinstall after every change.

`.` tells the pip to treat the current directory you are in as the root project directory when you run the command. Make sure the `pyproject.toml` is in this root directory (i.e. the top level of the project). In most cases you should be in the project root directory when you run `pip install -e .`.

What you specify in `pyproject.toml` is important. However, there is no single solution to this as it depends on your project structure and the tool you are using to build the package. The examples in this course all assume you are using `setuptools` to build the package; however if you are using poetry you will need to read the relevant poetry documentation instead. There is guidance for writing pyproject.toml is here (https://packaging.python.org/en/latest/guides/writing-pyproject-toml/) - make sure you select setuptools in the code examples it gives you.

If you are using a default Python project structure with your app code in a `src` directory (note `src` is a plain directory and not a Python package, there is no `src/__init__.py`), then setuptools and pip should auto discover your packages. If you deviate from this, e.g. using a folder other than src that has code in, or you have multiple packages in the root of the project folder, then you need to specify the package location for setuptools to find. Refer to the setuptools documentation (https://setuptools.pypa.io/en/latest/userguide/pyproject_config.html) for what to put in the `[tool.setuptools.packages.find]` section of `pyproject.toml`.

If the data files are in a sub-package of your main application package then you should be able to import them without further configuration in `pyproject.toml`. If you place them elsewhere then refer to the setuptools documentation for data files (https://setuptools.pypa.io/en/latest/userguide/datafiles.html).

Checking that your own package code is installed

Execute the command `pip list` in the venv. It should list your project package name as one of the installed packages, e.g.:

```
(.venv) pip list
Package    Version Editable project location
---------- ------- ----------------------------------------
aproject    1       /Users/someone/aproject
pip        24.3.1
setuptools 80.9.0
```

If not, then execute the command `pip install -e .`. This should return a message such as "Successfully installed myproject-2025.0.1" and you may see a folder created with the extension `.egg-info`, e.g. `myproject.egg-info`.

If it returns something else then check for the error messages in the output in the terminal.

For example:

```
Getting requirements to build editable did not run
successfully.
```

This indicates that you have more than one package at the root of the project so it cannot determine which is the main package code. A solution is to move these into a 'src' directory or if you want to keep the flat-layout then move them to be sub-packages of a single top level package.

```
  × Getting requirements to build editable did not run successfully.
  │ exit code: 1
  ╰─> [14 lines of output]
      error: Multiple top-level packages discovered in a flat-layout:
['onepackage', 'anotherpackage'].
```

```
ERROR: file:///Users/sarahsanders/PycharmProjects/snake does
not appear to be a Python project: neither 'setup.py' nor
'pyproject.toml' found.
```

This is often because you named `pyproject.toml` incorrectly; `pyproject.toml` isn't in the root folder; or you are running the command while not in the project root folder.

Data and database file locations

How to reference data files was already covered in 2-01-python-structure.md (../2_pandas/2-01-python-structure.md#file-locations)

Please recap if you don't remember the guidance. Incorrectly referencing files frequently causes issues in coursework submissions.

Check the pyproject.toml for your project

- Read the guidance (https://packaging.python.org/en/latest/guides/writing-pyproject-toml/#static-vs-dynamic-metadata) for the `[project]` section and include `name`, `version` and `dependencies`
- Read the guidance (https://packaging.python.org/en/latest/guides/writing-pyproject-toml/#declaring-the-build-backend) for the `[build-system]` section using `setuptools`
- If not using `src/my_package` style layout, read the guidance for setuptools package discovery (https://setuptools.pypa.io/en/latest/userguide/pyproject_config.html).

Next activity (3-08-error-handling.md)

8. Error handling

NB This activity mentions database connections. Database connections are covered in week 4. This should not prevent you completing this activity as any database specific code is given to you within this activity and the associated starter code files.

Introduction

Errors and exceptions can lead to unexpected behaviour, or even cause an app or script to stop running.

Errors is an issue that occurs that prevents the code from completing. For example, a syntax error such as `print(0 / 0))` (unmatched brackets).

Exceptions are slightly different. The code may be syntactically correct yet when the code is run an error occurs. Exceptions can be caught and handled using try/except.

Linting and as static analysis tools will typically find errors; however potential exceptions are harder to identify.

Instead, it is better to consider how your code might fail when you are writing it and identify potential exceptions.

You should then write code that allows your code to handle the exception gracefully.

try/except/else/finally

Use the try and except block to catch and handle exceptions. You can optionally add else and/or finally.

- **try** Python executes code following the `try` statement as a normal part of the program execution.
- **except** If there is an exception is raised, execute the code within the `except` statement. You can have multiple except statement to handle different types of exception.
- **else** If there is no exception, execute the code after the `else` statement after the code in the `try` statement
- **finally** This is optional but if included the code after the `finally` statement is always run, even if there is an exception. Often used for cleaning up resources, for example closing a file.

There is an example of this in the `print_data` function in starter_exceptions.py (../../src/activities/starter/starter_exceptions.py)

Alternative methods for catching multiple exceptions

More detail on catching multiple exceptions in this tutorial: Real Python: How to Catch Multiple Exceptions in Python (https://realpython.com/python-catch-multiple-exceptions/)

Structural pattern matching

Structural pattern matching is a feature introduced in Python 3.10 that allows you to match complex conditions using a match statement, similar to switch statements in other languages. This is considered more readable than if/elif chains.

The technique is not specifically for exception handling. However, it could be used to write the code to handle multiple potential errors more clearly.

There is an example of this in the `print_data_pattern_example` function in starter_exceptions.py (../../src/activities/starter/starter_exceptions.py)

Exception groups and add_note

Introduced from Python 3.11. Bundles multiple unrelated exceptions, useful when multiple tasks fail at once, and you want to raise them together e.g. useful in asynch functions in web apps

Exception groups take two arguments:

- The usual description
- A sequence of sub-exceptions

Which exception to use?

Python and most Python libraries have built in exceptions with descriptions that can be raised when a given type of issue arises.

As far as you can try to identify which exception is likely to be raised, and be as specific as possible

For example, if you try to open a file that may not be found in the location specified, don't raise the base Python `Exception` instead raise an OS `FileNotFoundError` exception.

Most guidance, and therefore linters (e.g., ruff (https://docs.astral.sh/ruff/rules/bare-except/)), warn that you should not raise a bare exception. A bare exception is when you don't specify the type of exception to catch, e.g.:

```
try:
    # some code
except:
    # handle any exception
```

A bare except will catch all exceptions, including system-exiting exceptions like KeyboardInterrupt and SystemExit. This can make it hard to stop your program with Ctrl+C, or can hide errors and bugs. It is considered bad practice because it makes your code less predictable and harder to maintain. It's considered better to catch only the exceptions you expect and know how to handle.

You can also define your own exceptions. If you do this note that PEP8 says exceptions should be named with the suffix `Error` e.g. `ConnectionError`

Which exceptions to handle?

Deciding when you need to handle an exception is tricky. You will learn more as you experiment with running code, reading documentation and tutorials etc.

As a starting point for this coursework, errors typically arise when the file can't be found, a database can't be opened, or a data attribute or value is not found.

Consider the following situations that might lead to an error:

- File and I/O errors for files that don't exist, insufficient permissions
- Database errors: connection errors, data integrity issues, programming syntax error, invalid data types
- Data parsing and format when reading csv, xlsx, JSON data
- Type and value errors for incorrect data types and invalid values
- Index or key errors when index is out of range or a key is not found
- Math computation errors such as division by zero

Some of the more common exceptions to use:

Python file handling:

- FileNotFound (https://docs.python.org/3/library/exceptions.html#FileNotFoundError)

Pandas dataframes:

- AttributeError (https://docs.python.org/3/library/exceptions.html#AttributeError)
- KeyError (https://docs.python.org/3/library/exceptions.html#KeyError)
- DtypeWarning (https://pandas.pydata.org/docs/reference/api/pandas.errors.DtypeWarning.html)

sqlite3 invalid data:

- IntegrityError (https://docs.python.org/3/library/sqlite3.html#sqlite3.IntegrityError)

Also consider using a context manager to handle sqlite3 database connections (https://docs.python.org/3/library/sqlite3.html#sqlite3-connection-context-manager). This automatically commits or rolls back transactions and is useful to combine with exceptions. This example is from the sqlite3 documentation:

```
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lang(id INTEGER PRIMARY KEY, name VARCHAR
        UNIQUE)")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))

# con.rollback() is called after the with block finishes with an
        exception,
# the exception is still raised and must be caught
try:
    with con:
        con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))
except sqlite3.IntegrityError:
    print("couldn't add Python twice")

# Connection object used as context manager only commits or rollbacks
        transactions,
# so the connection object should be closed manually
con.close()
```

Activity: add exception handling

1. Consider the `create_db` and `describe` functions in starter_exceptions.py (../../src/activities/starter/starter_exceptions.py).
2. Identify where errors might be likely to occur.
3. Add exception handling.
4. Test out your exception handling by passing incorrect values to the functions.

AI spoiler: If you add a `Raises:` section to the docstring and specify the exceptions and conditions when they should be raised, copilot can more easily add the relevant try/catch for you.