Activities 9: Testing with pytest

Table of contents

Activities 9: Testing (unit testing and continuous integration in GitHub)

Theme: Working with code (for applications)

This is the final set of activities for COMP0035.

Pre-requisites

Before starting, check you:

1. Configured your IDE to support pytest:
    - Pycharm help: Testing frameworks (https://www.jetbrains.com/help/pycharm/testing-frameworks.html)
    - Python testing in VS Code (https://code.visualstudio.com/docs/python/testing)
2. Installed `pytest` and `pytest-cov` in your venv `pip install pytest pytest-cov`
3. Pytest good practices (https://docs.pytest.org/en/stable/explanation/goodpractices.html) recommends you install your project code in your virtual environment (venv) using

    ```
    pip install -e .
    ```

    Check that you updated your pyproject.toml if you changed your code directory structure or name.

Complete the activities

Instructions and activities can be found in the docs/9_testing folder:

1. Introduction to testing and conventions (9-01-introduction.md)
2. Testing with pytest (9-02-pytest-tests.md)
3. Pytest fixtures (9-03-fixtures.md)
4. Reporting test coverage (9-04-coverage.md)
5. Running tests with GitHub Actions (9-05-ci-github.md)
6. Further information (9-06-further.md)

Next activity (9-01-introduction.md)

1. Introduction to unit testing

Please read the following guidance before you start to write unit test code.

Unit testing in Python

Unit testing is a method of software testing that focuses on the smallest testable parts of a program, known as **units**, typically individual functions or methods.

The goal of unit testing is to ensure that each part of the code works correctly and as intended, including helper functions that may not be directly visible to the user.

Other types of testing, such as integration testing, were introduced in the lecture but are not covered in this tutorial or the coursework. Integration testing will be explored further in COMP0034.

In Python, the built-in unittest (https://docs.python.org/3/library/unittest.html) module provides basic tools for testing.

This tutorial focuses on pytest (https://docs.pytest.org/en/stable/getting-started.html), a framework that builds on unittest. Unlike `unittest`, `pytest` must be installed separately e.g. using `pip install pytest`.

We are using `pytest` here because you will also need it next term in COMP0034.

`pytest` is a testing framework. It provides tools to write and run tests, but it is not limited to unit testing.

Simply using pytest does not automatically make a test a unit test.

A test qualifies as a unit test if it meets the criteria for unit testing:

- It tests a small, isolated piece of code (usually a function or method).
- Each test should be short and focused on one behaviour, condition, or variation.
- Tests should not rely on external dependencies (e.g. databases, APIs). If such dependencies are needed, mocks are often used to simulate them.
- Tests should be independent of each other and runnable in any order.
- Each test should be self-descriptive and easy to understand without needing extra explanation.

So, the type of test depends on what is being tested and how, not on the tool used to write it.

Patterns for naming and writing tests

There are common patterns for testing. Following these make it easier for you to write and run the test code, and for others to understand it.

- Test directory structure
- Test class, function and module names
- Patterns that help you to structure the test (**GIVEN–WHEN–THEN** and **ARRANGE–ACT–ASSERT**)

Test directory structure

There are common patterns for organizing where tests should be placed in a project.

One widely used approach is to place all tests in a separate directory outside the main application code.

This separation helps keep the codebase clean and makes it easier to manage and run tests independently of the application logic.

The pytest documentation describes two typical project layout patterns (https://docs.pytest.org/en/stable/explanation/goodpractices.html#choosing-a-test-layout):

1. Tests placed in directory at the same level as the source code

```
├── pyproject.toml
├── src/
│   ├──package/
│   │   ├── __init__.py
│   │   ├── my_module.py
├── tests/
│   ├── test_my_module.py
```

2. Tests placed within the source code.

```
├── pyproject.toml
├── src/
│   ├── package/
│   │   ├── __init__.py
│   │   ├── my_module.py
│   ├── tests/
│   │   ├── __init__.py
│   │   ├── test_my_module.py
```

The course activities all follow the first of these as it separates the packaging of the application code from the code that tests it.

Test naming

Packages such as pytest will detect test files that follow certain naming patterns.

The directory containing the tests is usually named 'tests' or 'test'.

The test modules file names are prefixed with `test_` such as `test_module.py` or suffixed with `_test` such as `module_test.py`.

The test file name typically also refers to the module, package or feature being tested; though this does not affect auto discovery of tests.

The tests cases within the modules may be written as classes or functions. The naming convention should make it clear what they test, e.g. `class TestMyPackage` is a class containing test cases for the MyPackage package. `test_redirect_success` tests that a call to a given url successfully redirects.

In general, you should have a pretty good idea what the test is testing for from its name. Given this, test names such as `test_function_a_1`, `test_function_a_2` etc. are not considered a good style, while they relate to 'function a', it is not clear what 1 and 2 refer to.

The way that Pytest discovers names is documented here (https://docs.pytest.org/en/stable/explanation/goodpractices.html#conventions-for-python-test-discovery).

GIVEN-WHEN-THEN pattern

The **GIVEN–WHEN–THEN** pattern is a helpful way to think about the behaviour you want to test.

This pattern comes from an approach called Behaviour-Driven Development (BDD). You may come across guides that use theGherkin (https://cucumber.io/docs/gherkin/) syntax to express this pattern. You do not need to learn Gherkin syntax to use the GIVEN–WHEN–THEN structure effectively.

Here's an example:

```
Test Scenario: Simple Google search
    Given the Google home page is displayed
    When the user searches for "Python pandas"
    Then the results page shows links related to "Python pandas"
```

In this course, I often use this pattern in test case documentation or docstrings to describe what the test is doing.

These elements also influence the structure of the test code, which is explained in the next section.

For a more detailed example, see Pytest with eric (https://pytest-with-eric.com/bdd/pytest-bdd/).

ARRANGE-ACT-ASSERT pattern

The **ARRANGE–ACT–ASSERT** pattern is a common structure for writing unit tests, and is referenced in the pytest documentation.

It aligns closely with the **GIVEN–WHEN–THEN** pattern:

- Arrange ≈ Given
- Act ≈ When
- Assert ≈ Then

This pattern helps structure your test code clearly:

1. **Arrange**: Set up everything the test needs before running

   Examples: initialise objects, create test data, mock external services such as login to a web app.

2. **Act**: Perform the action being tested.

   For example: call a function.

3. **Assert**: Check that the result matches expectations. Assertions determine if the test passes or fails.

   This could be a simple value check or a more complex validation.

Here's an example using Python's built-in absolute value function (https://www.w3schools.com/python/ref_func_abs.asp):

```
def test_abs_for_a_negative_number():
    # Arrange: define a negative number to be tests
    negative = -6

    # Act: call the function to be tested and pass the negative number
    answer = abs(negative)

    # Assert: use a pytest assertion, in this case the 'correct'
        behaviour from the function
    # when called on -6 is to return 6
    assert answer == 6
```

Next activity (9-02-pytest-tests.md)

2. Writing unit tests with Pytest

Test case structure

Given the guidance in the introduction, here is a suggested approach for writing a unit test:

1. **Write a descriptive test function name**
    - Test functions should start with the prefix `test_`.
    - Use a name that clearly describes what is being tested
      e.g. `test_add_success_with_integers` is more informative than `test_add`, as it describes both the function and the behaviour being tested.
2. **Add a docstring**.

   Consider include a description that uses the 'GIVEN > WHEN > THEN' pattern to clearly document the purpose of the test.

```
def test_valid_email():
"""
    GIVEN valid values an Admin object, email='test@test.com' and
        password='testpassword'
    WHEN the Admin object 'admin' is created
    THEN admin.email should equal 'test@test.com'
"""
```

3. **Arrange**: Set up the test conditions
    - Provide the necessary values or objects e.g. `email = 'test@test.com`, `password = 'testpassword'`
4. **Act**: Call the function or method being tested:
    - e.g. `admin = Admin(email, password)`
5. **Assert**: Check that the result is as expected.
    - Use an assertion to verify the outcome e.g. `assert admin.email == email`
6. **Run the test** and check that it passes.
    - There are different ways to run tests:
        - VS Code green run icon (https://code.visualstudio.com/docs/python/testing#_run-tests)
        - PyCharm green run icon (https://www.jetbrains.com/help/pycharm/pytest.html#create-pytest-test)
        - Add command to a main() function
        - From the command line in the venv terminal. If you run `pytest -v` it will look for all the tests it can autodiscover and run them. You can specify folders, modules or even specific test functions, e.g.: `pytest -v tests/test_modulename.py::test_valid_email`

These activities give command line instructions only, but you can use any method.

An example is in test_playing_cards.py (../../tests/playing_cards/test_playing_cards.py).

Testing errors and exceptions

Write assertions that are expected to pass.

Don't write assertions just to make them fail deliberately.

The purpose of a test is to verify that the code behaves as expected. A test should only fail if there is an actual issue with the code being tested.

Writing tests that are designed to fail can be confusing and misleading, unless you're specifically testing error handling or failure scenarios—and even then, the assertion should still reflect the expected behaviour in that context.

For example, **do not** write tests like this:

```
def test_addition_with_invalid_data():
    """
    GIVEN two string values 'a' and 'b'
    WHEN they are passed to an addition function expecting integers
    THEN the test will fail because the input is invalid
    """
    # Arrange
    a = "a"
    b = "b"

    # Act
    result = a + b  # This will succeed in Python, but it's not
        meaningful for an integer addition function

    # Assert
    assert result == 2  # This will fail because 'a' + 'b' results in
        'ab', not 2
```

The syntax to test exceptions (https://docs.pytest.org/en/7.1.x/how-to/assert.html#assertions-about-expected-exceptions) is different.

Example from the `pytest` documentation:

```
def test_zero_division():
    with pytest.raises(ZeroDivisionError):
        1 / 0
```

Given this, the bad test example above is rewritten as:

```
import pytest

import pytest


def test_add_raises_type_error_on_strings():
    """
    GIVEN two string inputs 'a' and 'b'
    WHEN passed to the add() function
    THEN a TypeError should be raised
    """
    # Arrange
    a = "a"
    b = "b"

    # Act & Assert
    with pytest.raises(TypeError):
        add(a, b)
```

Activities

1. Run a test

1. Look at the code in test_playing_cards.py (../../tests/playing_cards/test_playing_cards.py).

   It contains:

   - `Rank` and `Suit`: SQLModel classes that store the suit and rank values used in playing cards
   - `Deck`: a Python class functions including `deal`, `shuffle`, etc.
   - `create_cards_db`: a Python function that creates database with Rank and Suit and saves to the test folder. It returns an error if the database is not created.

2. Run the test and check it passes: `pytest -v tests/test_playing_cards.py::test_suit_returns_suitstring`

2. Run a test that fails

You want all tests to pass. If a test fails, read the output to try and identify the problem.

1. Run the test using the option `-v` which gives a more detailed test output `pytest -v tests/test_playing_cards.py::test_suit_returns_suitstring`
2. Read the output and try to work out what the error message is telling you
3. Fix the bug in the code and re-run the test. Did it pass?

3. Write a test

1. Complete the code for `test_deck_cards_count()`
2. Run the test and check if it passes

Add at least one more test, e.g., one for the Rank class similar to Suit, or test one of the methods of the Deck class. Try to write unit tests if you can.

4. Write a test that raises an exception

1. Complete the code for `test_create_cards_db_raises_on_invalid_path()`
2. Run the test and check if it passes

5. Use IDE or Copilot to generate a unit test

If you have copilot enabled in your IDE, try generating a test automatically.

Right-click on a class or function name in the code and find the option to generate a test:

- PyCharm: Generate... | Test
- VS Code: Copilot | Generate tests...

Review the generated code. Do you agree with it?

Run the tests and see if they run.

Further practice

If you want to practice further, write tests for some of the paralymics model classes or functions created in earlier activities (e.g. data prep in week 2, database in week 4, classes in week 5).

Next activity (9-03-fixtures.md)

3. Using Pytest fixtures

If you find yourself writing tests where you use the same set-up ('arrange') steps, you can create reusable 'fixtures'.

Pytest fixtures (https://docs.pytest.org/en/stable/how-to/fixtures.htmle) are used to provide common functions that you may need for your tests. They are created (set up, yield) and removed (tear down, finalise) using the `@fixture` decorator.

Fixtures are established for a particular scope using the syntax `@pytest.fixture(scope='module')`. Options for scope are:

- `function` fixture is executed/run once per test function (if no scope is specified then this is the default)
- `class` one fixture is created per class of tests (if creating test classes)
- `module` fixture is created once per module (e.g., a test file)
- `session` one fixture is created for the entire test session

You may not need to use fixtures for COMP0035 coursework, however, you will need to use in COMP0034, so it is a good idea to learn and practice now.

Fixtures can be added either within the test file (module) or in a separate python file called `conftest.py`. Placing them in `conftest.py` to make them available to other test modules. `conftest.py` is typically placed in the root of the `tests` directory, though you can have multiple `conftest.py` files (not covered here).

Activity: Create a fixture

The tests `test_deal_hand_return_amount()` and `test_deck_cards_count()` both need a deck of cards to be created.

It would be useful to create a fixture that creates a deck of cards that can be used by tests that need it.

1. Add code to create the following fixture in `tests/conftest.py`:

```
import pytest
from activities.starter.playing_cards import Suit, Rank, Deck


@pytest.fixture
def deck_cards():
    suit_values = [Suit(suit=s) for s in ['Clubs', 'Diamonds', 'Hearts',
        'Spades']]
    rank_values = [Rank(rank=str(r)) for r in
                   [2, 3, 4, 5, 6, 7, 8, 9, 10, 'Jack', 'Queen', 'King',
        'Ace']]
    deck_cards = Deck(suits=suit_values, ranks=rank_values)
    yield deck_cards
```

Activity: Modify the tests to use the fixture

Update the code in the test module so that the test functions use the fixture.

You can pass a fixture to the function like this: `def test_query_select_succeeds(db):` where a fixture called `db` is being passed to the function.

1. Modify the `test_deck_cards_count()` test function to use the fixture.

```
def test_deck_cards_count(deck_cards):
    # Arrange: deck_cards now comes from the fixture
    deck_length = len(deck_cards.deck)  # Act
    assert deck_length == 52  # Assert
```

2. Repeat for the `test_deal_hand_return_amount()` test.

3. Run the tests and check the output.

Database fixtures

Although a test that relies on an external component such as a database is not considered a unit test, it is still a test.

You will be using databases in COMP0034.

You would not want tests to change data in a live database, so you would create a temporary database.

For small databases like the card database, you can create a database in memory, i.e. that is not stored as a file.

The database can then be recreated for each test so that any changes to the database from one test don't impact another.

The SQLModel example (https://sqlmodel.tiangolo.com/tutorial/fastapi/tests/?h=test#pytest-fixtures) shows a fixture which can be adapted for the cards database:

```
import pytest
from sqlmodel import create_engine, SQLModel, Session
from sqlmodel.pool import StaticPool

from activities.starter.playing_cards import create_cards


@pytest.fixture(name="session")
def session_fixture():
    # Create the cards
    suits, ranks, cards = create_cards()

    # Create the session and yield
    engine = create_engine(
        "sqlite://", connect_args={"check_same_thread": False},
        poolclass=StaticPool
    )
    SQLModel.metadata.create_all(engine)
    with Session(engine) as session:
        # Add the objects to the database
        session.add_all(suits)
        session.add_all(ranks)
        session.add_all(cards)
        session.commit()
        yield session
```

Activity: Create and use a database fixture

The following code is in test_playing_cards.py (../../tests/playing_cards/test_playing_cards.py):

```
def test_select_returns_cards():
    """ Test that database returns results when the cards table is
        queried

    GIVEN an existing database in memory
    WHEN a query is made to the cards table
    THEN it should return a result set with 52 rows
    """
    db_path = ":memory:"
    engine = create_cards_db(db_path=db_path)
    with Session(engine) as session:
        statement = select(CardModel)
        result = session.exec(statement)
        cards = result.all()
        assert len(cards) == 52
```

1. Create a fixture that returns the session
2. Modify `test_select_returns_cards()` to use the session fixture
3. Run the test and check it works

Next activity (9-04-coverage.md)

4. Coverage

What is coverage?

In software testing, coverage refers to a metric that measures the extent to which the codebase is tested by a set of test cases. It helps ensure that the tests are validating the functionality and quality of the software.

Theoretically, higher code coverage should correlate with fewer defects, but this depends on the quality and scope of the tests. Unit testing alone is unlikely to be sufficient to catch all kinds of errors.

Aiming for 100% coverage is not usually practical due to the cost and time involved; and it may not be necessary depending on the potential impact of system failure. This Google blog post (https://testing.googleblog.com/2020/08/code-coverage-best-practices.html) provides a summary of the implications of coverage. In it, they suggest the following guidelines for their organisation: > ... at Google we offer the general guidelines of 60% as "acceptable", 75% as "commendable" and 90% as "exemplary".

These guidelines are meaningful in their organisation for the scale, scope and nature of their applications. Other organisations and situation will likely have different metrics. For example, sonar (https://www.sonarsource.com/resources/library/code-coverage/) suggest a "typically accepted goal" is 80%.

For coursework 2 your code base is very small in comparison to these examples, so achieving 100% is realistic.

Python coverage tools

Tools that measure code coverage report their results to a specified output, such as the terminal, a text file, or an HTML report.

Popular tools include:

- `coverage.py` (https://pypi.org/project/coverage/) is a widely used Python coverage tool
- `pytest-cov` (https://pypi.org/project/pytest-cov/) extends coverage with additional reporting.

These tools typically report on:

- Statement coverage: how many lines or statements in the codebase are executed by tests.
- Uncovered lines: identifying which lines of code are not exercised by any test.
- Branch coverage: evaluates whether all possible branches (e.g., paths through if statements) are tested. Branches represent different execution paths, and branch coverage shows how many of these are covered.

Some tools may also report on other types of coverage, such as function coverage or condition coverage, depending on their capabilities.

Pytest coverage using pytest-cov

Coverage can be configured to run within your IDE. Look in your IDE's documentation.

To run in the venv terminal, include coverage in the command to run pytest. The following would run pytest and produce a coverage report to show how well any tests it finds cover the all the python code in the project. It reports the result in the terminal.

```
pytest -v --cov
```

Some of the options that can be specified include:

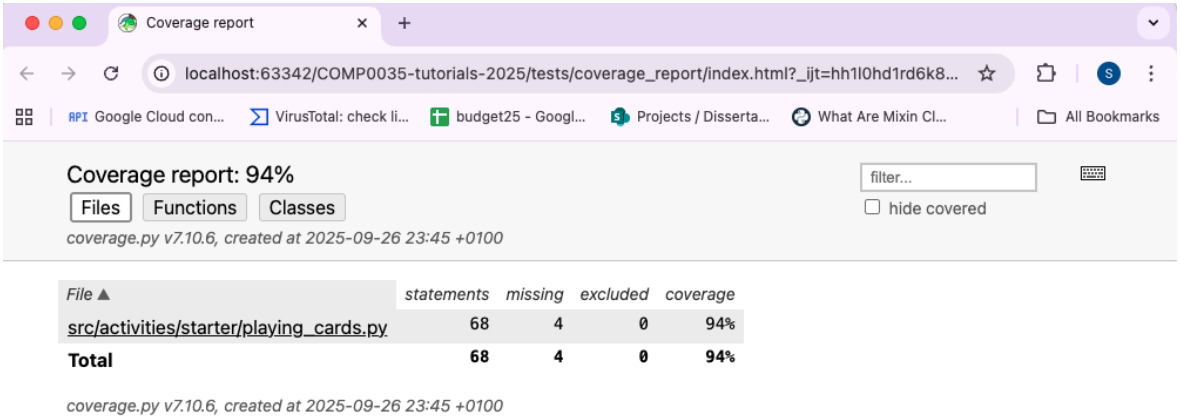
- `--cov=path_to_your_code` to specify what code should be included when assessing how far the tests cover it. Don't run coverage on the test code itself!
- `--cov-report=html:coverage_report` to specify the output format (`html`) and the folder name to generate the html to (coverage_report)
- `--cov-report=term-missing` specify the type of report, `term-missing` indicates code not covered (missed) by the tests
- `--cov-branch` report on branch coverage

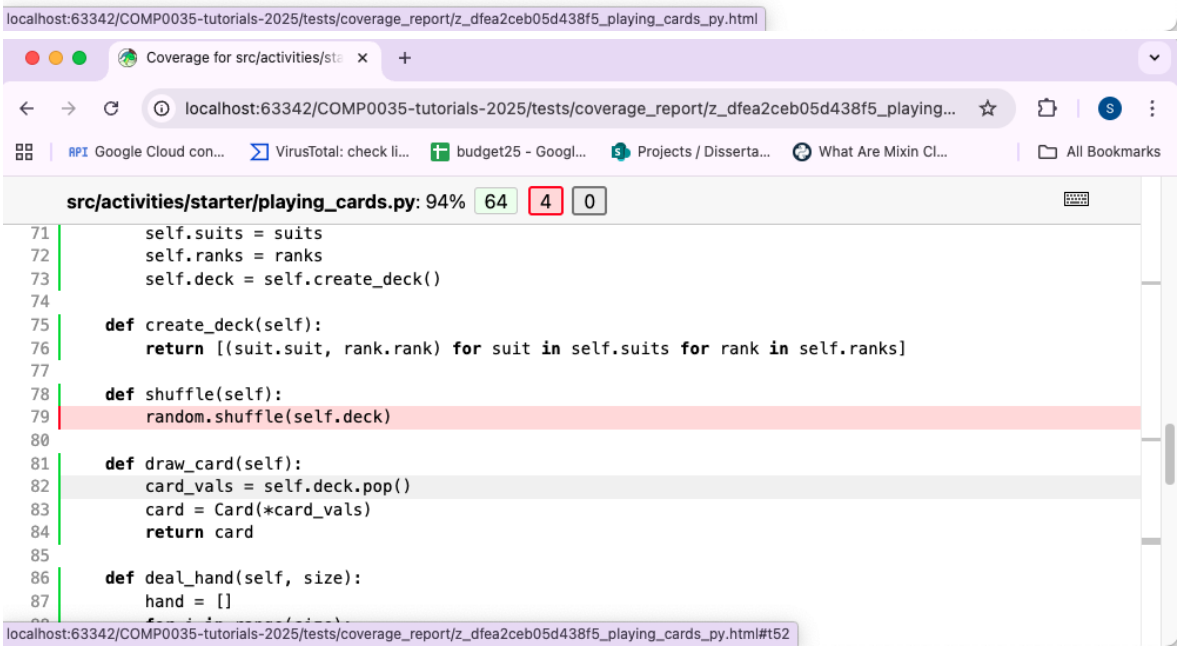
Activity: Report on coverage and assess the output

Run the tests you have written for this tutorial again with different options:

1. `pytest tests/test_playing_cards.py --cov`
2. `pytest tests/test_playing_cards.py --cov=activities.starter.playing_cards`
3. `pytest tests/test_playing_cards.py --cov-report=term-missing --cov=activities.starter.playing_cards`
4. `pytest tests/test_playing_cards.py --cov-report=html:tests/coverage_report --cov=activities.starter.playing_cards`

   After you run this, find the coverage_report folder that was created in tests/coverage_report. Find index.html and open in a browser. The file, function, class are hyperlinks and if you click through to a file, function or class not 100% you will see the code that is not tested is highlighted.





5. `pytest tests/test_playing_cards.py --cov-branch --cov-report=term-missing --cov=activities.starter.playing_cards`

My thoughts on what the report contents give you:

1. 96% Report incorrectly includes the test code itself
2. 94% Only checks how far the tests written test the code in playing_cards.py which is more accurate than 1.
3. 94% As for 2, only now you also know which lines have not been tested 32, 79 and 109-110.
4. 94% As for 3 but more visually and easily lets you click through the HTML to find the code that is not tested.
5. 95% As for 2 but also shows how many branches are missed. Branch is a route through the code, so for example branches missed could one side of an if statement not executed.

Assessing tests solely on coverage % has limitations:

- High coverage doesn't guarantee good tests. You might have tests that execute code but don't actually assert correct behavior.
- Coverage tools measure execution, not intent. They don't know if your tests are checking edge cases, error handling, or business logic. You could have 100% coverage but miss critical scenarios.

- Coverage tools typically focus on unit tests. They don't measure whether components work together correctly (integration testing), or whether the system behaves as expected in real-world scenarios.
- If you only measure line coverage, you might miss untested branches (e.g., if/else paths). Branch coverage helps, but even that may not catch complex logical conditions or data-driven bugs.
- Some code (e.g., logging, trivial getters/setters) may not need tests. Trying to achieve 100% coverage can waste time on low-value tests.
- Developers could write superficial tests just to increase coverage.

In your coursework course try to be critical in reviewing coverage:

- If not 100% what code is not covered? Are critical functions and high-risk areas well-covered? Is it important to write tests to cover these?
- Have you considered edge cases? Error paths? Invalid inputs?
- Are your assertions and test cases meaningful?

Next activity (9-05-ci-github.md)

5. Running tests in GitHub Actions

Continuous Integration (CI) is a development practice where code changes are automatically built, tested, and validated as soon as they're committed to a repository.

GitHub provides a way to configure and run tests on a given action, such as when a new commit is made to your repository. Their tool is **GitHub Actions**.

How to use GitHub Actions is documented on GitHub's site (https://docs.github.com/en/actions/writing-workflows).

Creating and editing a workflow was covered in activity 4-03 (../4_code_quality/4-03-github-actions.md)

Activity: Create a GitHub Actions workflow

- Go to your repository on GitHub
- Go to the **Actions** tab
- If you already have a workflow, then look for the 'New workflow' button; otherwise go to next step
- Find the workflow named **'Python package'** or **Python application** (both have similar steps) and click on **' Configure'**
    - You will see a workflow `.yml` file generated on the screen. Edit this with the following changes:
        - In the section `name: Install dependencies` at the end of this section but before the `name: Lint with Flake 8` add a line to install your code `pip install -e .` and make sure pytest-cov is installed

```
- name: Install dependencies
  run: |
    python -m pip install --upgrade pip
    pip install flake8 pytest pytest-cov
    if [ -f requirements.txt ]; then pip install -r
        requirements.txt; fi
    pip install -e .
```

        - In the section `- name: Test with pytest`, edit the line that runs pytest to one that also runs coverage, e.g. `pytest tests/test_playing_cards.py --cov-branch --cov-report=term-missing --cov=activities.starter.playing_cards`
        - Find the **'commit changes...'** button which is likely to top right of the screen and press it. Change the message if you wish and then **'Commit changes'** again.

This workflow will now run every time you push a change to GitHub. This is useful as it runs all your tests so you can see if new code you have written breaks any previously working functionality.

> By default, GitHub Actions sends an email when the flow fails which you might find annoying!. You can change the notification settings (https://docs.github.com/en/account-and-profile/managing-subscriptions-and-notifications-on-github/setting-up-notifications/configuring-notifications#) in GitHub.

View the workflow results

To view the results of the workflow:

1. Go to the Action tab in your GitHub repository.
2. There should now be at least one workflow run. If all went well it has a green tick, if there were issues there will be a red cross. If there is a Amber/Yellow circle then the workflow is still running.
3. Click on the tick/cross/circle on the workflow.
4. Click on the tick/cross/circle on 'build'.
5. You should now see headings that correspond to the `name:` sections in the .yml file.
6. Expand the `Test with pytest` section. The output should look similar to what you saw when you ran pytest from the terminal.

If there is a red cross then find the section at step 5 that has the red cross, expand it and see what the error message it. It should say what failed and why. You will then need to fix the error.

Note: if the tests fail with a 'module not found' error this is likely due to either not adding the line `pip install -e .` in the installation section of the .yml, or an issue with the content of pyproject.toml.

Next activity (9-06-further.md)

7. Going further

Paralympics example tests

This week's lecture included a demo of unit testing using pytest for a model class for the paralympics database. These tests are included in paralympics (../../tests/paralympics).

There are also tests that use a pytest fixture with an instance of a database. The fixture is used in tests for a query class.

Further information

There is more to unit testing with Pytest than can be covered in one tutorial.

If you are interested, investigate other aspects such as:

- tests for error conditions (exceptions) (https://docs.pytest.org/en/7.1.x/how-to/assert.html#assertions-about-expected-exceptions)
- edge/boundary cases i.e. handling extreme values of valid limits, and just beyond (pytest-with-eric tutorial (https://pytest-with-eric.com/introduction/python-testing-strategy/#Boundary-Conditions))
- organising tests in classes (https://stackoverflow.com/questions/50016862/grouping-tests-in-pytest-classes-vs-plain-functions) rather than functions.
- parameterised tests (https://docs.pytest.org/en/latest/how-to/parametrize.html)
- doctests using the docstrings (https://realpython.com/python-doctest/)
- use of mocks - e.g. pytest-with-eric tutorial (https://pytest-with-eric.com/pytest-advanced/pytest-mocking/)

Other tutorials:

- RealPython: Effective testing with Python (https://realpython.com/pytest-python-testing/)
- freeCodeCamp: Python testing for beginners (video) (https://www.youtube.com/watch?v=cHYq1MRoyI0) - includes mocks and using ChatGPT for testing