Introduction to pytest

pytest

- Defacto standard testing tool for Python
- unittest is in the standard library, pytest is better
- pytest is a command line tool for collecting and running tests
- Also a framework for writing tests
- Extendable by plugins (not very hard to write), for example pytest-asyncio for async testing
- Very widely used, lots of documentation and videos
- The purpose of testing is to ***verify behaviour***
- *https://pytest.org/*

Installing pytest

- Install pytest into a virtual environment
- pipenv is commonly used to manage environments and dependencies
- Pipfile and Pipfile.lock specify dependencies
- Install and create the virtual environment with "***pipenv install***"
- Activate the virtual environment with "*pipenv shell*"

```
✔ Successfully created virtual environment!
Virtualenv location: /home/michael/.local/share/virtualenvs/pytest-fzR5mxNh
Installing dependencies from Pipfile.lock (ed1b2d)...
To activate this project's virtualenv, run pipenv shell.
Alternatively, run a command inside the virtualenv with pipenv run.
michael@lappy:~/code/talks/pytest$ pipenv shell
Launching subshell in virtual environment...
michael@lappy:~/code/talks/pytest$  . /home/michael/.local/share/virtualenvs/
(pytest) michael@lappy:~/code/talks/pytest$ which python
/home/michael/.local/share/virtualenvs/pytest-fzR5mxNh/bin/python
(pytest) michael@lappy:~/code/talks/pytest$ which pip
/home/michael/.local/share/virtualenvs/pytest-fzR5mxNh/bin/pip
```

Creating a Test Suite

- Test collection is done with a naming convention:
- Write tests as functions* in files called "*test_something.py*" (etc)
- They probably live in a project directory called "*tests*"
- Run the tests with pytest
- A "*test suite*" is a collection of tests found from test files

(*) Tests can be collected in classes or generated. Test functions are most common though.

Test Functions

- Functions should be named "*test_something*" as well
- Use the assert statement to verify something
  - `assert actual_value == expected_value`
- The test fails with a useful error message if an assert fails or something goes wrong (an exception raised)

```
def test_function():
    result = 1 + 2
    assert result == 3

def test_failing_test():
    result = 1 + 2
    assert result == 4
```

Test Run

```
(pytest) michael@lappy:~/code/talks/pytest$ pytest
============================= test session starts ==============================
platform linux -- Python 3.10.12, pytest-8.2.0, pluggy-1.5.0
rootdir: /home/michael/code/talks/pytest
collected 2 items

test_first.py .F                                                         [100%]

=================================== FAILURES ===================================
_____________________________ test_failing_test _____________________________

    def test_failing_test():
        result = 1 + 2
>       assert result == 4
E       assert 3 == 4

test_first.py:9: AssertionError
=========================== short test summary info ============================
FAILED test_first.py::test_failing_test - assert 3 == 4
========================== 1 failed, 1 passed in 0.02s =========================
(pytest) michael@lappy:~/code/talks/pytest$
```

Different Types of Test

- Types of test (jargon and usage can vary):
  - **Unit tests**
    - Short tests to verify behaviour of individual components, written by devs
    - Test the unit of behaviour not implementation
    - Should be fast and ideally not use external resources
    - Often depend on mocking
  - **Functional tests**
    - QA tests are usually functional tests
    - Also called end to end tests
    - The most useful tests (for CI/CD and refactoring)
    - Verify behaviour from the point of view of the user
    - Do as little mocking as possible
  - **Integration tests**
    - Verify connection between components
    - Sometimes a synonym for functional tests
    - Avoid over testing and skip this layer...

Setting up the System Under Test

- The code you're testing is the "*System under test*"
- It usually needs setting up before you can test it
  - You might need to run a server
  - You might need to provide or populate test data
  - You might need to mock out some external services for the tests to work
- We can setup the system under test using pytest **fixtures**

Fixtures

- Test functions specify test fixtures as parameters
- Fixtures are made available once they've been imported
- When a test is run the fixture is called by pytest and passed into the function for you
- A common place to put them is *conftest.py* which pytest always checks
  - But fixtures and other test helpers may live in separate modules
- Mark the function as fixture a with the pytest.fixture decorator
- Fixtures can also take fixtures, so you can build them on top of each other
- Fixtures can be hard to trace in code, don't go overboard! Avoid "*fixture hell*" (fixtures that take fixtures that take fixtures that take …)

A test_client Fixture

- These fixtures starts a web app running (using connexion 3 and Flask 3) and return a client for testing
- The fixtures in *conftest.py* are automatically called and the result passed to the "*test_app*" function

```
from app import create_app
import pytest

@pytest.fixture
def app(event_loop):
    return create_app()

@pytest.fixture
def test_client(app):
    return app.test_client()
```

conftest.py

```
def test_app(test_client):
   response = test_client.get('/healthz/live')
   assert response.json() == {'response': 'Healthy'}
```

test_second.py

Parameterise Tests

- Running tests with combinations of inputs (and expected outputs) can be done with parameterize decorator

```
import pytest
from operator import add

sample_test_cases = [
    # (x, y, result)
    (1, 2, 3),
    (0, 0, 0),
    (-1, 2, 1),
]

@pytest.mark.parametrize("x,y,result", sample_test_cases)
def test_add(x, y, result):
    assert add(x, y) == result
```

Note: similar to `subTest` in `unittest`

Parameterise Tests

- pytest creates a test case for every input

```
(pytest) michael@lappy:~/code/talks/pytest$ pytest -vv -k test_add
================================= test session starts ==================================
platform linux -- Python 3.10.12, pytest-8.2.0, pluggy-1.5.0 -- /home/michael/.local/s
hare/virtualenvs/pytest-fzR5mxNh/bin/python
cachedir: .pytest_cache
rootdir: /home/michael/code/talks/pytest
plugins: anyio-4.3.0, asyncio-0.23.6
asyncio: mode=strict
collected 6 items / 3 deselected / 3 selected

test_third.py::test_add[1-2-3] PASSED                                          [ 33%]
test_third.py::test_add[0-0-0] PASSED                                          [ 66%]
test_third.py::test_add[-1-2-1] PASSED                                         [100%]

================================= warnings summary ==================================
```

Marking Tests

- We can mark tests to group them and only run a subset, like smoke tests or slow tests (etc)
- Check the documentation for some useful standard marks:
  - **skip** – always skip the test
  - **skipIf** – skip on a condition
  - **xfail** – mark a test that is expected to fail (it will raise an error if the test passes)
- We can also use custom marks like "slow" or "smoke_test"

Marking Tests

```
import pytest
import sys
import time

@pytest.mark.skipif(sys.platform=='win32', reason="Skipped
on windoze")
def test_not_windows():
    assert sys.platform != 'win32'

@pytest.mark.smoke_test
def test_core_functionality(test_client):
    def test_app(test_client):
        response = test_client.get('/healthz/live')
        assert response.status == 200

@pytest.mark.slow
def test_slow():
    time.sleep(10)
    assert True
```

Command Line Arguments

- `pytest -k test_name`
  - Select test functions or file by [partial] name
- `pytest -v/-vv`
  - Verbose (and more verbose) test run output
- `pytest -m smoke_test`
  - Run a subset of tests with marks
- `pytest --runslow`
  - Custom command line options
  - Setup in conftest.py (see example code)
- `pytest --help/-h`

Testing Exceptions

- Test error handling/exceptions with `pytest.raises`
- If no exception is raised, or a different type of exception, the test fails

```
import pytest

expected_message = 'can only concatenate str (not "int") to
str'

def test_exception():
    with pytest.raises(TypeError) as exc_info:
        "3" + 4

    assert str(exc_info.value) == expected_message
```