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
```

Asserts

- Any comparison operator can be used in an assert
  - ➔ ==, !=, <, >, <=, >=, is, is not

```
def test_not_equals():
    bad_result = 'not this'

    assert 'actual result' != bad_result


def test_comparisons():
    result = 6
    assert result > 5
    assert result < 9
    assert result >= 6
    assert result <= 6
```

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 "s*ystem under test*"
- "*Pure*" functions are easy to test without mocking – functions without state or side effects
  - So decoupled, modular components are more "testable". Testable code tends to be better code.
- 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
  - Fixtures and test helpers may live in separate modules (every test suite becomes a test framework)
- Mark the function as fixture 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

```python
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

```python
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"

```
collected 11 items / 7 deselected / 4 selected

test_four.py::test_not_windows PASSED                                    [ 25%]
test_four.py::test_core_functionality PASSED                             [ 50%]
test_four.py::test_slow SKIPPED (need --runslow option to run)           [ 75%]
test_four.py::test_expected_failure XFAIL                                [100%]
```

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`

Additional options include: --junitxml, --maxfail, --lf etc. See docs.

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_msg = '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_msg
```

Useful Testing (and coding) Advice

- https://opensource.com/article/17/5/30-best-practices-software-development-and-testing

Intermittently failing tests erode the value of your test suite, to the point in which eventually everyone ignores test run results because there's always something failing. Fixing or deleting intermittently failing tests is painful, but worth the effort.

Generally, particularly in tests, wait for a specific change rather than sleeping for an arbitrary amount of time. Voodoo sleeps are hard to understand and slow down your test suite.

Always see your test fail at least once. Put a deliberate bug in and make sure it fails, or run the test before the behavior under test is complete. Otherwise you don't know that you're really testing anything. Accidentally writing tests that actually don't test anything or that can never fail is easy.

Mocking

- Sometimes code calls external services in ways you don't want in tests
- We can use mock objects, from `unittest.mock`
- pytest has its own monkeypatching, unittest.mock is better (or use pytest-mock plugin)
- Mock objects can be configured
  - Setting attributes (or deleting them)
  - Setting return values
  - Side effects like raising exceptions
- Mock objects can be interrogated to see how they've been used (calls and calls to child mocks)
- We use the "AAA" pattern: Arrange, Act, Assert
  - **Arrange**: setup the system under tests (including mocking)
  - **Act**: call the system under test
  - **Assert**: assert the system behaved correctly

unittest.mock.MagicMock

```
>>> from unittest.mock import MagicMock, sentinel, call
>>>
>>> def frobulate(frobulator):
...     return frobulator.frobulate(frobulator.data)
...
>>> mock_frobulator = MagicMock()
>>> mock_frobulator.data = sentinel.MOCK_DATA
>>> mock_frobulator.frobulate.return_value = sentinel.RESULT
>>>
>>> result = frobulate(mock_frobulator)
>>>
>>> assert result == sentinel.RESULT
>>> expected_calls = [call.frobulate(sentinel.MOCK_DATA)]
>>> assert mock_frobulator.mock_calls == expected_calls
```

```
mock_frobulator.frobulate.assert_called_once_with(sentinel.MOCK_DATA)
```

Patching

- patch can be used as a decorator or context manager
- You specify the target of the patch as a string (the import path)
- The target is temporarily replaced with a mock object

```
>>> from unittest.mock import patch
>>> import time
>>>
>>> with patch('time.sleep') as mock_sleep:
...     mock_sleep.return_value = None
...     time.sleep(10)
...
>>> time.sleep
<built-in function sleep>
>>> mock_sleep.assert_called_once_with(10)
```

Namespaces are one honking great idea -- let's do more of those!
(The Zen of Python, Tim Peters, "import this")

Patch Target

- patch the namespace where the target is **used**
- patch the namespace *where the name is looked up*

Wrong!

```
>>> from unittest.mock import patch
>>> from time import sleep
>>>
>>> print(__name__)
__main__
>>> with patch('time.sleep') as mock_sleep:
...     sleep(5)
...
```

Right!

```
>>> from unittest.mock import patch
>>> from time import sleep
>>>
>>> with patch('__main__.sleep') as mock_sleep:
...     sleep(5)
...
```

Patching as a Fixture

- We can do the patching in a fixture, as a generator
- The patching is done automatically by the fixture
- We use AsyncMock as it can be await'ed
- Use just as a normal fixture, the patching is hidden

```
@pytest.fixture
def mock_get_sf_client():
    with patch(
        'app.get_salesforce_client', mock_class=AsyncMock
    ) as mock_get_sf_client:

        yield mock_get_sf_client
        # Optional teardown of fixture after test completes
```

For more examples see:
https://docs.python.org/3/library/unittest.mock-examples.html

Fixture Teardown

- Fixture teardown (cleaning up afterwards) can be done with "*finalizers*" using the `request` fixture
- Use the `request` fixture and call `addfinalizer`
- Generally cleaner and simpler to use `yield` in fixtures

```python
@pytest.fixture
def setup_data(request):
    print("Setting up test data...")

    # Add some data to the database
    data = 5

    # Define a finalizer function for teardown
    def finalizer():
        # Clean up the data
        print("Cleaning up test data...")

    # Register the finalizer to ensure cleanup
    request.addfinalizer(finalizer)
    return data  # Provide the data to the test
```

Fixture Teardown

- Use `-rP` command line option to show stdout (prints) from passing tests and `--disable-warnings` to silence warnings

```
(pytest) michael@lappy:~/code/talks/pytest$ pytest -rP -k test_setup_data --disable-warnings
========================================= test session starts ==========================================
platform linux -- Python 3.10.12, pytest-8.2.0, pluggy-1.5.0
rootdir: /home/michael/code/talks/pytest
configfile: pytest.ini
plugins: anyio-4.3.0, asyncio-0.23.6
asyncio: mode=strict
collected 24 items / 23 deselected / 1 selected

test_seven.py .                                                                          [100%]

============================================ PASSES ============================================
_____________________________________ test_setup_data_fixture _____________________________________
---------------------------------- Captured stdout setup -----------------------------------

Setting up test data...
---------------------------------- Captured stdout teardown ----------------------------------

Cleaning up test data...
====================== 1 passed, 23 deselected, 7 warnings in 0.03s ======================
```

More on Fixtures

- Fixtures are normally recreated (and torn down) for every test.
- Some fixtures (database setup for example) you might want to do just once for a test run
- Some fixtures (putting test data into the database and cleaning it up) might be specific to a few tests – grouped in a class or a module
- Ideally write tests that are isolated from each other and don't depend on one another

  ➔ Test isolation is an important principle, ideally tests should leave the system in the same state they found it

  ➔ Tests that only work if you run them in a certain order are a recipe for pain

Fixture Scope

- We can control fixture setup and teardown by setting the "scope": *function, class, module, package* or *session*.
  ➔ The default scope for a fixture is "*function*"
  ➔ `scope` is a keyword argument to `pytest.fixture`

```
@pytest.fixture(scope="class")
def class_fixture():
    print(f"Calling class fixture at {time.time()}")


class TestClassOne:

    def test_one(self, class_fixture, module_fixture,
                 session_fixture):
        print(f"{self.__class__.__name__}: Test one")
        assert True
```

See the output of test_eight.py