Pytest

Pytest is a widely-used Python testing framework designed to make writing, running, and organising tests simple and expressive. It supports Python 3.10+ and provides a rich plugin ecosystem (over 1300+ plugins), automatic test discovery, and advanced tooling for writing maintainable and scalable test suites.

Key Features of Pytest

Advanced Assertion Introspection

Pytest enhances Python's built-in assert statement.
When a test fails, pytest prints detailed information about **intermediate values**, so you don't need to remember legacy methods like self.assertEqual().

Example failure output:

```
E       assert 4 == 5
E        + where 4 = func(3)
```

This is useful when debugging data processing, packet extraction, or comparing ML outputs.

Automatic Test Discovery

Pytest automatically finds tests in files named: test_*.py and *_test.py.
This allows a clean structure for tests related to:

- PCAP reading
- Feature extraction
- Data preprocessing
- Machine learning model training
- Dashboard functions

Modular Fixture System

Fixtures in pytest help to:

- Reuse setup logic
- Provide test data (e.g., dummy PCAP packets)
- Manage temporary directories
- Control ML model setup
- Mock network traffic

Fixtures can also be:

- Parametrised
- Autouse
- Scoped (function, class, module, package, session)

In our project we can have fixtures for fake PCAP files, clean datasets and ML models.

Exception Testing with pytest.raises

Used to assert that code raises the correct exception.

Example:

```
with pytest.raises(SystemExit):
    f()
```

Useful when testing:

- invalid PCAP files
- unsupported protocols
- empty traffic flows
- ML model errors

Test Grouping with Classes

Tests can be grouped in classes starting with Test for better organisation.

Example:

```
class TestExtraction:
    def test_packet_size(self):
        assert extract_size(p) == 128
```

Note:
Each test gets a new class instance, ensuring isolation.

Floating-Point Comparisons with pytest.approx()

Used for ML predictions or statistical calculations that may contain floating-point rounding errors. Useful for ML model testing.

Example:

```
assert (0.1 + 0.2) == pytest.approx(0.3)
```

Temporary Directories with tmp_path

Pytest can provide unique temporary directories for, saving processed PCAP files, caching ML intermediate files, writing logs or outputs for dashboard tests

Example:

```
def test_write(tmp_path):
    file = tmp_path / "data.json"
    file.write_text("hello")
```

Built-in Fixtures

Useful built-in fixtures for debugging and testing for network traffic profiler:

- caplog - test logging from PCAP parser
- monkeypatch - mock packet reader or ML model
- tmp_path - store temporary extracted features

Fixture Scope & Resolution

Pytest determines fixture usage based on:

1. scope (function, class, module, package, session)

2. dependencies (fixture requiring another fixture)

3. autouse (always executed)

Fixture Sharing via conftest.py

Fixtures placed in a conftest.py file become available to all tests in that directory.

Directory structure for the project could be:

```
tests/
    conftest.py        # shared fixtures: dummy pcap loader, fake flows
    test_pcap.py
    test_features.py
    test_ml.py
    test_dashboard.py
```

Plugin Architecture

For your project, the most useful ones are:

- **pytest-cov** - pytest plugin for measuring coverage, to ensure PCAP parser + ML code is covered

- **pytest-mock -** thin-wrapper around the mock package for easier use with pytest, for mock packet data and network components.

- **pytest-benchmark -** A ``pytest`` fixture for benchmarking code. It will group the tests into rounds that are calibrated to the chosen timer, helps measure parsing speed.

- **pytest-xdist -** plugin for distributed testing, most importantly across multiple CPUs, to speed up tests on large synthetic datasets

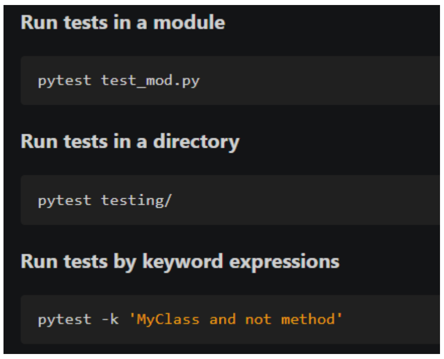
How to invoke pytest

Run with the command:

```
pytest
```

This automatically discovers and runs all tests in files named: test_*.py and *_test.py, in the current directory and all subdirectories.

- Running specific tests



Run tests in a module

```
pytest test_mod.py
```

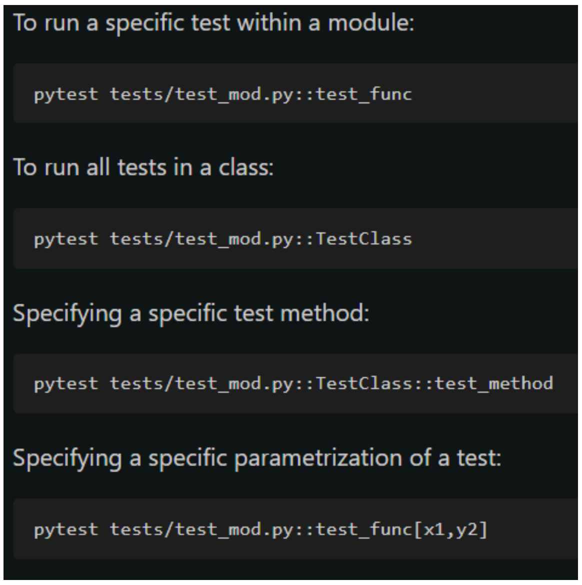
Run tests in a directory

```
pytest testing/
```

Run tests by keyword expressions

```
pytest -k 'MyClass and not method'
```

This runs tests whose names match the expression (case-insensitive).



To run a specific test within a module:

```
pytest tests/test_mod.py::test_func
```

To run all tests in a class:

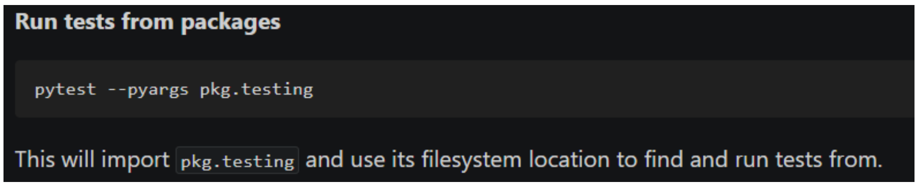
```
pytest tests/test_mod.py::TestClass
```

Specifying a specific test method:

```
pytest tests/test_mod.py::TestClass::test_method
```

Specifying a specific parametrization of a test:

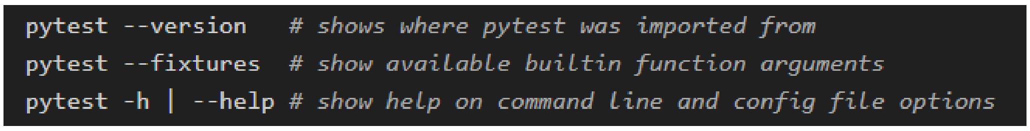
```
pytest tests/test_mod.py::test_func[x1,y2]
```


Run tests from packages

```
pytest --pyargs pkg.testing
```

This will import `pkg.testing` and use its filesystem location to find and run tests from.

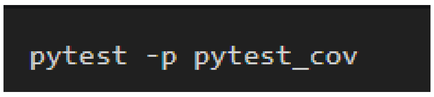
- Useful helper commands



```
pytest --version   # shows where pytest was imported from
pytest --fixtures  # show available builtin function arguments
pytest -h | --help # show help on command line and config file options
```

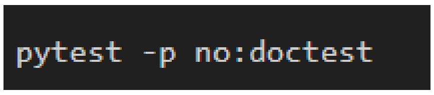
- Managing plugins

Early-load plugin:



```
pytest -p pytest_cov
```

Disable a plugin:



```
pytest -p no:doctest
```

How to monkeypatch/mock

The monkeypatch fixture allows tests to temporarily modify:

- attributes
- functions
- classes
- environment variables
- dictionary items
- the current working directory
- the Python import path

All changes are automatically undone after the test.

Monkeypatch use for the project

1. Mock an external function (e.g., reading PCAPs). Replace real PCAP reader with a fake one:

   monkeypatch.setattr(myreader, "read_pcap", lambda x: sample_data)

   This avoids loading real files in unit tests.

2. Mock returned objects (e.g., fake API responses or ML models). Example for mocking a JSON API call:

   class MockResponse:

```
        @staticmethod

        def json():

            return {"status": "ok"}

    monkeypatch.setattr(requests, "get", lambda url: MockResponse())
```

Useful for dashboard or classification functions.

3. Mock environment variables. Very useful for testing config-based behaviour:

```
monkeypatch.setenv("MODE", "test")

monkeypatch.delenv("MODE", raising=False)
```

Asserts in pytest

Is helpful to verify:

- PCAP feature extraction outputs are correct
- ML model prediction outputs match expectations
- Dashboard functions return correct values
- Errors are raised when files or inputs are invalid

- Basic assert statements

Pytest uses normal Python assert statements:

```
def test_feature_extraction():

    assert extract_features(sample_pcap)["packet_count"] == 120
```

If the test fails, pytest shows a detailed message:

```
E assert 100 == 120

E + where 100 = extract_features(...)
```

- Assertions for expected exceptions (pytest.raises)

For testing error-handling, e.g.: loading a corrupt PCAP file, missing file, invalid user activity label, ML model not loaded.

Example:

```
import pytest

def test_invalid_pcap():
    with pytest.raises(FileNotFoundError):
        load_pcap("nonexistent.pcap")
```

To inspect the exception:

```
with pytest.raises(ValueError) as excinfo:
    parse_user_action(None)
assert "invalid" in str(excinfo.value)
```

Pytest fixtures

Fixtures are reusable setup functions that provide test data or resources. Tests “request” fixtures simply by naming them as function arguments:

```
@pytest.fixture
def sample_pcap():
    return load_test_pcap()



def test_feature_extraction(sample_pcap):
    features = extract_features(sample_pcap)
    assert features["packet_count"] > 0
```

Parametrized tests

Parametrization in pytest lets you run the same test multiple times with different inputs.

- @pytest.mark.parametrize

This lets you pass multiple sets of inputs to one test function.

Example:

```
@pytest.mark.parametrize(
    "pcap_file, expected_packets",
    [("sample1.pcap", 1200), ("sample2.pcap", 530)]
)
def test_pcap_packet_count(pcap_file, expected_packets):
    pcap = load_pcap(pcap_file)

    assert len(pcap.packets) == expected_packets
```

- Parametrizing fixtures

Parametrize fixtures the same way:

```
@pytest.fixture(params=["google.pcap", "youtube.pcap", "netflix.pcap"])

def sample_pcap(request):

    return load_pcap(request.param)
```

Then any test depending on sample_pcap will run once per PCAP.