C206 L06a

Unit Testing using PyTest

L06a Learning objectives

- Analyze the role, characteristics and benefits of unit testing in software quality assurance
- Apply the "Arrange, Act, Assert" pattern to effectively structure unit tests
- Discuss the use of test coverage metrics to assess the comprehensiveness of unit testing
- Configure and use Pytest for writing and automating unit tests

Unit Testing

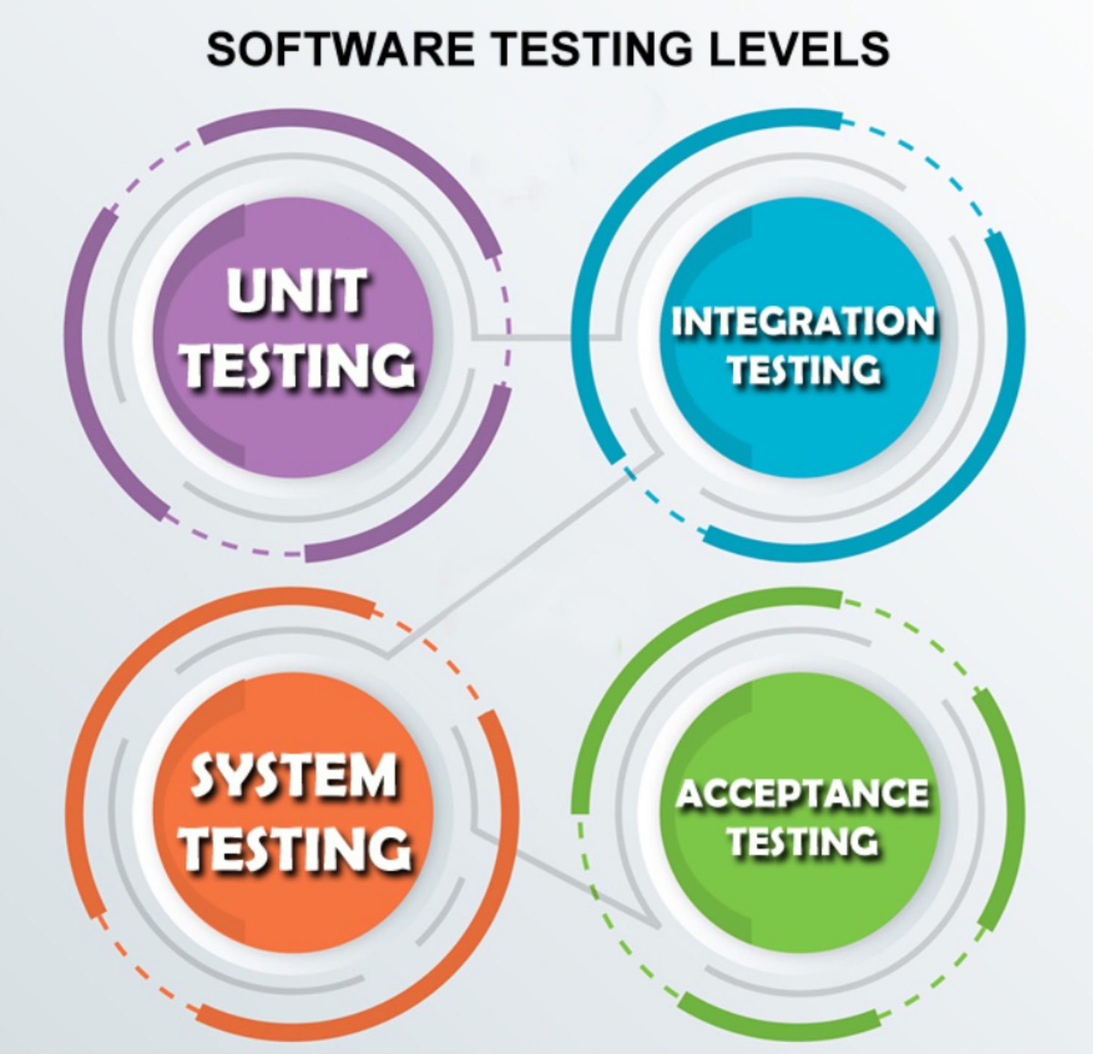
Software Testing Levels - Role of Unit Testing

Software testing levels

1. Unit testing
2. Integration testing
3. System testing
4. Acceptance testing

Unit testing is the first level of testing

- It involves testing individual units or components of the software
- It is performed by **developers** to ensure that each unit functions correctly and as expected
- It is fundamental in identifying and fixing bugs **early** in the development process



Unit Testing in Software Development

Focus - Tests the **smallest** part of the application, like a **method** or **function**

Developer-driven - Written by the same **developers** who write the **application code**

Early Execution - Conducted **early** in the Software Development Life Cycle (SDLC) to catch and fix issues promptly

Multiple tests per method - Covers **various** scenarios including normal, boundary, and error conditions

Automation - Tests are automated for **quick** and **frequent** execution

Benefits of Unit Testing

Early Bug Detection - Finds issues early in the development cycle

Facilitates Code Changes - Safely refactor or update code with less risk

Simplifies Integration - Helps ensure units work correctly before full integration

Acts as Documentation - Shows how the code should work, aiding new team members

Improves Code Quality - Encourages modular and less error-prone coding practices

Benefits of Unit Testing

Enhances Design - Promotes thoughtful design before coding

Cost-Effective - Reduces long-term bug fix and maintenance costs

Enables Continuous Integration / Delivery - Crucial for frequent and reliable code updates

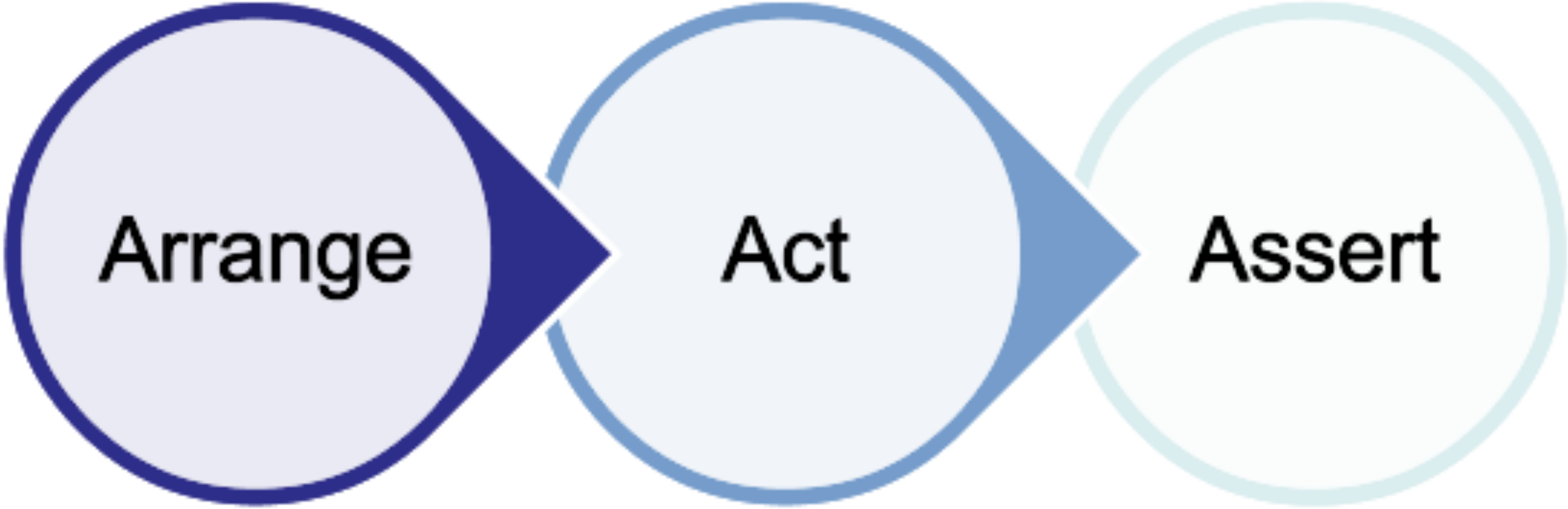
Speeds Up Development - Faster, safer code changes in the long run

Increases Confidence - Provides peace of mind about code reliability

Structure of a unit test case - Arrange, Act, Assert

The structure of a typical unit test can be presented by the AAA pattern

Arrange – Act – Assert



Structure of a unit test case - Arrange, Act, Assert

Arrange

- Set up the testing environment
- Initialize objects, variables, mocks, and any prerequisites necessary for the test
- Prepare the state and inputs for the code to be tested

Act

- Perform the test and execute the code to be tested

Assert

- Verify the outcome of the Act step
- Check if the results meets your expectations
- Assertions are used to compare the actual output of your code against expected values, throwing an error if the test fails

Example of a unit test case - Arrange, Act, Assert

```
class Calculator:
    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b

    def multiply(self, a, b):
        return a * b

    def divide(self, a, b):
        if b == 0:
            raise ZeroDivisionError("Division by zero error")
        return a / b
```

```
class TestCalculator:
    def test_add(self):

        # arrange
        a = 4321
        b = 1234
        cal = Calculator()

        # act
        result = cal.add(a, b)

        # assert
        expected = 5555
        assert result == expected
```

ARRANGE

ACT

ASSERT

Test coverage

- Test coverage is a metric used in software testing to determine how much of the application has been tested
- It can occur at all 4 testing levels: unit, integration, system, or acceptance
- At the integration level, focus might be on interfaces and interactions
- At the system or acceptance level, focus might be on requirements, menu options, screens, or typical business transactions

Unit Testing - 2 types of coverage

Statement coverage

- Measures if every line of code is executed at least once
- Focus: Identifying unexecuted lines

Decision coverage

- Measures if every possible path in a conditional statement is tested
- Focus: Testing all branches of control structures like 'if', 'else', and 'switch'

Statement Coverage

- Statement coverage is also known as line coverage or segment coverage
- Test requirements : all the statements in the program
- Coverage measure : number of executed statements / total number of statements

- Watch this video https://www.youtube.com/watch?v=9PSrhH2gtkU (3m 42s)

Decision Coverage / Branch Coverage

- Decision coverage is also known as branch coverage or all-edges coverage
- Test requirements : all branches in the program
- Coverage measure : number of executed branches / total number of branches

- Watch this video https://www.youtube.com/watch?v=JkJFxPy08rk (4m 17s)

Team discussion (10 min)

- What is a reasonable test coverage percentage?
- Does a higher test coverage percentage represent better code quality?
- Why not go for 100% test coverage?

Introduction To pytest

Unit testing frameworks

Most languages offer native or third-party unit testing tools, emphasizing the importance of testing in software development

Popular unit testing frameworks by programming language

- **Python** - pytest, unittest, nose2
- **JavaScript** - Jest, Mocha, Jasmine
- **Java** - JUnit, TestNG, Mockito
- **C#** - NUnit, xUnit.net, MSTest
- **C++** - Google Test, Boost.Test, Catch2
- **Ruby** - RSpec, Minitest, Test::Unit
- **PHP** - PHPUnit, Behat
- **Go** - Go's Testing Package, Ginkgo, Testify
- **Swift / Objective-C** - XCTest, Quick/Nimble
- **Kotlin / Android** - JUnit, Espresso, MockK

pytest

https://pytest.org/

- The pytest framework makes it easy to write small, readable tests, and can scale to support complex functional testing for applications and libraries
- pytest requires: **Python 3.8+** or PyPy3

Installation

- `pip install pytest`

Version check

- `pytest --version`

Running the tests

- `pytest`

pytest - Test discovery

Complete documentation

- https://docs.pytest.org/en/8.0.x/explanation/goodpractices.html

Simplified and incomplete version

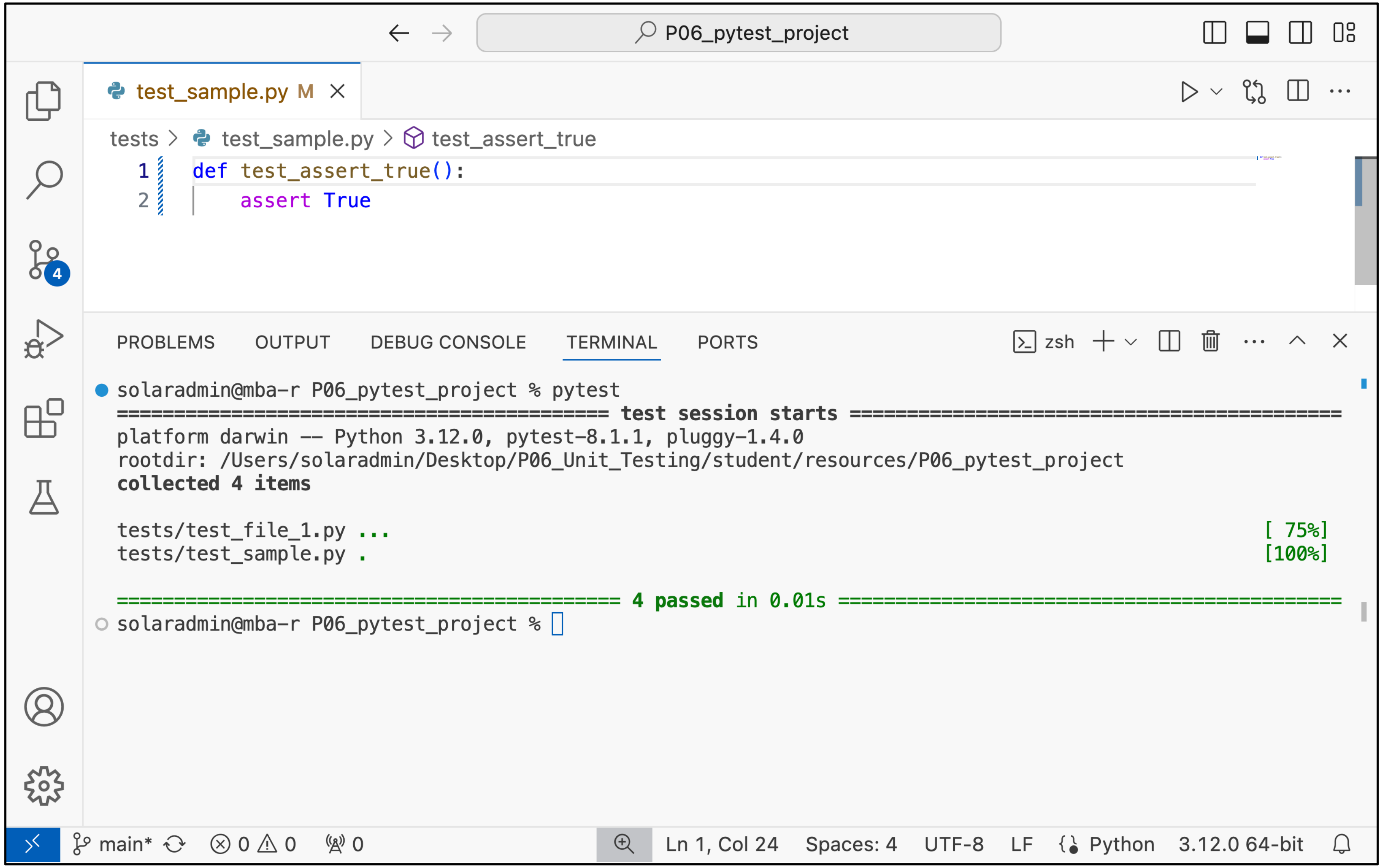
pytest implements the following standard test discovery:

- Recurse into directories in the current directory
- In those directories, search for `test_*.py` or `*_test.py` files

From those files, collect test items:

- `test` prefixed test functions or methods outside of class
- `test` prefixed test functions or methods inside `Test` prefixed test classes

Pass



```python
def test_assert_true():
    assert True
```

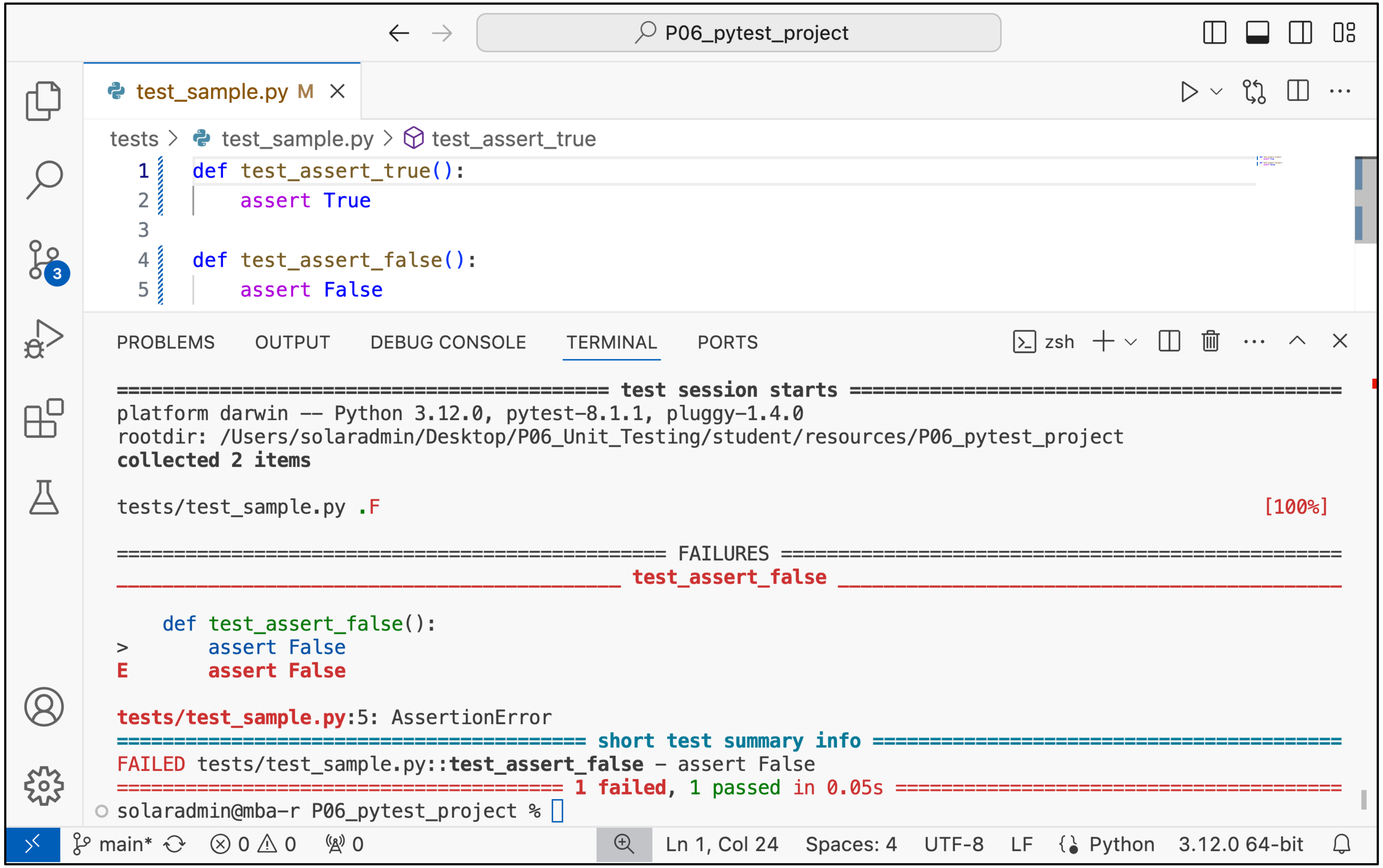
```
solaradmin@mba-r P06_pytest_project % pytest
=========================================== test session starts ============================================
platform darwin -- Python 3.12.0, pytest-8.1.1, pluggy-1.4.0
rootdir: /Users/solaradmin/Desktop/P06_Unit_Testing/student/resources/P06_pytest_project
collected 4 items

tests/test_file_1.py ...                                                                             [ 75%]
tests/test_sample.py .                                                                               [100%]

============================================ 4 passed in 0.01s =============================================
solaradmin@mba-r P06_pytest_project %
```

- A test function passes if the value being asserted evaluates to `True`
- In this example, pytest found 2 test files
  - test_file_1.py with 3 tests (3 dots) which passed
  - test_sample.py with 1 test (1 dot) which passed
- The less output you see, the better
  - no news is good news

Fail



```python
1  def test_assert_true():
2      assert True
3
4  def test_assert_false():
5      assert False
```

```
============================================ test session starts ============================================
platform darwin -- Python 3.12.0, pytest-8.1.1, pluggy-1.4.0
rootdir: /Users/solaradmin/Desktop/P06_Unit_Testing/student/resources/P06_pytest_project
collected 2 items

tests/test_sample.py .F                                                                              [100%]

================================================= FAILURES ==================================================
_____________________________________________ test_assert_false _____________________________________________

    def test_assert_false():
>       assert False
E       assert False

tests/test_sample.py:5: AssertionError
========================================== short test summary info ==========================================
FAILED tests/test_sample.py::test_assert_false - assert False
======================================== 1 failed, 1 passed in 0.05s ========================================
solaradmin@mba-r P06_pytest_project %
```

- A test function fails if the value being asserted evaluates to `False`

- In this example, test_sample.py had
  - 1 test which passed (one dot)
  - 1 test which failed with an AssertionError at line 5 of test_sample.py (one red F)

- Refer to this page for a demo of Python failure reports with pytest

Examples of tests which pass and fail

```python
# the following tests all pass
def test_assert_true():
    assert True

def test_assert_int_value_equality():
    x = 4
    y = 4
    assert x == y

def test_assert_value_equality():
    x = 4.0
    y = 4
    assert x == y

def test_assert_bool_value_equality():
    x = False
    y = 3 == 4
    assert x == y
```

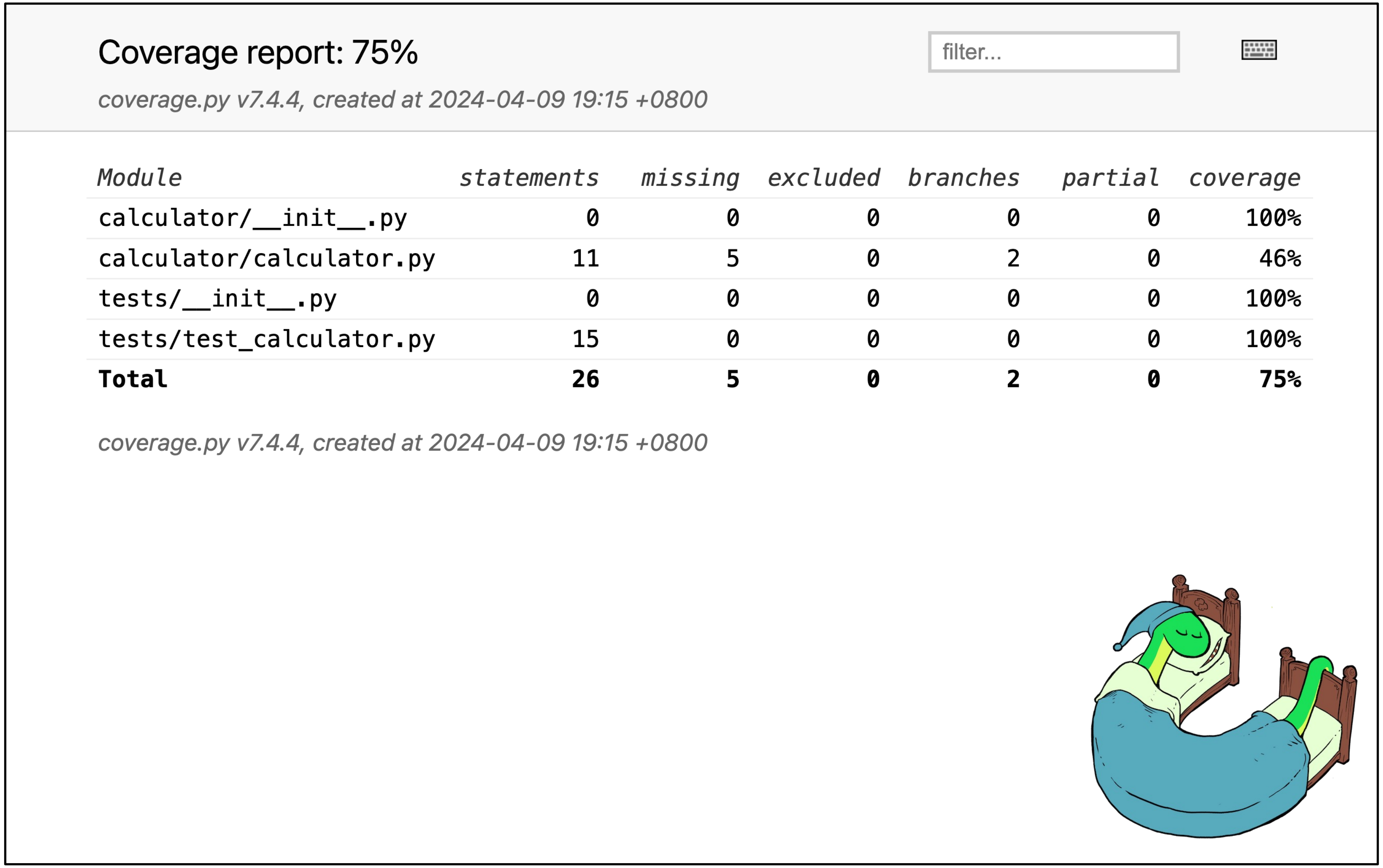
```python
# the following tests all fail
def test_assert_false():
    assert False

def test_assert_int_value_inequality():
    x = 3
    y = 4
    assert x == y

def test_assert_value_inequality():
    x = 4.1
    y = 4
    assert x == y

def test_assert_bool_value_inequality():
    x = True
    y = 3 == 4
    assert x == y
```

coverage.py



Coverage report: 75%

filter...

coverage.py v7.4.4, created at 2024-04-09 19:15 +0800

Module	*statements*	*missing*	*excluded*	*branches*	*partial*	*coverage*
calculator/__init__.py	0	0	0	0	0	100%
calculator/calculator.py	11	5	0	2	0	46%
tests/__init__.py	0	0	0	0	0	100%
tests/test_calculator.py	15	0	0	0	0	100%
Total	**26**	**5**	**0**	**2**	**0**	**75%**

coverage.py v7.4.4, created at 2024-04-09 19:15 +0800

Coverage.py is a tool for measuring statement coverage and branch coverage of Python programs

https://coverage.readthedocs.io/en/latest/index.html

https://coverage.readthedocs.io/en/latest/branch.html

- `pip install coverage`
- `coverage run --branch -m pytest`
- `coverage report -m`
- `coverage html`

Hands on activity - TestCalculator (40 min)

- Let us do unit testing on a python class **Calculator**, which has 4 basic integer calculation methods
- Build a python class **TestCalculator** to have all the test cases for the 4 methods
- Create at least 1 test case for each method in the Calculator class
- Apply the "Arrange, Act, Assert" pattern to effectively structure your unit tests
- Use the assert method to compare the expected result with the actual result
- Run coverage to generate the coverage report
- Refer to L06a Worksheet

```
class Calculator:
    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b

    def multiply(self, a, b):
        return a * b

    def divide(self, a, b):
        if b == 0:
            raise ZeroDivisionError("Division by zero error")
        return a / b
```

Summary

- Analyze the role, characteristics and benefits of unit testing in software quality assurance
- Apply the "Arrange, Act, Assert" pattern to effectively structure unit tests
- Discuss the use of test coverage metrics to assess the comprehensiveness of unit testing
- Configure and use Pytest for writing and automating unit tests