Unit Testing and Test Driven Development

Dushyant Mehra

Unit Testing overview

- Importance of unit testing in software development.
- Writing effective unit tests using python's testing frameworks.
- incorporating test-driven development principles into the development process

Software Review Overview

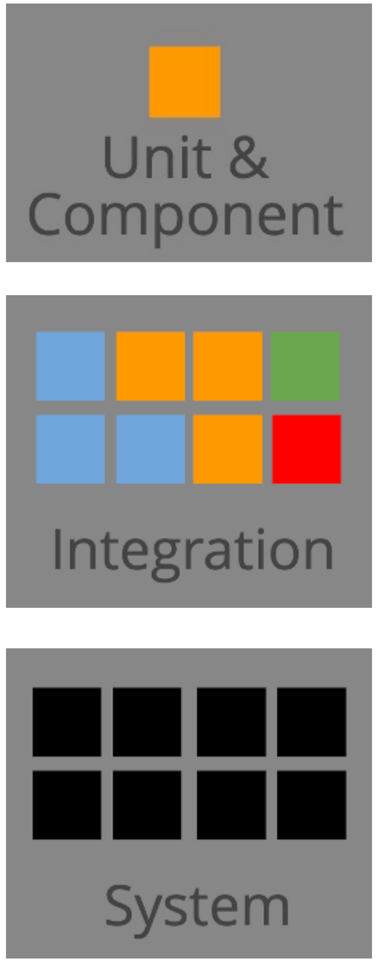
- **Formal reasoning** about a program, usually called *verification* . Verification constructs a formal proof that a program is correct. Verification is tedious to do by hand, and automated tool support for verification is still an active area of research.
- **Code review.** Having somebody else carefully read your code, and reason informally about it, can be a good way to uncover bugs. It’s much like having somebody else proofread an essay you have written. We’ll talk more about code review in the next reading.
- **Testing** . Running the program on carefully selected inputs and checking the results.

Software Testing

- Tests helps ensure code is functioning as expected
- Find defects in codes during the development phase
  - Type tests on inputs
    - Do functions or features have the correct type and number of inputs
  - Testing functionality of code
    - Are functioned executed properly and results are expected
  - Are hardware control systems functioning properly
  - Correcting for indexing or logic errors
  - Prevent code driven system or hardware failure

Types of Tests

- Unit tests
  - Test ‘isolated units’
    - e.g. a method or function
  - Super high coverage
  - Most of the tests
- Integration tests
  - Combine units and test them together
  - Fill in the cracks between the tests
- System or Smoke tests
  - Test with everything plugged together and configured as expected
  - From the end user’s perspective
- Acceptance tests
  - Test the customer’s use cases



What are unit tests and what are they good for?

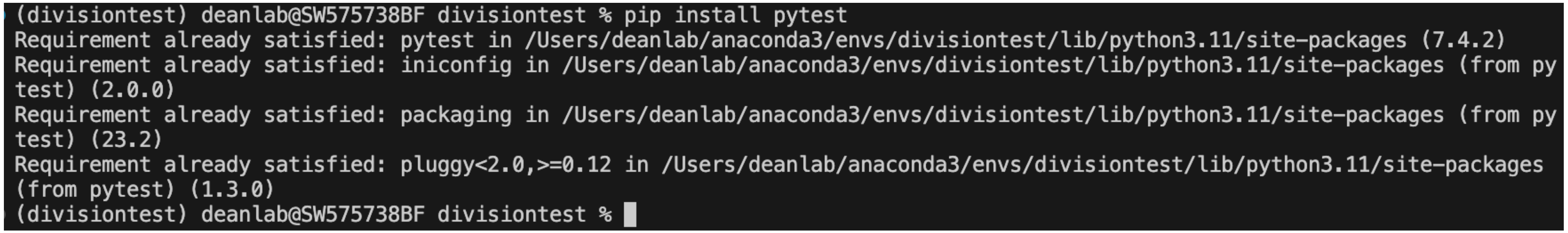
- Tests written for individual units functions or features within a larger code bases to examine functionality
- Writing test reduces bugs in new features and existing features
  - Prevent errors from reoccurring in other sections of code base
  - Ensure that existing features are still functional if new features are included
  - Finding bugs during development
- Faster Debugging
- Faster development and useful cost of change
- Useful in collaborative environment when features are written by different people
  - Writing maintainable code
  - Documenting a developer's intentions

Python Testing Frameworks

- unittest: A testing framework included in the Python standard library. It provides a basic set of tools for writing and executing tests using classes and methods.
- **pytest**: A popular third-party testing framework known for its simplicity, powerful test discovery, and expressive syntax. It encourages test-driven development (TDD) and provides features like fixtures and parameterized testing.
- nose2: A successor to the original nose testing framework, nose2 enhances test discovery and offers various plugins for extending its functionality.
- doctest: A testing framework that allows you to embed tests within docstrings, making it useful for creating documentation that also serves as executable test cases.

Installing Pytest

- Install pytest in specific conda environment
- pip install pytest
- Can also install in pyproject.toml in the dev dependency
- integrate into IDEs such as vscode or PyCharm

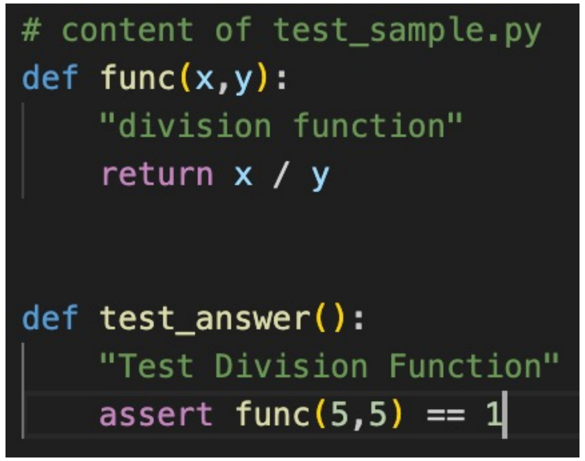


```
(divisiontest) deanlab@SW575738BF divisiontest % pip install pytest
Requirement already satisfied: pytest in /Users/deanlab/anaconda3/envs/divisiontest/lib/python3.11/site-packages (7.4.2)
Requirement already satisfied: iniconfig in /Users/deanlab/anaconda3/envs/divisiontest/lib/python3.11/site-packages (from py
test) (2.0.0)
Requirement already satisfied: packaging in /Users/deanlab/anaconda3/envs/divisiontest/lib/python3.11/site-packages (from py
test) (23.2)
Requirement already satisfied: pluggy<2.0,>=0.12 in /Users/deanlab/anaconda3/envs/divisiontest/lib/python3.11/site-packages
(from pytest) (1.3.0)
(divisiontest) deanlab@SW575738BF divisiontest %
```

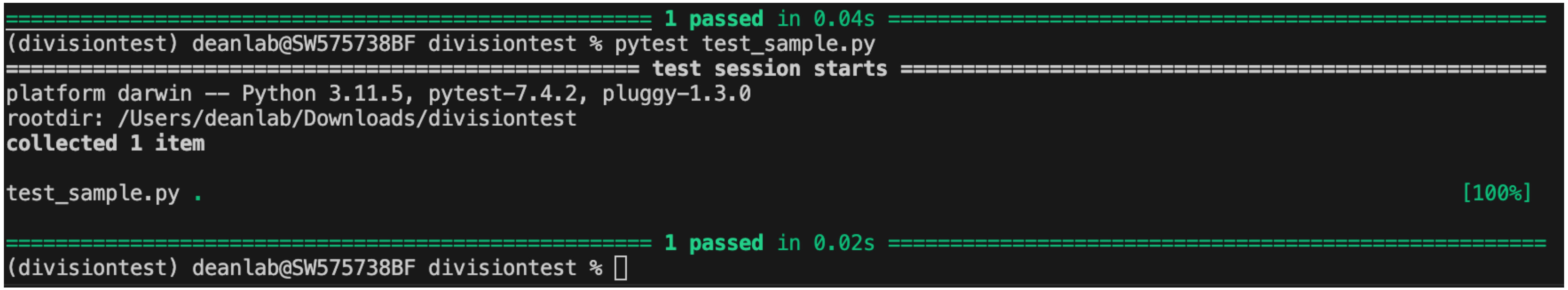
How to Write Unit Test

- **Arrange**: prepare for test, preparing objects, starting or killing services, entering records,
- **Act**: action that we would like to test, function or process
- **Assert**: checking the result state to see if it matches expectations. Look at output and make judgment
- **Cleanup**: cleanup tests so other downstream tests aren’t influenced by results or attributes.

Unit Test Example: Passing Test

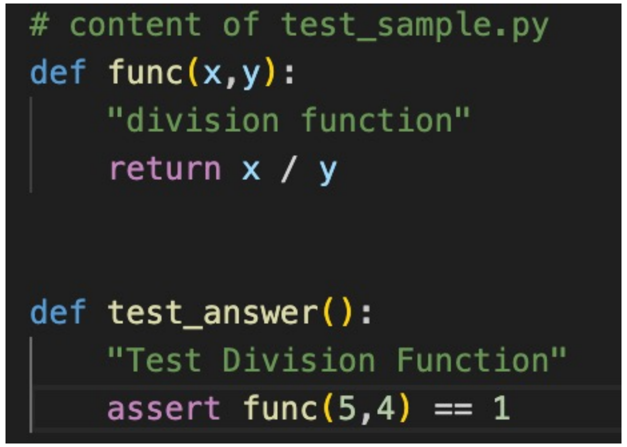


```python
# content of test_sample.py
def func(x,y):
    "division function"
    return x / y


def test_answer():
    "Test Division Function"
    assert func(5,5) == 1
```

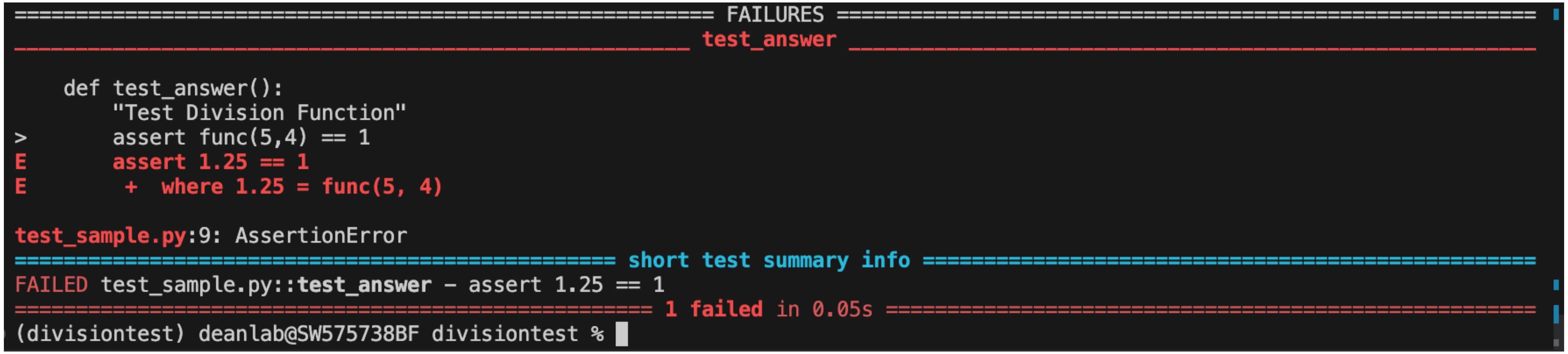


```
=================================================== 1 passed in 0.04s ===================================================
(divisiontest) deanlab@SW575738BF divisiontest % pytest test_sample.py
================================================== test session starts ==================================================
platform darwin -- Python 3.11.5, pytest-7.4.2, pluggy-1.3.0
rootdir: /Users/deanlab/Downloads/divisiontest
collected 1 item

test_sample.py .                                                                                                  [100%]

=================================================== 1 passed in 0.02s ===================================================
(divisiontest) deanlab@SW575738BF divisiontest % 
```

Unit Test Example: Failing Test



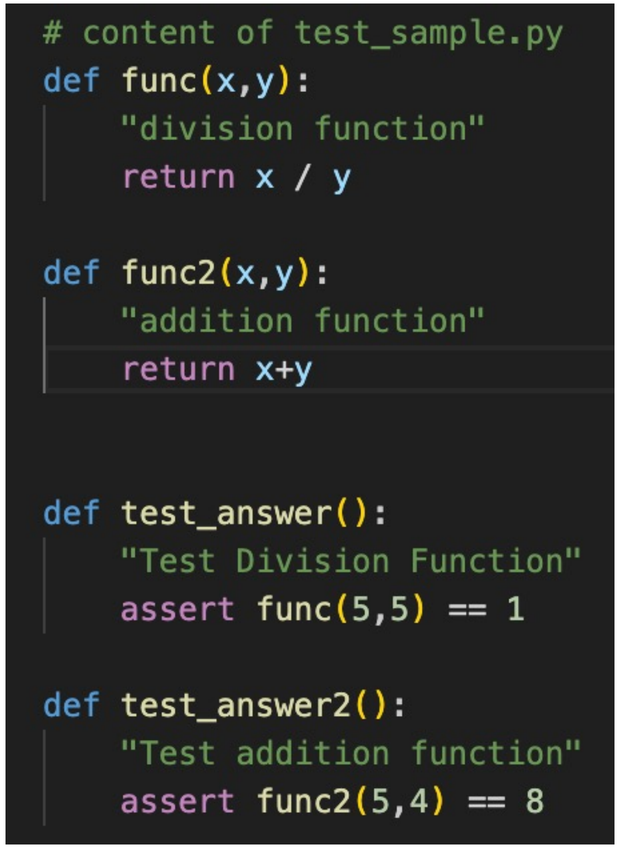
```python
# content of test_sample.py
def func(x,y):
    "division function"
    return x / y


def test_answer():
    "Test Division Function"
    assert func(5,4) == 1
```



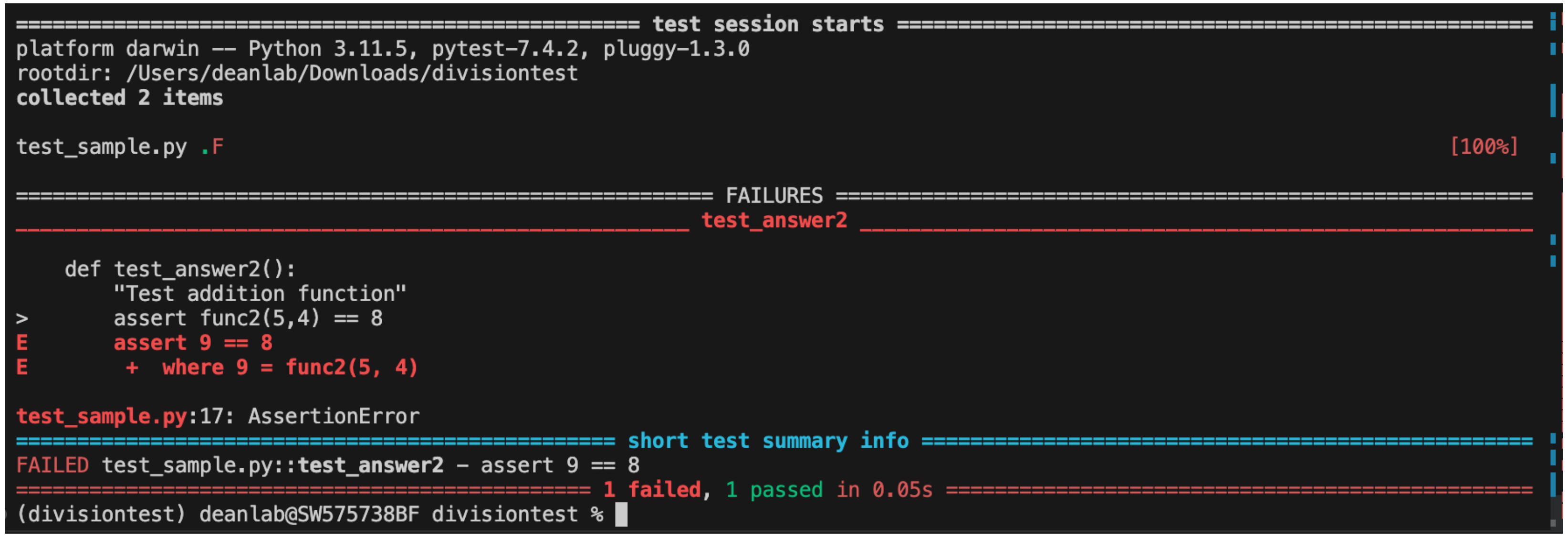
```
========================================================== FAILURES ==========================================================
________________________________________________________ test_answer _________________________________________________________

    def test_answer():
        "Test Division Function"
>       assert func(5,4) == 1
E       assert 1.25 == 1
E        +  where 1.25 = func(5, 4)

test_sample.py:9: AssertionError
================================================== short test summary info ===================================================
FAILED test_sample.py::test_answer - assert 1.25 == 1
===================================================== 1 failed in 0.05s ======================================================
(divisiontest) deanlab@SW575738BF divisiontest % █
```

Recognizing tests using pytest

- Pytest can recognize tests with prefix test_



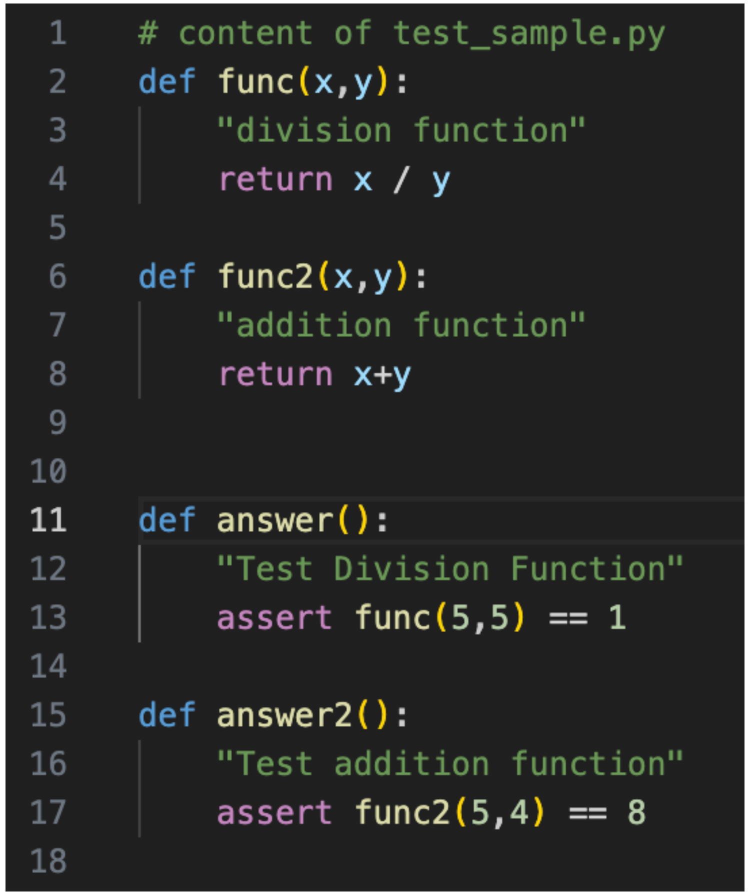
```
# content of test_sample.py
def func(x,y):
    "division function"
    return x / y

def func2(x,y):
    "addition function"
    return x+y


def test_answer():
    "Test Division Function"
    assert func(5,5) == 1

def test_answer2():
    "Test addition function"
    assert func2(5,4) == 8
```



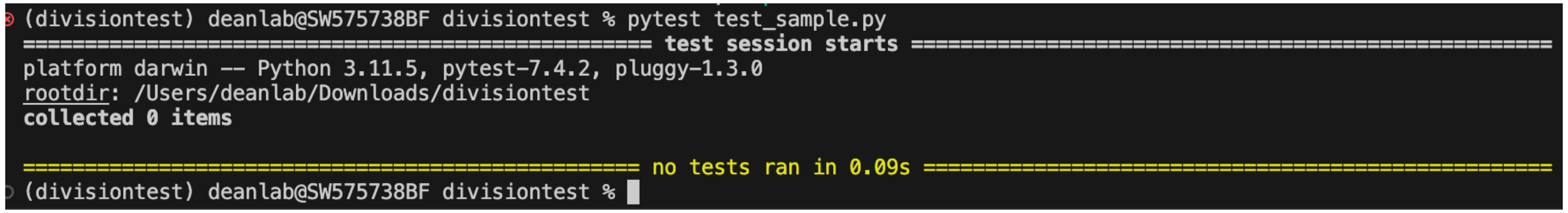
```
================================================== test session starts ===================================================
platform darwin -- Python 3.11.5, pytest-7.4.2, pluggy-1.3.0
rootdir: /Users/deanlab/Downloads/divisiontest
collected 2 items

test_sample.py .F                                                                                                 [100%]

======================================================== FAILURES ========================================================
______________________________________________________ test_answer2 ______________________________________________________

    def test_answer2():
        "Test addition function"
>       assert func2(5,4) == 8
E       assert 9 == 8
E        +  where 9 = func2(5, 4)

test_sample.py:17: AssertionError
================================================ short test summary info =================================================
FAILED test_sample.py::test_answer2 - assert 9 == 8
============================================== 1 failed, 1 passed in 0.05s ===============================================
(divisiontest) deanlab@SW575738BF divisiontest %
```

Recognizing tests using pytest

- Pytest can recognize tests with prefix test_



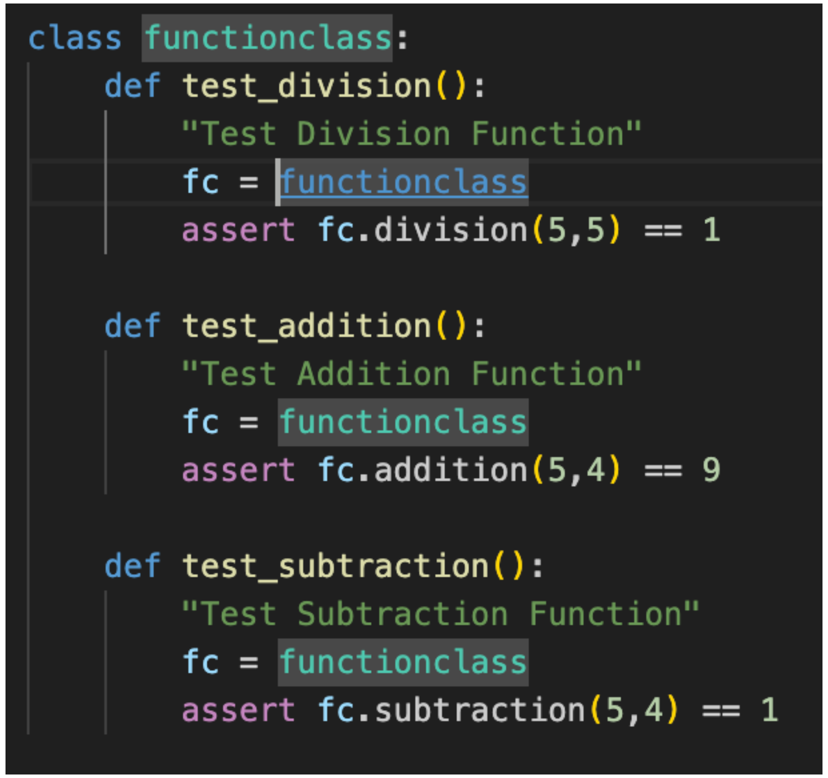
```python
# content of test_sample.py
def func(x,y):
    "division function"
    return x / y

def func2(x,y):
    "addition function"
    return x+y


def answer():
    "Test Division Function"
    assert func(5,5) == 1

def answer2():
    "Test addition function"
    assert func2(5,4) == 8

```



```
(divisiontest) deanlab@SW575738BF divisiontest % pytest test_sample.py
=================================================== test session starts ===================================================
platform darwin -- Python 3.11.5, pytest-7.4.2, pluggy-1.3.0
rootdir: /Users/deanlab/Downloads/divisiontest
collected 0 items

================================================== no tests ran in 0.09s ==================================================
(divisiontest) deanlab@SW575738BF divisiontest %
```

Running tests on classes and using to run multiple tests

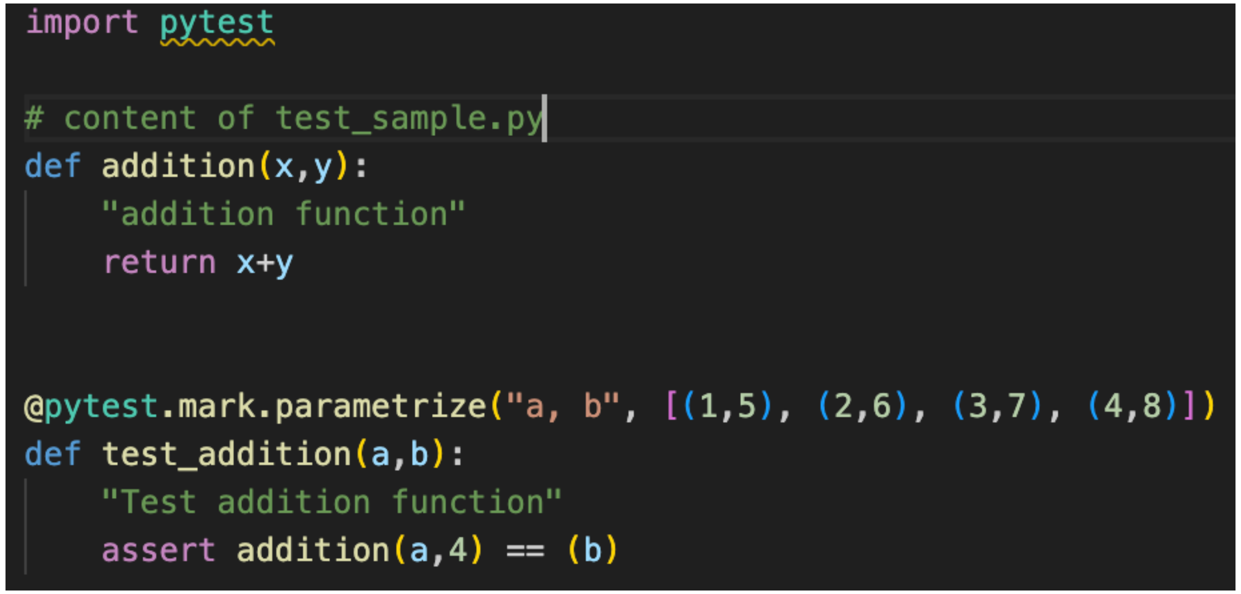
- Can create testing classes to run multiple tests
  - Test organization
  - Sharing fixtures for tests only in that particular class
  - Applying marks at the class level and having them implicitly apply to all tests
- Be careful when having tests within a class because they can share attributes at class levels which can lead to test interdependence



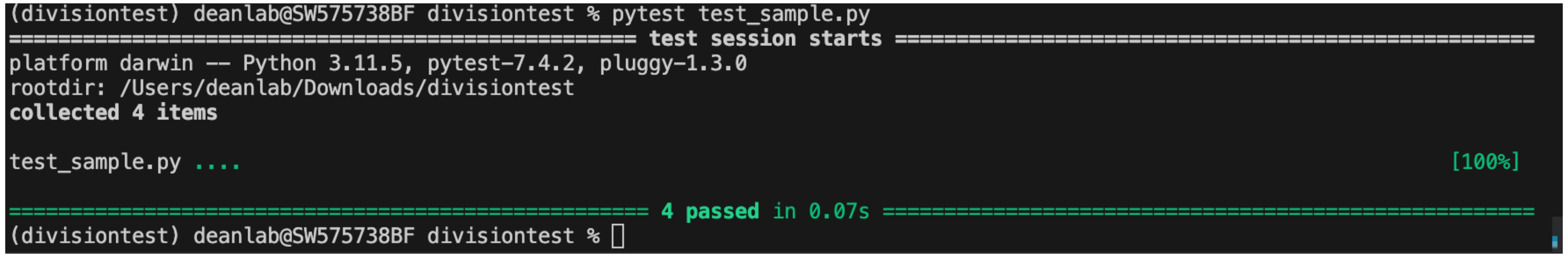
```
class functionclass:
    def test_division():
        "Test Division Function"
        fc = functionclass
        assert fc.division(5,5) == 1

    def test_addition():
        "Test Addition Function"
        fc = functionclass
        assert fc.addition(5,4) == 9

    def test_subtraction():
        "Test Subtraction Function"
        fc = functionclass
        assert fc.subtraction(5,4) == 1
```

Parameterize tests

- Test multiple parameters at once using parameterize



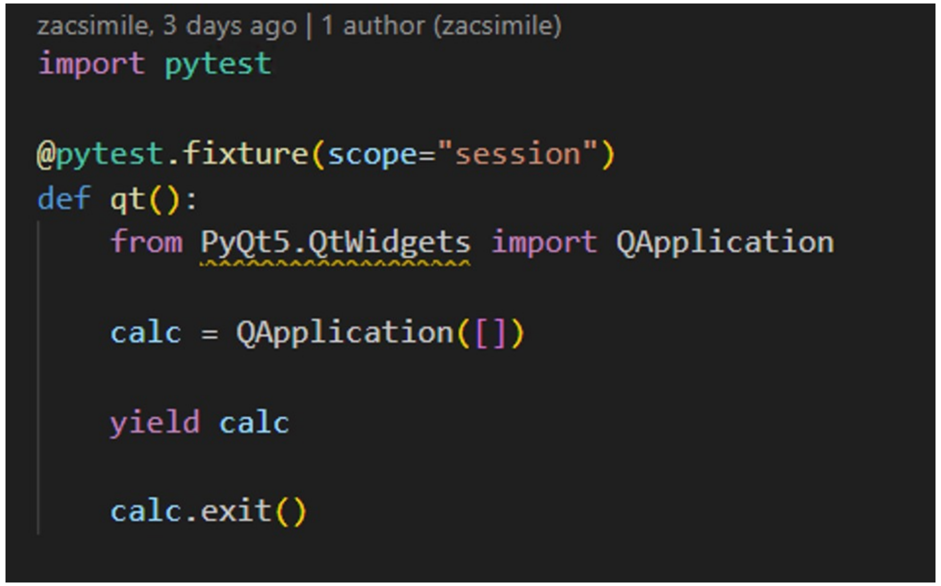
```python
import pytest

# content of test_sample.py
def addition(x,y):
    "addition function"
    return x+y


@pytest.mark.parametrize("a, b", [(1,5), (2,6), (3,7), (4,8)])
def test_addition(a,b):
    "Test addition function"
    assert addition(a,4) == (b)
```



```
(divisiontest) deanlab@SW575738BF divisiontest % pytest test_sample.py
=================================================== test session starts ====================================================
platform darwin -- Python 3.11.5, pytest-7.4.2, pluggy-1.3.0
rootdir: /Users/deanlab/Downloads/divisiontest
collected 4 items

test_sample.py ....                                                                                                   [100%]

==================================================== 4 passed in 0.07s =====================================================
(divisiontest) deanlab@SW575738BF divisiontest % 
```

Pytest Fixtures

- Fixtures are a way to define reusable components that are required by your tests.
- **Pytest** will automagically hook up your fixtures to your tests (or other fixtures!) that require them.
- By default, fixtures are recreated for every test that requires them.
- It is possible to control the lifetime of a fixture (e.g. create it once for all the tests) See https://pytest.org/latest/fixture.html. Setup and teardown fixtures by defining scope
  - Scope can be a Function, Class, Module, or Session



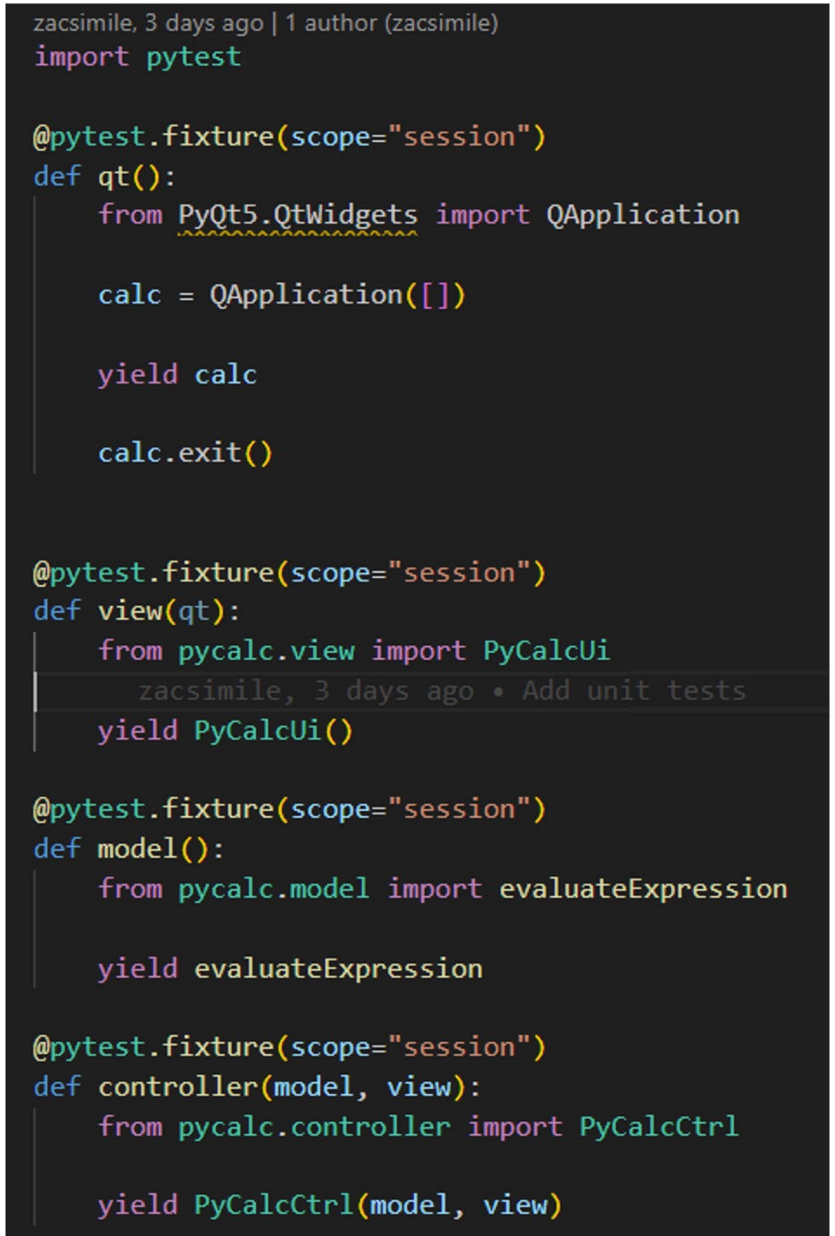
zacsimile, 3 days ago | 1 author (zacsimile)

```
import pytest

@pytest.fixture(scope="session")
def qt():
    from PyQt5.QtWidgets import QApplication

    calc = QApplication([])

    yield calc

    calc.exit()
```

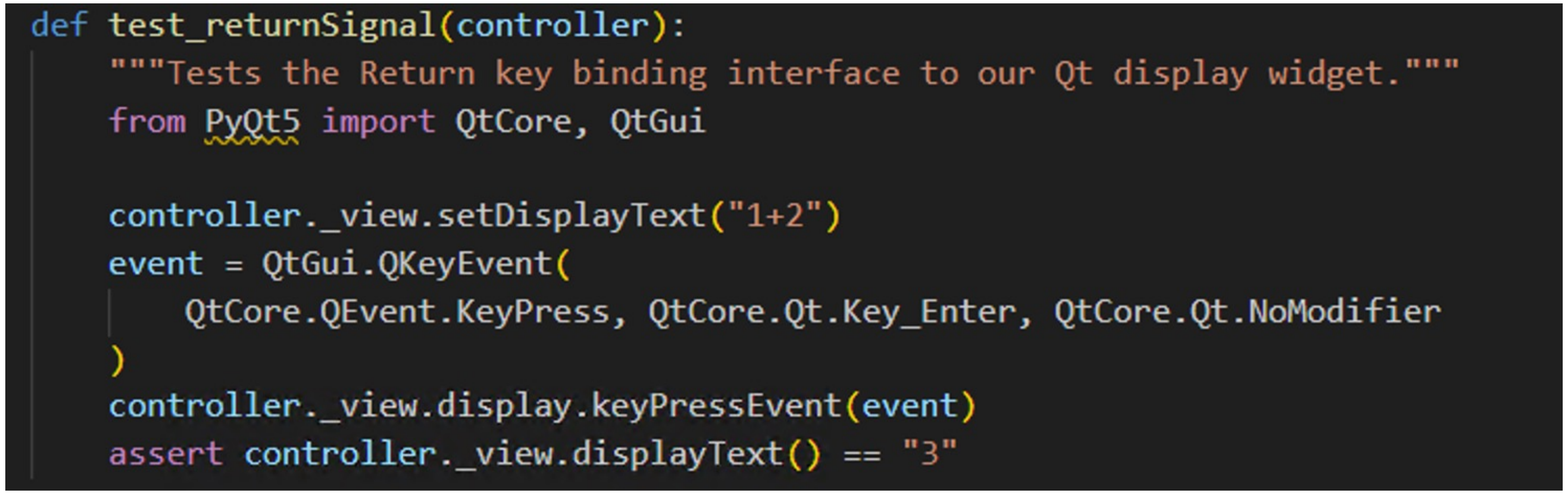
Adding Fixtures using Conftest.py

- In pytest: Conftest.py can be used to define fixtures for unit test.
- **Fixtures**: Define fixtures for static data used by tests. This data can be accessed by all tests in the suite unless specified otherwise. This could be data as well as helpers of modules which will be passed to all tests.
- **External plugin loading**: conftest.py is used to import external plugins or modules. By defining the following global variable, pytest will load the module and make it available for its test. Plugins are generally files defined in your project or other modules which might be needed in your tests.
- **Hooks**: You can specify hooks such as setup and teardown methods and much more to improve your tests.

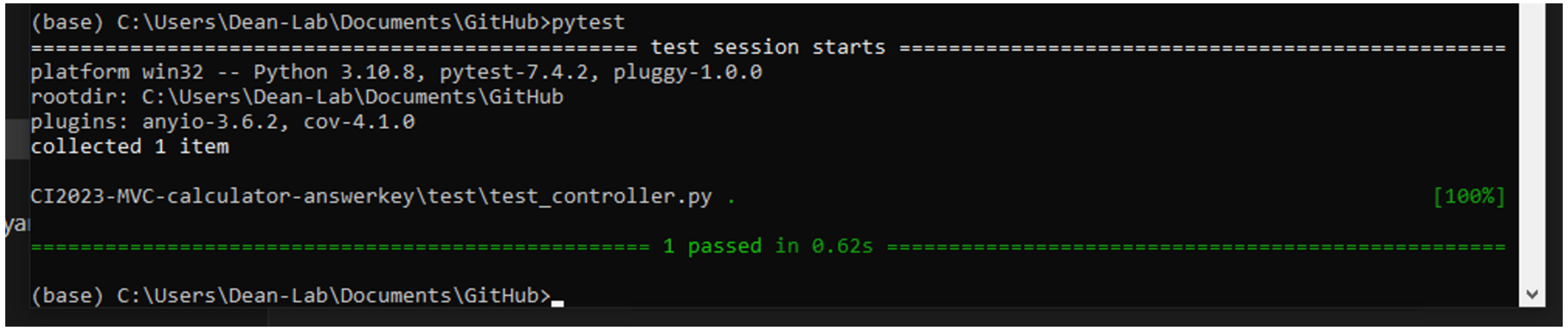
Testing Binding of Key in Calculator



```
zacsimile, 3 days ago | 1 author (zacsimile)
import pytest

@pytest.fixture(scope="session")
def qt():
    from PyQt5.QtWidgets import QApplication

    calc = QApplication([])

    yield calc

    calc.exit()


@pytest.fixture(scope="session")
def view(qt):
    from pycalc.view import PyCalcUi
        zacsimile, 3 days ago • Add unit tests
    yield PyCalcUi()

@pytest.fixture(scope="session")
def model():
    from pycalc.model import evaluateExpression

    yield evaluateExpression

@pytest.fixture(scope="session")
def controller(model, view):
    from pycalc.controller import PyCalcCtrl

    yield PyCalcCtrl(model, view)
```



```
def test_returnSignal(controller):
    """Tests the Return key binding interface to our Qt display widget."""
    from PyQt5 import QtCore, QtGui

    controller._view.setDisplayText("1+2")
    event = QtGui.QKeyEvent(
        QtCore.QEvent.KeyPress, QtCore.Qt.Key_Enter, QtCore.Qt.NoModifier
    )
    controller._view.display.keyPressEvent(event)
    assert controller._view.displayText() == "3"
```

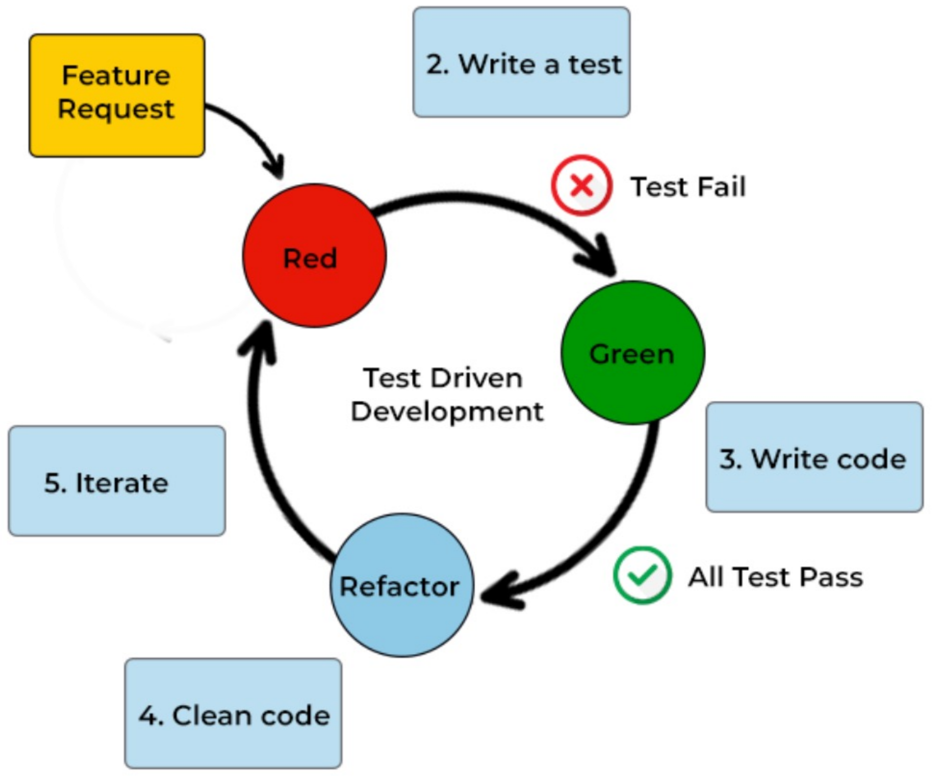


```
(base) C:\Users\Dean-Lab\Documents\GitHub>pytest
================================================= test session starts =================================================
platform win32 -- Python 3.10.8, pytest-7.4.2, pluggy-1.0.0
rootdir: C:\Users\Dean-Lab\Documents\GitHub
plugins: anyio-3.6.2, cov-4.1.0
collected 1 item

CI2023-MVC-calculator-answerkey\test\test_controller.py .                                                        [100%]

================================================== 1 passed in 0.62s ==================================================

(base) C:\Users\Dean-Lab\Documents\GitHub>
```

Unit Testing Best Practices

- Unit test cases should be independent. In case of any enhancements or change in requirements, unit test cases should not be affected.
- Test only one code at a time.
- Follow clear and consistent naming conventions for your unit tests
- In case of a change in code in any module, ensure there is a corresponding unit test case for the module, and the module passes the tests before changing the implementation
- Bugs identified during unit testing must be fixed before proceeding to the next phase of development
- Adopt a “test as your code” approach. The more code you write without testing, the more paths you have to check for errors.

Testing Driven Development

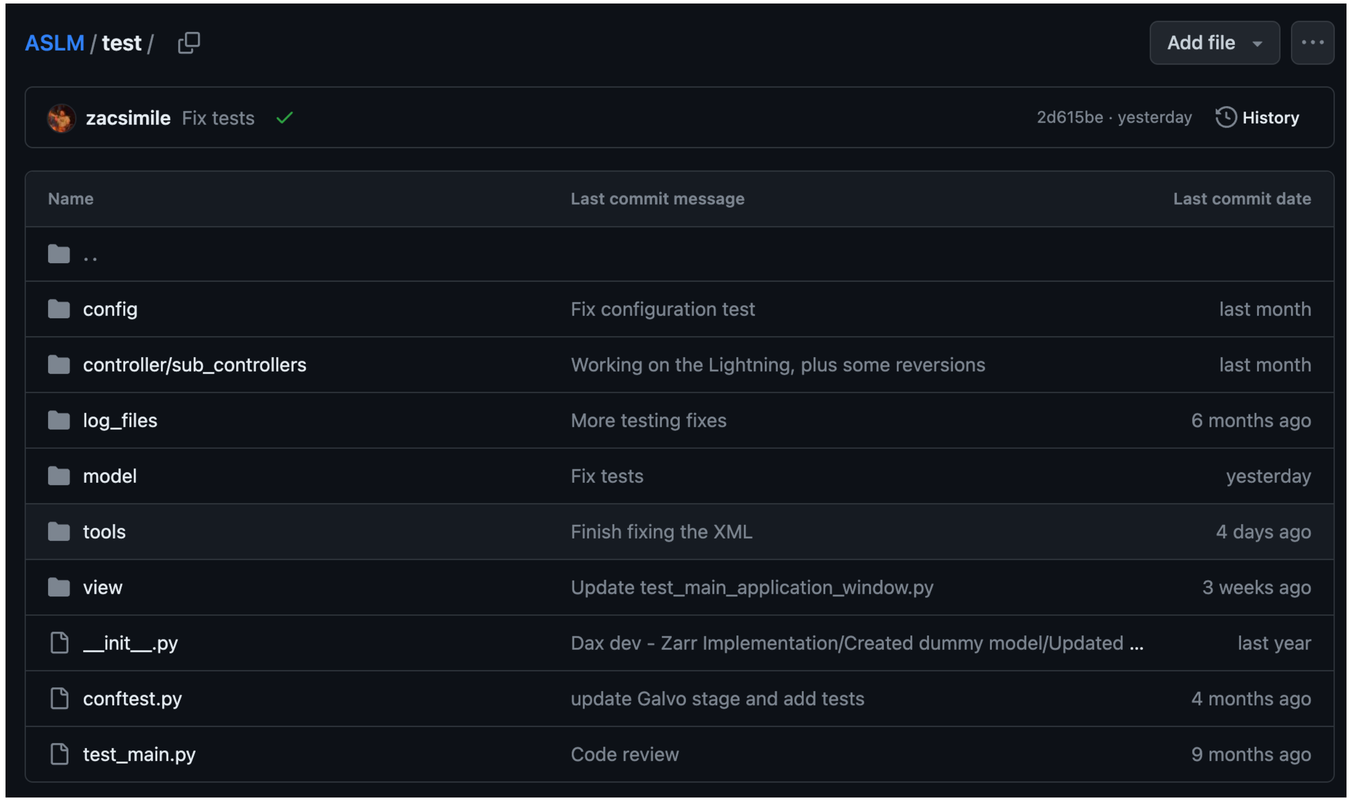


When to write unit tests

- Ideally, best time to write unit test is before you have written your function or when you have identified and solved a bug in your code.
- Adopt a "test as your code" approach. The more code you write without testing, the more paths you have to check for errors.
- Organize code during testing, remove hardcoded test data to insure tests cover all use conditions.
- Only proceed to integration when all tests pass and existing functionality isn't broken when new feature is added.

Examples of Unit Tests and Test Driven Development in ASLM code

- Device Control within Microscope
  - Functionality, I/O
- Integration of new features that doesn't break control of other microscopes



Name	Last commit message	Last commit date
..		
config	Fix configuration test	last month
controller/sub_controllers	Working on the Lightning, plus some reversions	last month
log_files	More testing fixes	6 months ago
model	Fix tests	yesterday
tools	Finish fixing the XML	4 days ago
view	Update test_main_application_window.py	3 weeks ago
__init__.py	Dax dev - Zarr Implementation/Created dummy model/Updated ...	last year
conftest.py	update Galvo stage and add tests	4 months ago
test_main.py	Code review	9 months ago

Conclusions

- Writing tests are useful for making sure code is functioning properly and removing bugs during development
  - Easier to identify and fix new bugs
  - Provides documentation of features
- Pytest is a useful framework for setting up tests

Future Topics

- Automated Unit Testing with Continuous Integration and Github Actions
- Topics not covered
    - Mock Objects

Further Reading about unit tests

- Pytest documentation: https://pytest.org/en/7.4.x/contents.html
- Creating Mock Objects for Unit Tests: https://changhsinlee.com/pytest-mock/
- Unit testing for deep learning libraries: https://pypi.org/project/pytest-pytorch/, https://www.tensorflow.org/api_docs/python/tf/test/TestCase, https://theaisummer.com/unit-test-deep-learning/
- Unit testing for image analysis:
- Unit testing for sequencing data analysis:

Exercise

- Install pytest in environment using pyproject.toml
- Write a unit test to test _calculateResult function in controller.py
- If time is remaining, parameterize the test for _calculateResult