Unit Testing and Test Driven Development

Conor McFadden

Continuous Integration with Python Nanocourse 2023

Unit Testing overview

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

Software Review Overview

- **Verification:** Formal proof that a program is correct.

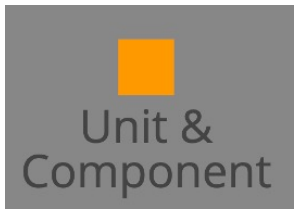
  Tedious to do by hand, and automated tool support for verification is still an active area of research.

- **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.

- **Testing:** Running the program on carefully selected inputs and checking the results.

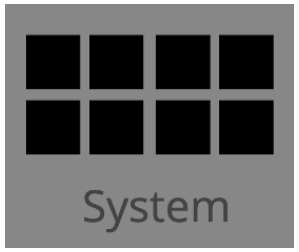
  User Inputs? Logic errors? Hardware bugs?

Types of Tests



Unit Test: Isolated method, function or component.

Integration Test: Combined Unit Tests tested as a group.



System (Smoke) Testing: Tests the entire system as a whole to verify that critical functionalities work.

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.

pytest: A popular and powerful third-party testing framework. It encourages test-driven development (TDD) and provides features like fixtures and parameterized testing.

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.

Writing a 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.

Running pytest

```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
```

Just type >> pytest in prompt

Assert is True, the test passes!

```
============================================================ 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 %
```

Running pytest

```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
```

Assert is False, the test fails!

```
=================================================== 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 %
```

Any “test_” function will be recognized by pytest automatically

```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 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 %
```

Parameterization in tests

- Test multiple parameters at once using *pytest* 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 %
```

Fixtures in pytest

A way to set up and clean up things before and after running tests.

They help avoid repeating setup code in multiple tests.

Why Use Fixtures?

- Avoid duplicating setup code in every test.
- Keep test files clean and readable.
- Automatically clean up after tests run.

Fixtures in pytest

Example: Using a `pytest` Fixture

Let's say we have a simple `Calculator` class:

python

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

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

Without a Fixture (Repeated Setup Code)

Each test creates a `Calculator` instance separately:

python

```python
import pytest
from calculator import Calculator

def test_add():
    calc = Calculator()  # Repeating setup code
    assert calc.add(2, 3) == 5

def test_subtract():
    calc = Calculator()  # Repeating setup code
    assert calc.subtract(5, 3) == 2
```

Fixtures in pytest

Example: Using a `pytest` Fixture

Let's say we have a simple `Calculator` class:

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

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

Using a `pytest` Fixture

Instead of creating a `Calculator` instance in each test, we define a **fixture**:

```python
import pytest
from calculator import Calculator

@pytest.fixture
def calculator():
    """Fixture that provides a Calculator instance"""
    return Calculator()

def test_add(calculator):
    assert calculator.add(2, 3) == 5

def test_subtract(calculator):
    assert calculator.subtract(5, 3) == 2
```

1. The `@pytest.fixture` decorator marks `calculator()` as a fixture.
2. Each test that needs a `Calculator` **automatically receives it** as an argument.
3. **No need to manually create** a `Calculator` object in every test.
4. `pytest` **handles the fixture**—it runs before each test and provides the setup object.

Key Binding Test in PyCalc Exercise

conftest.py

```python
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

    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)
```

test_controller.py

```python
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"
```

Fixtures → conftest.py

Yield keyword to free resources after test.

What is yield doing?

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

    calc = QApplication([])

    yield calc

    calc.exit()
```

SETUP: prepare resources

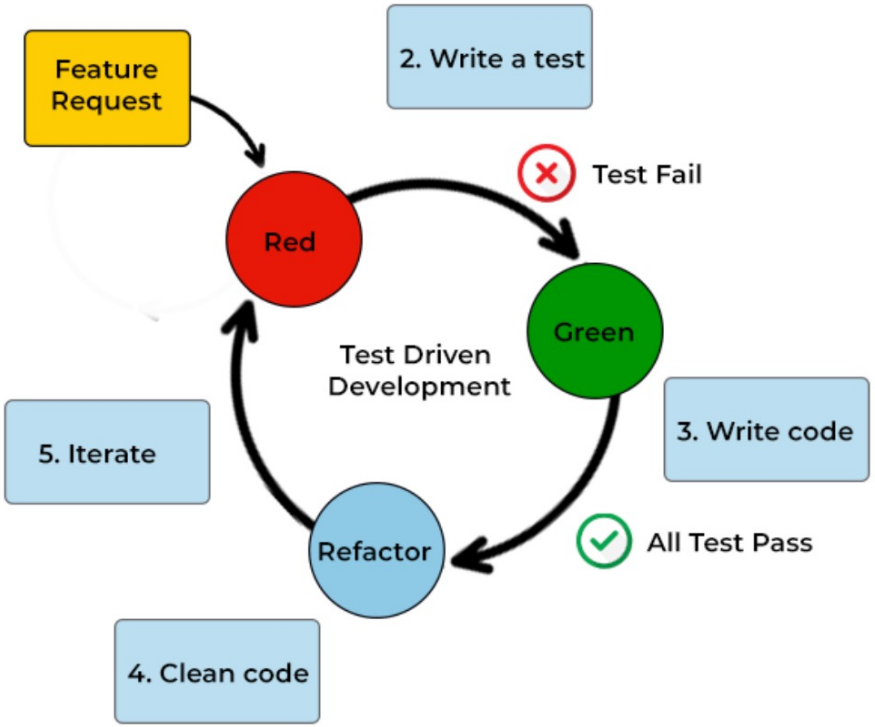
YIELD: give it to your tests

TEARDOWN: free your resources

Unit Testing Best Practices

- Unit test cases should be independent.
- Test only one code or component at a time.
- Clear and consistent naming conventions.
- Fix bugs before moving on!
- “Test as you code” → Write you tests while the idea is still fresh.

Testing-Driven Development/Deployment



Conclusions

- Writing tests are useful for making sure code is functioning properly and removing bugs during development

- Pytest is a useful framework for setting up tests

Exercise

- Install pytest in environment using pyproject.toml

  pip install –e .[dev] or *pip install –e ".[dev]" on Mac*

- Write a unit test to test _calculateResult function in controller.py

- If time is remaining, parameterize the test for _calculateResult