Unit Testing with `pytest`

Why unit testing?

- Test the building blocks of your software
- Earlier detection of errors
- Test oriented development: First write a failed test, than implement function to pass it, iteratively
- It is a good practice to include in as a build step in development/integration of software.

Python has the build-in **unittest** module for writing unit testing. **pytest** is compatible and more powerful framework.

How it works?

`pytest` is a module and a command line tool executing tests. When started, it executes a *test discovery* procedure, searching for all descendant directories to find Python files start with `test_` as `test_abc.py`, `src/test_main.py`, `library/networking/test_connection.py`. After module discovery, each file is searched for functions starting with `test` or classes starting with `Test`. Each function and methods of the class starting with `Test` are executed in a special environment that also captures the standard output, error, and exceptions.

Whenever functions execute `assert`*bool* , it is considered as a test case. The expression determines the result of the case.

```
def test_alwayssuccess():
    assert True

def test_alwaysfail():
    assert False
```

Results in:

```
shell prompt$ pytest
========================================== test session starts ============================
collected 2 items

test_1.py .F

================================================ FAILURES =================================
_____________________________________ test_alwaysfail ____________________________

    def test_alwaysfail():
>       assert False
E       assert False
```

```
test_1.py:5: AssertionError
========================================= short test summary info ==========================
FAILED test_1.py::test_alwaysfail - assert False
=
```

If second function is modified to assert `True`, it will be:

```
shell prompt$ pytest
========================================= test session starts ==============================
collected 2 items

test_1.py ..

========================================= 2 passed in 0.09s ================================
```

The test functions usually reflect the function specification as:

```
def area(shape,r=0,w=0,h=0):
    if shape == 'circle':
        return 3.14*r*r
    elif shape == 'rectangle':
        return w*h
    else:
        return None

def test_circle():
    assert area('circle', r=2) == 12.56

def test_square():
    assert area('rectangle', w=10,h=10) == 100

def test_noshape():
    assert area('nosuchshape') == None
```

Class Methods

Implementing test functions under a class helps in organization of tests, as namespaces. For example if you have classes `A, B, C` in your software, you can implement `TestA, TestB,` and `TestC` to group tests of each class.

```
class Shape():
    def __init__(self,x,y):
        self.x, self.y = x, y
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, x, y, r):
```

```
        self.r = r
        super().__init__(x,y)
    def area(self):
        return 3.14*self.r**2

class Rectangle(Shape):
    def __init__(self, x, y, w, h = None):
        super().__init__(x,y)
        self.w = w
        self.h = h if h else self.w
    def area(self):
        return self.w*self.h
```

The classes above can be tested as:

```
import shape as s

class TestShape:
    def test_noshape(self):
        assert s.Shape(3, 3).area() == None

class TestCircle:
    def test_circle(self):
        assert s.Circle(10,20,2).area() == 12.56

class TestRectangle:
    def test_square(self):
        assert s.Rectangle(5, 5, 10).area() == 100

    def test_rectangle(self):
        assert s.Rectangle(10, 10, 5, 8).area() == 40
```

`pytest` will execute all of them as usual. On the command line you can select each class individually with `-k` option. The following will execute only two tests under class `TestRectangle`.

```
shell prompt$ pytest -k TestRectangle
========================================= test session starts ==============================
platform linux -- Python 3.13.5, pytest-8.3.5, pluggy-1.5.0
rootdir: /home/onur/Documents/445/samples/unit_test
plugins: typeguard-4.4.2, hypothesis-6.130.5
collected 13 items / 11 deselected / 2 selected

test_4.py ..

=================================== 2 passed, 11 deselected in 0.08s =====================
```

Testing Exceptions

You can use `with pytest.raises('*exceptionclass*)context to test if functions properly raises the expected exceptions (you need to importpytest). For example, if you do not implementarea()method ofShape‘ class above, you can test it as:

```
class TestShape:
    def test_noshape(self):
        with pytest.raises(AttributeError):
            s.Shape(3, 3).area()
```

This example will fail the test if the exception is not raised inside the context.

Marking and naming exceptions

`pytest.mark` decorator associates the tests with tags so that selective execution will be possible.

```
@pytest.mark.web
def test_connection():
    # ...

@pytest.mark.web
def test_download():
    # ...

@pytest.mark.web
def test_post():
    #. ....

@pytest.mark.db
def test_dbconn():
    # ...

@pytest.mark.db
def test_select():
    # ...

@pytest.mark.db
def test_update():
    # ....

@pytest.mark.game
class TestArena:
    # ...
```

```
@pytest.mark.game
class Test.Player:
    # ...
```

This example can be invoked as `pytest -m web`, `pytest -m db`, and `pytest -m game` to execute tests seperately depending on the semantics.

Parametrizing the tests

When the same function calls are tested against a group of data, the test inputs and expected results can be put in a list and test function is called with each data separately by using `pytest.mark.parametrize` decorator.

```
import shape as s
import pytest

@pytest.mark.parametrize("radius,area",[
    (0,0), (1,3.14), (10,314), (5, 78.5)])
def test_circle(radius, area):
    assert s.Circle(0,0,radius).area() == area
```

The first parameter to the decorator is the list of parameter names to the test function. The above function tests for 4 cases.

Fixtures

The tests often requires construction of values by executing a standard group of commands in advance. Test functions working on the same values repeat these actions.

In order to abstract this value construction, we can put the commands in a function returning the value and call it as we need. `pytest` provides `fixture` decorator to write such functions that are substituted as parameters of functions. Fixtures have a better control of the scope of the values, per test function, per class, per test module or per session.

```
@pytest.fixture
def sample_tree():
    bt = bst.BSTree()
    keys = ("z","g","b","a","y","c")
    for i, k in enumerate(keys):
        bt[k] = i
    return bt

def test_inorder(sample_tree):
    vals =  tuple(v for (k,v) in sample_tree.inorder())
    assert vals == (3, 2, 5, 1, 4, 0)
```

```
def test_preorder(sample_tree):
    vals = tuple(v for (k,v) in sample_tree.preorder())
    assert vals == (0, 1, 2, 3, 5, 4)
```

Assume the module `bst` has a binary search tree example and you need a large (even larger than the example above) tree for your tests. `sample_tree` function returns a fixture and each time tests reference `sample_tree` as a value, the fixture is called and the value is used. `pytest --setup-show` command shows how fixture is invoked as the tests are executed:

```
test_6.py
        SETUP    F sample_tree
        test_6.py::test_inorder (fixtures used: sample_tree).
        TEARDOWN F sample_tree
        SETUP    F sample_tree
        test_6.py::test_preorder (fixtures used: sample_tree).
        TEARDOWN F sample_tree
```

If the construction of the fixture is costly, it may be a better idea to reuse the values accross the test functions. `scope='module'` parameter passed to the fixture will reuse the value for all functions in the current test (python file). Also fixtures like network connections, database connections, authenticated connections had better have longer scopes.

```
@pytest.fixture(scope='module')
def sample_tree():
    bt = bst.BSTree()
    keys = ("z","g","b","a","y","c")
    for i, k in enumerate(keys):
        bt[k] = i
    return bt
...
```

The test output with `--setup-show` will be:

```
test_6.py
    SETUP    M sample_tree
        test_6.py::test_inorder (fixtures used: sample_tree).
        test_6.py::test_preorder (fixtures used: sample_tree).
    TEARDOWN M sample_tree
```

Capture the Output

Sometimes you need to test also what functions `print()` instead of or along with what they return. `pytest` captures all standard output and error by default and show them when one of the tests fail. If all succeeds, the output is hidden.

`capsys`, `capfd`, `capsysbinary` and `capfdbinary` built-in fixture make output available in test functions. `readouterr()` method of the fixture returns a named

tuple, `out` and `err` fields give current state of standard output and standard error captures respectively.

```
def drawrect(n):
    for i in range(1,n+1):
        for j in range(i):
            print('#', end='')
        print()


def test_draw(capsys):
    drawrect(3)
    captured = capsys.readouterr()
    assert captured.out == '#\n##\n###\n'

def test_draw4(capsys):
    drawrect(4)
    captured = capsys.readouterr()
    assert captured.out == '#\n##\n###\n####\n'
```