Pytest with Unittest

In this tutorial will learn how to use pytest to create a unit test

Reference:

Youtube channel-pixegami:

Pixhttps://www.youtube.com/watch?v=YbpKMIUjvK8&list=PLZJBfja3V3RvxooZ5SNOr7CMFzURr4NBs&index=8&t=1274s

Unit Test Sample

You can use this two code to run the test, it will pass, because no code is implement yet. Throughout the tutorial will update the code

We will need two file, one will contain the functionality of the code, another file will be the unit test or the testcase which will run the testcase. You just image you are testing that code, and one for writing a testcase to test the code.

shoppingcarty.py

```
from typing import List
class ShoppingCart:
    def __init__(self)-> None:
        pass
    def add(self, item: str):
       pass
    def size(self)-> int:
        pass
    def get_items(self)->List[str]:
        pass
    def get_total_price(self, price_map):
        pass
```

test_shoppingcart.py

```
from shoppingcart import ShoppingCart
import pytest

def test_can_add_item_to_cart():
    pass
```

Unit Test Case1: create apple and check len size

Make unit test fail

Let learn how to compare value using asset

shoppingcart.py

```
class ShoppingCart:
....
    def size(self)-> int:
        return 0
```

test_shoppingcart.py

```
def test_can_add_item_to_cart():
    #adding to chart
    cart=ShoppingCart()
    cart.add('apple')
    assert cart.size() == 1
```

Now let make this test fail, by adding an assert keyword, it will compare the code and expected value. In this case our size will return 0 which will be current value, and expected value is `assert cart.size() == 1` so indeed 0 is not equal 1 it will fail

Implement adding item to cart

You can use either self.items = [] or self.items:List[str]=[]

shoppingcart.py

```
class ShoppingCart:
    def __init__(self)-> None:
        #self.items = []
        self.items:List[str]=[]

    def add(self, item: str):
       self.items.append(item)

    def size(self)-> int:
        return len(self.items)
```

The test_shoppingcart.py is the same I didn't modify anything

```
def test_can_add_item_to_cart():
    #adding to chart
    cart=ShoppingCart()
```

```
    cart.add('apple')
    assert cart.size() == 1
```

Now run the test again it will pass

```
test_shoppingcart.py::test_can_add_item_to_cart PASSED

========================================= 1 passed in 0.11s ======
PS C:\pytest_apitest>
```

Unit Test Case2: check item in cart

Shoppingcart.py

```
class ShoppingCart:
    def __init__(self)-> None:
        #self.items = []
        self.items:List[str]=[]

    def add(self, item: str):
       self.items.append(item)

    def size(self)-> int:
        return len(self.items)

    def get_items(self)->List[str]:
        return self.items
```

test_shoppingcart.py

```
def test_when_item_added_then_cart_contains_items():
    cart=ShoppingCart(5)
    cart.add('apple')
    assert "apple" in cart.get_items()
```

Unit Test Case3: Raise Exception max value

Will not raise exception

- Don’t raise error, when adding max_size-1

shoppingcart.py

```
class ShoppingCart:
    def __init__(self, max_size: int)-> None:
```

```
        #self.items = []
        self.items:List[str]=[]
        self.max_size=max_size

    def add(self, item: str):
        #check size with max_size, if same then raise error
        if self.size()== self.max_size- 1:
            raise OverflowError("Cannot add more items")
        #else continue adding
        self.items.append(item)
```

test_shoppingcart.py

```
def test_when_add_morethan_max_should_fail():
    cart=ShoppingCart(5)
    #if throw this error mean pass
     #if you run this will not catch the bug if self.max_size-1
    with pytest.raises(OverflowError):
        for i in range(6):
            cart.add('apple')
```

In the above code you will not catch the bug, if you add self.max_size-1, it will still be pass. It will run add 4 item, but our max is 6. The reason is because for loop write under with.

When you use the condition `self.size() == self.max_size - 1`, you're essentially checking if the cart is *almost full* before adding an item.

- **Cart is empty:** `size = 0`, `max_size = 5`. Condition is `False`, so the item is added.
- **Cart has 4 items:** `size = 4`, `max_size = 5`. Condition is `True`, so the item is *not* added (which is correct).
- **Cart has 5 items:** `size = 5`, `max_size = 5`. Condition is `False`, so the item is *added* even though the cart is full!

The problem: The check happens *before* the item is added, so it doesn't accurately reflect the cart's state *after* the potential addition.

The code will never actually reach the 6th loop iteration.

Here's a breakdown of what happens:

1. The `for` loop iterates 5 times, adding an item to the cart in each iteration.
2. On the 5th iteration, the cart becomes full.
3. The loop ends because it has reached its defined range (0 to 4).
4. The code moves to the line outside the loop: `with pytest.raises(OverflowError):`
5. This line attempts to add another item to the cart, which is now full.
6. An `OverflowError` is raised, indicating that the cart cannot hold any more

1. **When `Size=0` and max_value= `5-1` condition false so add item to list:** This is correct. The condition `size == max_value - 1` is false, so the item is added.
2. **When `size=1` and `max_value=5-1` conditions false so add item to list:** This is also correct. The condition is still false.
3. **When `size=4` and `max_value=5-1` condition true so will not add to item:** This is correct. The condition is true, so the item is not added.
4. **when size=5 and max _value=5-1 condition false will not add, but because check happen before add, so will not trigger error:** This is where the issue lies.

The problem is that the check happens *before* the item is added. So, w**hen size is 5, the condition size == max_value - 1 is false**. The cart is already full after the 4th item, so we need to add one more to cause an overflow. Please refer below solution to fix this problem

Solution to trigger exception

So to fix this we can change the code like this:

I have cross the code and change to new code, to solve the issue

test_shoppingcart.py

def test_when_add_morethan_max_should_fail():
cart=ShoppingCart(5)
# if throw this error mean pass 	
# if you run this will not catch the bug if self.max_size-1 	
~~with pytest.raises(OverflowError):~~
~~for i in range(6):~~
~~cart.add('apple')~~
# #run 5 times 	
for i in range(5):
cart.add('apple')
with pytest.raises(OverflowError):
cart.add('apple')

From above you can see iteration 5 loop, and then try to add one more item will run exception. So when you run this test it will fail the test:

```
        if self.size()== self.max_size -1:
>           raise OverflowError("Cannot add more items")
E           OverflowError: Cannot add more items

shoppingchart.py:12: OverflowError
========================================= short test summary info =================
FAILED test_shoppingchart.py::test_when_add_morethan_max_should_fail - OverflowError
==================================== 1 failed, 3 passed in 0.42s ==============
PS C:\pytest_apitest>
```

I will remove the if self.size()== self.max_size -1 code in shoppingcart.py. I just want to show you if you write add self.size()== self.max_size- 1will have a bug, it will not trigger the execption and how to fix it

Unit Test **4 total the item**

Adding total price

shoppingcart.py

```
class ShoppingCart:

    def get_total_price(self, price_map):
        total_price=0
        for item in self.items:
            total_price+=price_map.get(item)
# you can also use index method price_map[item]
        return total_price
```

You can use price_map.get(item) or price_map[item], since it's a dictionary you can use the get method to get the key value.

test_shoppingcart.py

```
def test_togetthe_price():
    cart= ShoppingCart(5)
    cart.add('apple')
    cart.add('banana')

    price_map={ 'apple': 4.0,'banana': 1.0 }
    assert cart.get_total_price(price_map) == 5.0
```

If you don't know where is the price_map this variable you can refer below picture. Basely price_map is define in the unittest file.

```
def get_total_price(self, price_map):
    total_price=0
    for item in self.items:
        total_price+=price_map.get(item) # you can also use index method price_map[item]
    return total_price
```

Receive price_map

```
def test_togetthe_price(cart):
    #cart= ShoppingCart(5)
    cart.add('apple')
    cart.add('banana')

    price_map={ 'apple': 4.0,'banana': 1.0 }
    assert cart.get_total_price(price_map) == 5.0
```

Pass price_map

Clean Code: Duplicate code using fixture

Now you can see above having many duplicate code like cart= ShoppingCart(5), instead of writing so many related code we can add fixture to solve duplicate code.

```
from shoppingchart import ShoppingCart
import pytest
def test_can_add_item_to_cart():

    #adding to chart
    cart=ShoppingCart(5)
    cart.add('apple')
    assert cart.size() == 1

def test_when_item_added_then_cart_contains_items():
    cart=ShoppingCart(5)
    cart.add('apple')
    assert "apple" in cart.get_items()

def test_when_add_morethan_max_should_fail():
    cart=ShoppingCart(5)
    # if throw this error mean pass
```

Duplicate code

We will create pytest.fixture and add a function below it, and all the test will pass in argument cart. In below I mark red color need to change. From original cart=ShoppingCart(5) now you can remove it, and pass in cart argument which is a fixture.

```
@pytest.fixture
def cart():
```

```
    return ShoppingCart(5)

def test_can_add_item_to_cart(cart):
.....
def test_when_item_added_then_cart_contains_items(cart):
    cart.add('apple')
    assert "apple" in cart.get_items()
....
```

```python
@pytest.fixture
def cart():
    return ShoppingCart(5)

def test_can_add_item_to_cart(cart):

    #adding to chart
    #cart=ShoppingCart(5)
    cart.add('apple')
    assert cart.size() == 1

def test_when_item_added_then_cart_contains_items(cart):
    # cart=ShoppingCart(5)
    cart.add('apple')
    assert "apple" in cart.get_items(cart)
```

Code please refer the full code

Mock dependency

https://docs.python.org/3/library/unittest.mock.html

A mock object is a simulated object used in testing to isolate the component you're testing.

Create Fake Database

In the previous example we use the price_map, and we know that there is a get method. Let assume if our get method is not implement yet, then what should I do.

```python
def get_total_price(self, price_map):
    total_price=0
    for item in self.items:
        total_price+=price_map.get(item) # you can also use index method price_map[item]
    return total_price
```

```python
def test_togetthe_price(cart):
    #cart= ShoppingCart(5)
    cart.add('apple')
    cart.add('banana')

    price_map={ 'apple': 4.0,'banana': 1.0 }
    assert cart.get_total_price(price_map) == 5.0
```

Let create a fake database and think that the get function will implement by other develop, and it's not done yet. In this situation we can use a mock. Let create a fake database

Item_db.py

```python
class ItemDatabase:
    def __init__(self)>None:
        pass
    def get(self, item: str)->float:
        pass
```

test_shoppingcart.py

```python
from shoppingcart import ShoppingCart
from item_db import ItemDatabase
from unittest.mock import Mock
import pytest
....
def test_togetthe_price(cart):
    cart.add('apple')
    cart.add('banana')
    cart.add('banana')
```

```
    item_database=ItemDatabase()
    assert cart.get_total_price(item_database) == 3.0
```

When running this will fail

```
>           total_price+=price_map.get(item) # you can also use index method price_map
E           TypeError: unsupported operand type(s) for +=: 'int' and 'NoneType'

shoppingchart.py:25: TypeError
```

Instead of waiting for the get method to implement in order to relied on item_database or ItemDatabase(), we can use mock behavior of item_database.

Mock unittest

To use the mock you need to import mock library from unittest.mock import Mock

```
def test_togetthe_price(cart):
    #cart= ShoppingCart(5)
    cart.add('apple')
    cart.add('banana')
    #cart.add('banana')

    item_database=ItemDatabase()
    #mock
    item_database.get = Mock(return_value=1.0)
    assert cart.get_total_price(item_database) == 3.0
```

I know that the get method exist in item_db.py but it's not implement yet, still developer. But I need to test this, so in this case I will mock the item_database. So I will return a 1.0 value, this mean it will pass 1.0 to get

```python
def test_togetthe_price(cart):
    #cart= ShoppingCart(5)
    cart.add('apple')
    cart.add('banana')
    #cart.add('banana')

    #item_database.get=Mock(return_value=1.0)
    #price_map={ 'apple': 4.0,'banana': 1.0 }
    #assert cart.get_total_price(price_map) == 3.0

    item_database=ItemDatabase()
    #mock
    item_database.get = Mock(return_value=1.0)
    assert cart.get_total_price(item_database) == 3.0
```

test_shoppingchart.py

Will pass 1.0

```python
def get_total_price(self, price_map):
    total_price=0
    for item in self.items:
        total_price+=price_map.get(item) # y
    return total_price
```

shoppingchart.py

Will receive 1.0

- **Assert Fail**

It will fail on asset assert 2.0 == 3.0, so because we only add two item, so add another item will pass. In above remove the comment of yellow mark and run again will pass.

So when it's add apple will pass 1.0, banana pass1.0, and banana pass 1.0, total up to 3.0.

- **Problem:**
  - **Two item and get up to 3.0**
  - **Each item should have different price**

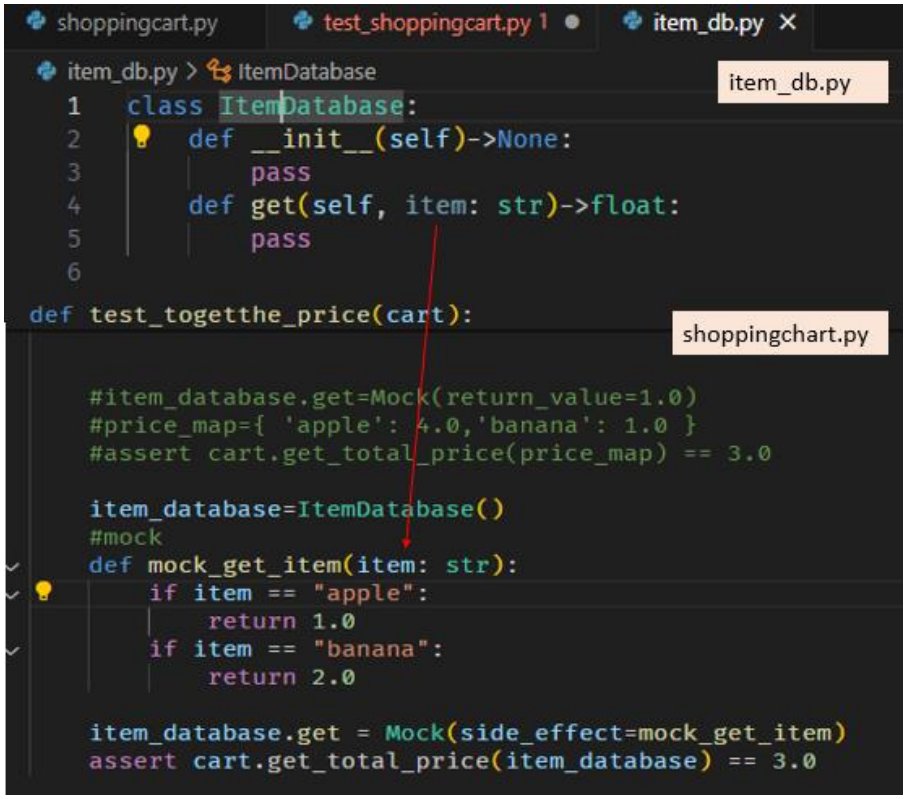
Customize mockup behavior

We need to use side_effect argument that mock provide. Below is an example. You need to create a side_effect function first.

```python
>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]
...
>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
>>> mock(), mock(), mock()
(5, 4, 3)
```

```
def test_togetthe_price_mock(cart):
    #cart= ShoppingCart(5)
    cart.add('apple')
    cart.add('banana')
    item_database=ItemDatabase()
    #mock
    def mock_get_item(item: str):
        if item == "apple":
            return 1.0
        if item == "banana":
            return 2.0

    item_database.get = Mock(side_effect=mock_get_item)
    assert cart.get_total_price(item_database) == 3.0
```



Now let run will pass

```
plugins: xdist-3.6.1
collected 4 items

test_shoppingcart.py::test_can_add_item_to_cart PASSED
test_shoppingcart.py::test_when_item_added_then_cart_contains_items PASSED
test_shoppingcart.py::test_when_add_morethan_max_should_fail PASSED
test_shoppingcart.py::test_togetthe_price PASSED

========================================== 4 passed in 0.18s ===================
PS C:\pytest_apitest>
```