Python Unittest Mock

Summary: in this tutorial, you'll learn about the Python unittest (https://www.pythontutorial.net/python-unit-testing/python-unittest/) Mock class and how to use it to mock other classes.

Introduction to Python unittest Mock class

Mocks simulate the behaviors of real objects. To test an object that depends on other objects in an isolated manner, you use mock objects to mock the real objects.

To mock objects, you use the `unittest.mock` module. The `unittest.mock` module provides the `Mock` class that allows you to mock other objects.

It also provides the `MagicMock` class that is a subclass of the `Mock` class. Besides the methods and properties of the `Mock` class, the `MagicMock` class has the implementations of all the dunder methods e.g., __str__ (https://www.pythontutorial.net/python-oop/python-__str__/) and __repr__ (https://www.pythontutorial.net/python-oop/python-__repr__/) .

See the following example:

```
from unittest.mock import Mock
```

```
# create a new mock object
mock = Mock()

# mock the api function
mock.api.return_value = {
    'id': 1,
    'message': 'hello'
}
# call the api function
print(mock.api())
```

Output:

```
{'id': 1, 'message': 'hello'}
```

How it works.

First, import the `Mock` class from the `unittest.mock` module:

```
from unittest.mock import Mock
```

Second, create a new instance of the `Mock` class:

```
mock = Mock()
```

Third, mock the `api()` function and assign its return value to a dictionary:

```
mock.api.return_value = {
    'id': 1,
    'message': 'hello'
}
```

Finally, call the `api()` from the mock object. It'll return the assigned value:

```
print(mock.api())
```

In this example, we have two mock objects: `mock` & `mock.api` .

Let's add the `print()` statement to the program to see how it works:

```
from unittest.mock import Mock


# create a new mock object
mock = Mock()
print(mock)

# mock the api function
mock.api.return_value = {
    'id': 1,
    'message': 'hello'
}
print(mock.api)

# call the api
print(mock.api())
```

Output:

```
<Mock id='1830094470496'>
<Mock name='mock.api' id='1830100086416'>
{'id': 1, 'message': 'hello'}
```

The output shows two `Mock` objects.

In short, if you assign a property that doesn't exist on the `Mock` object, Python will return a new mock object. Because of this dynamic, you can use the `Mock` class to mock any objects that you want.

When to use mock

These are cases that you may consider using mocks:

- System calls
- Networking
- I/O operation
- Clocks & time, timezones
- Or other cases whose results are unpredictable

Why using mocks

The following are benefits of mocks:

- Speed up the test
- Exclude external redundancies
- Make unpredictable results predictable

Python Unittest Mock example

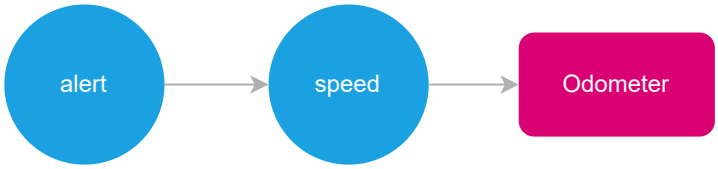
Suppose you have a module called `odometer.py`:

```
from random import randint


def speed():
    return randint(40, 120)


def alert():
    s = speed()
    if s < 60 or s > 100:
        return True
    return False
```

In the sensor.py module:

- The `speed()` returns the current speed of a vehicle. It returns a random value between 40 and 120. In the real world, the function would read the data from the odometer.

- The `alert()` function returns true if the current speed is lower than 60 km/ and higher than 120 km/h. The `alert()` function uses the `speed()` function to get the current speed.



It'll be difficult to test the `alert()` function because the value returned by the `speed()` function is varied. To resolve it, you can use `Mock` class.

The following creates a `test_odometer.py` test module that tests the `alert()` function:

```
test_alert_normal (test_odometer.TestOdometer) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

How it works.

First, assign a Mock object to the `odometer.speed` function:

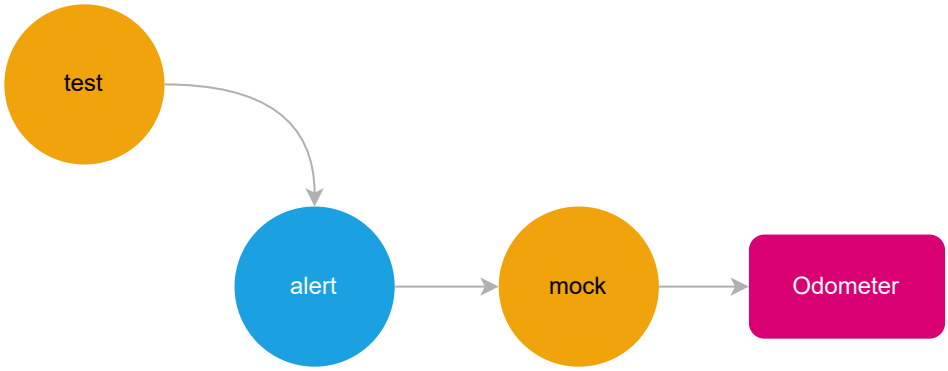
```
odometer.speed = Mock()
```

Second, set the return value of the `speed()` function to 70:

```
odometer.speed.return_value = 70
```

Third, call the `alert()` function and test if it returns False. The `alert()` function will call the mock object instead of the actual `speed()` function.

The following picture illustrates how the test works with mock objects:



Run the test:

```
python -m unittest test_odometer.py -v
```

Output:

```
test_alert_normal (test_odometer.TestOdometer) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

The following adds the test cases that are over and under speed:

```
import unittest
from unittest.mock import Mock
import odometer


class TestOdometer(unittest.TestCase):
    def test_alert_normal(self):
        odometer.speed = Mock()
        odometer.speed.return_value = 70
        self.assertFalse(odometer.alert())

    def test_alert_overspeed(self):
        odometer.speed = Mock()
```

```
        odometer.speed.return_value = 100
        self.assertFalse(odometer.alert())

    def test_alert_underspeed(self):
        odometer.speed = Mock()
        odometer.speed.return_value = 59
        self.assertTrue(odometer.alert())
```

Run the test:

```
python -m unittest test_odometer.py -v
```

Output:

```
test_alert_normal (test_odometer.TestOdometer) ... ok
test_alert_overspeed (test_odometer.TestOdometer) ... ok
test_alert_underspeed (test_odometer.TestOdometer) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Summary

- Use the `Mock` class of the `unittest.mock` class to mock other objects.