Matt Pease Follow
Software engineer and architect. Interested in Python, designing software solutions and Lindy Hop.
Nov 3, 2016 · 9 min read

Python Mocking, You Are A Tricksy Beast

Python mocking there is something unintuitive about you. You are not easy to get to grips with and it takes several "Ah, now I get it!" moments before you are better understood. You are a powerful, fantastic tool for enabling automated unit tests, but I know many developers who "get by" on the knowledge they have and writing those tests takes longer than it should.

There are many articles about mocking, and yes, I am now writing another one. However, I really want to have a go at clearly explaining it, especially those "Aha!" moments that pushed forward my understanding.

Firstly, what is mocking? Here is a quote from the docs:

> *mock is a library for testing in Python. It allows you to replace parts of your system under test with mock objects and make assertions about how they have been used.*

So *"it allows you to replace parts of your systems"*, but what parts? It turns out you can mock pretty much anything you can define, such as:

- functions
- classes
- objects

You replace them with *"mock objects"* which are instances of the Mock or MagicMock class.

You then *"make assertions"* about that Mock instance, which is a way of checking that the Mock instance was used in the way you were expecting.

In this article I am going to cover the basics of replacing parts of your systems but I will not be covering making assertions.

Mocking Functions

Let's start with a simple example. I am going to create a new module called **simple.py** and in that module I am going to define a function called **simple_function** that just returns a string like so:

```
def simple_function():
    return "You have called simple_function"
```

Now I am going to create a second module called **use_simple.py**. In that module I am going to import the mock package from unittest and import the simple module:

```
from unittest import mock

import simple
```

Next I am going to a write two functions, one that calls **simple_function** normally and one that mocks the call to it. Note that normally you would do this in a testing context but I am purposefully stripping that away so I can focus on the mocking part. The first function:

```
def use_simple_function():
    result = simple.simple_function()
    print(result)


use_simple_function()
```

As you can see it just prints out the result of **simple_function**. Specifically, the output is:

```
You have called simple_function
```

For the second function, to mock **simple_function** I am going to use the mock.patch decorator. This decorator allows you specify what you want to mock by passing it a string in the format '**package.module.FunctionName**'. In our example there is no

package, but the module is called **simple** and the function is called **simple_function**, so the decorator will look like:

```
@mock.patch('simple.simple_function')
```

We then have to write our second function as so:

```
@mock.patch('simple.simple_function')
def mock_simple_function():
```

I have called it **mock_simple_function**, but it can be called anything. We are missing a parameter here because when you decorate a function with **@mock.patch** it will pass an instance of the MagicMock class (a MagicMock object) that is used to replace the function you are mocking. So it will look like this:

```
@mock.patch('simple.simple_function')
def mock_simple_function(mock_simple_func):
```

So by specifying **@mock.patch('simple.simple_function')** I am saying I want to replace **simple_function** with the MagicMock object assigned to the parameter called **mock_simple_func**.

Initially let's flesh out the function by printing out mock_simple_func and then call it:

```
@mock.patch('simple.simple_function')
def mock_simple_function(mock_simple_func):
    print(mock_simple_func)

mock_simple_function()
```

When run, the output is:

```
<MagicMock name='simple_function' id='4315966936'>
```

Which is a MagicMock object that will be called instead of **simple_function**. It is important to note the **name** attribute of the object is the thing that is being mocked, and the **id** is a unique number for the object.

If we now also print out **simple_function** (but not call it)**:**

```
@mock.patch('simple.simple_function')
def mock_simple_function(mock_simple_func):
    print(mock_simple_func)
    print(simple.simple_function)
```

Which produces the output:

```
<MagicMock name='simple_function' id='4315966936'>
<MagicMock name='simple_function' id='4315966936'>
```

You can see that from the matching ids that **mock_simple_func** is the same MagicMock object as **simple_function**.

So far so good. We have mocked **simple_function**. To be explicit, **simple_function** has been mocked because it has been replaced with a MagicMock object.

Let's add the lines in from the first function:

```
@mock.patch('simple.simple_function')
def mock_simple_function(mock_simple_func):
    print(mock_simple_func)
    print(simple.simple_function)
    result = simple.simple_function()
    print(result)
```

When run, the output is:

```
<MagicMock name='simple_function' id='4315966936'>
<MagicMock name='simple_function' id='4315966936'>
```

```
<MagicMock name='simple_function()' id='4463533752'>
```

As you can see from the third line of output when **simple_function** is actually called it creates a different MagicMock object. Remember that since we have mocked **simple_function,** it is actually a MagicMock object, so when we call **simple_function** we are actually calling the MagicMock object!

For me, this is where the confusion sets in.

- Why is a new MagicMock object created?
- And how on earth can you call an object anyway? You normally get "TypeError: '****' object is not callable" when you try.

Let's pause for a minute to look more closely at MagicMock.

MagicMock has magic in it's name because it has default implementations of most of the python magic methods. What is a python magic method you ask? It is all special python functions that have double underscore at the start and end of their name. You can find a list of them here. The one of interest here is __call__. The call function is described as:

> *Called when an object is called as a function*

So if a class implements this function you can create an instance of the class, and then call that instance like a function i.e. :

```
class ExampleClass(object):
    def __call__(self, *args, **kwargs):
        print("Hell yeah!")

# Create an instance of ExampleClass
inst = ExampleClass()

# Call the object as a function!
inst()
```

Therefore when MagicMock is called as a function, the __call__ function is called. MagicMock's implementation of this function creates a new mock!

So when we call **simple_function()** a new mock is created and returned. Not only that but we have the concept of parent and child mocks. When one mock creates another it becomes the parent and the newly created one the child. That child mock could create other mocks so you end up with a heirarchy of mock objects.

Learning point 1: ***When a MagicMock object is called like a function, by default it creates and returns a new MagicMock object.***

Back to our example. So far we have just mocked **simple_function** but not done anything with it other than print out the MagicMocks that are created as a result. Say I wanted to change what was returned from **simple_function()**, how would I do that? I would do that using the **return_value** property of MagicMock like so (I am also going to remove the first two prints as they are no longer needed):

```
@mock.patch('simple.simple_function')
def mock_simple_function(mock_simple_func):
    mock_simple_func.return_value = "You have mocked
simple_function"
    result = simple.simple_function()
    print(result)
```

When run, the output is:

```
You have mocked simple_function
```

As you can see when **simple_function()** is now called the MagicMock return_value is returned instead of creating a second MagicMock object.

Learning point 2: ***To change what is returned when a MagicMock object is called like a function, set return_value.***

If you want to do more than change the return value then you can use the MagicMock.side_effect feature. It allows you to specify an entirely new function that will be called instead of the one you are mocking. Let's do this. Firstly, I will define a new function that will the side_effect (note it needs to have the same parameter set as the function you are mocking):

```
def side_effect_function():
    return "SKABLAM!"
```

Now we can define a new function that uses this side effect as so:

```
@mock.patch('simple.simple_function')
def
mock_simple_function_with_side_effect(mock_simple_func):
    mock_simple_func.side_effect = side_effect_function
    result = simple.simple_function()
    print(result)
```

Everything is the same as the **mock_simple_function** further up except I am setting the side_effect of mock_simple_func instead of return_value.

The output is:

```
SKABLAM!
```

A good use case for using side effect is if you want to test an error flow and therefore you want to raise an exception in your test. I will change the **side_effect_function** to raise an error instead:

```
def side_effect_function():
    raise FloatingPointError("A disastrous floating point
error has occurred")
```

Now the output is (I removed part of the traceback for clarity):

```
Traceback (most recent call last):
 File “use_simple.py”, line 18, in side_effect_function
 raise FloatingPointError(“A disastrous floating point
error has occurred”)
FloatingPointError: A disastrous floating point error has
occurred
```

Learning point 3: ***To do more than just change the return_value of a MagicMock, set side_effect.***

Mocking Classes

So far we have just mocked a function, but how about a class? Let us define a class in the **simple.py** called **SimpleClass** with a method called **explode**:

```
class SimpleClass(object):
    def explode(self):
        return "KABOOM!"
```

Now in **use_simple.py** let's write a function that uses **SimpleClass** and then call that function:

```
def use_simple_class():
    inst = simple.SimpleClass()
    print(inst.explode())

use_simple_class()
```

When run the output is:

```
KABOOM!
```

Now let's do define a second function and choose to mock the entirety of **simple_class** using the **@mock.patch** decorator:

```
@mock.patch("simple.SimpleClass")
def mock_simple_class(mock_class):
```

As when mocking a function, the **@mock.patch** decorator passes a MagicMock object that replaces the class you are mocking into the function it is decorating. The MagicMock object in this case is assigned to the argument **mock_class**.

For the moment I will just print out the MagicMock object and then run the function:

```
@mock.patch("simple.SimpleClass")
def mock_simple_class(mock_class):
    print(mock_class)

mock_simple_class()
```

When run the output is:

```
<MagicMock name='SimpleClass' id='4321578288'>
```

As with the mocking a function a MagicMock object is created and passed in. Now lets print out SimpleClass so you can see it has been mocked and replaced with the same MagicMock object:

```
@mock.patch("simple.SimpleClass")
def mock_simple_class(mock_class):
    print(mock_class)
    print(simple.SimpleClass)
```

The output now is:

```
<MagicMock name='SimpleClass' id='4359851256'>
<MagicMock name='SimpleClass' id='4359851256'>
```

Let's create an instance of **SimpleClass** i.e. call **SimpleClass()**, and then print it out. Remember, since we have mocked **SimpleClass**, what actually will be happening is that we will be calling the MagicMock object like a function:

```
@mock.patch("simple.SimpleClass")
def mock_simple_class(mock_class):
    print(mock_class)
    print(simple.SimpleClass)
```

```
    inst = simple.SimpleClass()
    print(inst)
```

The output is:

```
<MagicMock name='SimpleClass' id='4359851256'>
<MagicMock name='SimpleClass' id='4359851256'>
<MagicMock name='SimpleClass()' id='4431410624'>
```

So as expected, calling **SimpleClass()** and therefore calling the MagicMock object as a function creates a new MagicMock object.

Up until now, mocking a class has been much like mocking a function. However, in reality when we define a class we will then create objects using that class and it is those objects more often than not that we really want to mock. The question is, how to you mock an instance created from a class? Well, when you mock a class and consequently a MagicMock object is created, if you refer to return_value on that MagicMock object it creates a new MagicMock object that represents the instance of the class created. Blimey! That was a sentence and a half! This was definitely one of my "Aha!" moments, but even writing it down does not make it clear. Let's print out return_value so you can see what I mean:

```
@mock.patch("simple.SimpleClass")
def mock_simple_class(mock_class):
    print(mock_class)
    print(simple.SimpleClass)
    inst = simple.SimpleClass()
    print(inst)
    print(mock_class.return_value)
```

The output is:

```
<MagicMock name='SimpleClass' id='4359851256'>
<MagicMock name='SimpleClass' id='4359851256'>
<MagicMock name='SimpleClass()' id='4431410624'>
<MagicMock name='SimpleClass()' id='4431410624'>
```

The third line is from printing the instance of the class created, and the fourth line is from printing the return_value of the MagicMock object passed in to the function. As you can see they are referring to the same MagicMock object.

Learning point 4: ***When you mock a class, a MagicMock object is created. When you create an instance of that class a new MagicMock object is created. When you refer to the return_value of that class's MagicMock object you get the same MagicMock object that was created when the instance of that class was created.***

"Ahhhhh, why is that important?!?!?" I hear you cry!

"It is important if you want to mock the explode function" I bellow in return!

The next question you need to ask is: is the method I want to mock a class method or an instance method? Because how you change the return_value of the method will be different depending upon the answer.

If the method was a class method then you would set the return_value like so:

```
MagicMockObject.ClassMethodName.return_value = "A
delightful return value"
```

But if it was an instance method you would do:

```
MagicMockObject.return_value.InstanceMethodName.return_valu
e = "A daring return value"
```

With that in mind let's change the return_value of explode, which is an instance method (I'll remove most of the print statements for clarity) :

```
@mock.patch("simple.SimpleClass")
def mock_simple_class(mock_class):
    mock_class.return_value.explode.return_value = "BOO!"
    inst = simple.SimpleClass()
    result = inst.explode()
    print(result)
```

In the code above, **mock_class.return_value** returns the MagicMock object that represents the instance of **SimpleClass**.

mock_class.return_value.explode returns the MagicMock object that represents the **explode** method of the instance of **SimpleClass.**

Therefore setting the **return_value** of **mock_class.return_value.explode** sets the **return_value** of the **explode** method of the instance of **SimpleClass.**

The output is:

```
BOO!
```

Much like when mocking a function, you can set the side_effect for methods of classes and objects too.

I hope this article helped you understand mocking a little more. Please recommend it if you liked it and feel free to leave a response as I would love to hear from you.