Heya! I've been busy with my exams for some time. Though that doesn't imply that I didn't learn anything new. I have learned something crucial, something which plays an important role if you are writing a good piece of code and that something is known as testing. Testing in python can be done in various ways but this post will be dedicated to use mocks in python.

Mock is a library which resides in standard python testing module **`unittest (https://docs.python.org/3/library/unittest.html)`**. So unless you are using an old python version such as Python2, it would already be available for you to use it. In case you are using an old python version you'd need to install it using pip as

```
pip install mock
```

And now you are ready to get along with this post.

What's the need?

If you are already familiar with testing in python, you might be aware of `unittest` module, the standard module used for python testing. Though some of you might've used `pytest` as well. Mock is not another testing framework, rather it is a library which provides utilities to solve a particular problem in testing. Let's understand it with an example

Suppose that your piece of code interact with a third party API. It deals with making requests and getting responses. When testing that piece of code, you'd need to create an actual request to the API. That would actually test your code very well. But as soon as your projects starts to scale or you might need to run the tests every other minute, that would lead to some problems such as

- Dealing with third party API is a time costing approach. It takes a reasonable amount of time to complete a request-response cycle, especially when it involves uploading a file or image. Interacting with API like that every time we run tests definitely won't be a good approach.
- Some APIs costs you money depending on the number of requests made to it. So if you are hitting the API every time you run the tests, that would give your wallet a really big deal.
- There could be situations when the API service is down for some maintenance work and throughout the time you won't be able to run the tests successfully.

Therefore to avoid such situations, we use mock objects.

What is Mock testing?

Now since you know the need of using mock objects, let's understand what they are and how one can use them. **Wikipedia (https://en.wikipedia.org/wiki/Mock_object)** defines a mock object very well as

> *In object-oriented programming, **mock objects** are simulated objects that mimic the behavior of real objects in controlled ways, most often as part of a software testing initiative. A programmer typically creates a mock object to test the behavior of some other object, in much the same way that a car designer uses a crash test dummy to simulate the dynamic behavior of a human in vehicle impacts.*

As what is said above, a mock object acts like a dummy object which can be used in place of real object. But how do we expect it to work like a real object. Well that's simple! we set the expectations.

Ah! I hear you say "Talk is cheap show me the code!". Okay then, let consider the following code excerpt to understand it easily

```
# getMyProfile.py

import requests
from settings import access_token

def get_response():
    url = "https://api.linkedin.com/v2/me"
    response = requests.get(url, headers={'Authorization': f"Bearer {access_token}"})
    return response


def format_response():
    response = get_response()
    json_response = response.json()
    first_name = json_response.get('localizedFirstName   last_name = json_response.get('localizedLastName   r


print(format_response())
```

The above code sends a GET request to a LinkedIn API endpoint and gets the detail about myself. It then prints my name taken out from the corresponding response.

Now we want to test these two functions. So for that we use the **unittest.mock (https://docs.python.org/3/library/unittest.mock.html)** library which helps us to create the mock objects. Let's go ahead and create a test file to test the functions present in above file.

`unittest.mock` contains within it a large number of functions and properties which can be used to simulate the real objects. Before testing the above code let's understand some basics of how `mock` library works in python

```
>>> from unittest import mock
>>> mock_object = mock.Mock()
>>> mock_object
<Mock id='140023121454864'>
```

The above piece of code shows that we can create a mock object by calling `Mock()` method and it's done. It's as simple as it seems.
There is a `MagicMock()` method as well that is a subclass of `Mock()` with all the magic methods pre-created and ready to use. You know what magic methods are, don't you? In case you don't, these are the inbuilt methods that start and end with a double underscore in python such as `__str__`, `__add__`, `__sub__`, `__init__` etc. They all called when a certain specific operation is performed eg `__str__` returns its value when you try to print an object. Let's apply the same on a `MagicMock` object

```
>>> mock_object = mock.MagicMock()
>>> mock_object.__str__.return_value = '5'
>>> print(mock_object)
5
```

As you can see, we have tried to configure the mock object by assigning a return value to it and that is how we actually set expectations to a mock object which defines how it should behave.

patch() method

We've understood the basics of mock objects, let's head back to our code excerpt and write the tests for it. To mock classes or functions residing at some specific location, we use `patch()` method which resides in `unittest.mock` library. The `patch()` method is helpful when we want to mock an object temporarily. It can be used as a decorator as well as a context manager.

`patch()` method takes the path to the function or class which is to be mocked.

Where to patch?

The basic principle which is to be focused upon whenever using patch is that we patch the object where it is used which is not necessarily the same place where it is defined. We'll see it in a few moment, so be alert when you see a patch() definition.

Case I: As a decorator

When used as a decorator, it returns the `MagicMock` object of the given function or class as an argument to the test function it is applied to.
Let's first write the test for `get_response` function. The function sends a GET request to the given url with given headers and returns the response. But we don't want to send the actual GET request in test. Hence we'd need to mock the `requests.get()` method for it.

```
import requests
from unittest import mock

from settings import access_token
from getMyProfile import get_response, format_response


@mock.patch('getMyProfile.requests.getef test_get_response(mock_get):
    url = "https://api.linkedin.com/v2/me"
    headers = {'Authorization': f"Bearer {access_token}"}
    mock_get.return_value = requests.Response()
    response = get_response()
    mock_get.assert_called_once_with(url, headers=headers)
    assert isinstance(response, requests.Response)
```

I have defined the patch decorator on the `test_get_response` method. Notice the path I have given to it. I have patched the `requests.get` method that resides inside the `getMyProfile` module rather than wherever it might have been defined. Hence, it will replace the actual `requests.get` method within the `getMyProfile` file with the `mock_get` object which will be a mock object.

I'll jot down the execution of above code in following points

- We set expectations by configuring the `return_value` attribute on the `mock_get` object by setting it to the `requests.Response()` type. It implies that `mock_get` should return a `Response` object which is what actual `requests.get()` method returns.
- We call the `get_response` function and collects its output within `response` variable.
- `unittest.mock` library provides various assertion methods one of which has been used in above defined test. We've used `assert_called_once_with()` method which as its name suggest assert if a mock object is called exactly once with specified arguments.
- Next assertion checks if the response provided by `get_response` function is a `Response` object.

Now let's complete the test for the other function as well. You might be able to understand it more easily now

```
...

@mock.patch('getMyProfile.requests.Response.jsonmock.patch('getMyProfile.get_responseef test_format_response(
    mock_get_response.return_value = requests.Response()
    mock_json.return_value = {
        'localizedFirstName': 'Prashant',
        'localizedLastName': 'Sharma'
    }
    formatted_response = format_response()
    mock_get_response.assert_called_once()
    assert formatted_response == 'Prashant Sharma'
```

The `format_response` function involves taking output from `get_response` function and returning the full name.

Let's jot down the functioning of above test as well in few points

- We don't want to actually call the `get_response` function within the tests as that will involve sending a request to the API. Therefore we can create a mock object of that function and use it within our test. So we give the path to the `get_response` method to the `patch` method which provides the mock object in the form of `mock_get_response` .
- `format_response` function also takes the json output from response using `response.json()` method. Since, we are not getting the actual response, we need to simulate the `json` method as well. So, here as well we give `patch` , the path to the `json()` method where it's been used which returns the mock object in the form of `mock_json` .
- We then set the expectations on both the mock objects and calls the actual `format_response()` .
- We make the assertion using one of the assertion methods provided by `unittest.mock` library i.e. `assert_called_once()` and assert the return value of the function.

You might have noticed that we take the mock object within the function definition in the same order as the patch is defined ie from downwards to upwards.

And that's it. This is how mocks work and simplify the testing. The scope of mock objects when used as a decorator lasts until the block of the function or class on which it is applied.

Case II: As a context manager

The usage of `patch` remains same, only the way it is defined changes. A `patch` is used as a context manager when we want to apply it within a block or context. If `test_get_response` has to be written using the `patch` as a context manager, it would seem like this

```
def test_get_response():
    url = "https://api.linkedin.com/v2/me"
    headers = {'Authorization': f"Bearer {access_token}"}
    with mock.patch('getMyProfile.requests.gets mock_get:
        mock_get.return_value = requests.Response()
        response = get_response()
        mock_get.assert_called_once_with(url, headers=headers)
    assert isinstance(response, requests.Response)
```

Here we created the context using the python `with` statement. Now `mock_get` object works within the `with` block. That's why all the statements which need to use `mock_get` object are defined within the block. As you may see the functioning remains same, what differs is the way it looks.

I like to use the patch as a decorator but that's a personal choice. You may like to use it as a context manager. That sometimes is subjective to the requirement and the code as well.

Hush! this was a quite long post. Though I hope, I was able to help you understand the basics of mock testing in python in an easy way. But, it's just a introduction and a basic idea of how does it work. There are a lot of methods and properties that awaits you to read about them and use them. I leave the rest to your curiosity.

References and further reading

1. **https://docs.python.org/3/library/unittest.mock.html (https://docs.python.org/3/library/unittest.mock.html)**
2. **Lisa Roach – Demystifying the Patch Function – PyCon 2018 (https://www.youtube.com/watch?v=ww1UsGZV8fQ)**
3. **Ana Balica – To mock, or not to mock, that is the question – PyCon 2016 (https://www.youtube.com/watch?v=KYG5C1CEkOk)**
4. **Mocking External APIs in Python (https://realpython.com/testing-third-party-apis-with-mocks/)**

Well then, see you next time. Till then be curious and keep learning!

Published by gutsytechster

I am an open source enthusiast who is always curious to know more about tech stuff. Let's get acquainted with each other and learn together. **View all posts by gutsytechster**

Posted on **May 12, 2019May 12, 2019** by **gutsytechster**Posted in **Programming/Languages**, **python**Tagged **mocks**, **patch()**, **pytest**, **python**, **stubs**, **testing**, **unittest.mock**.

Related

How to post on LinkedIn via its API using Python?

In "Programming/Languages"

Python testing with pytest

In "python"

How to setup a cron job in Django

In "django"

7 thoughts on “Mock Testing in Python”

1. **JASON BRAGANZA** SAYS:
   May 12, 2019 at 4:42 pm
   wonderfully written as usual.
   keep building up this body of work.
   proud of you

   ⚜ **Reply**

   GUTSYTECHSTER SAYS:
   May 12, 2019 at 6:10 pm
   Thank you! 🙂

   ⚜ **Reply**
2. **SHIVA SAXENA** SAYS:
   May 23, 2019 at 7:01 pm
   Whoa! It’s fantastic. 🙂
   And even I like to use *patch* as a decorator.

   ⚜ **Reply**

GUTSYTECHSTER SAYS:
May 23, 2019 at 7:14 pm
Thank you 🙂

⚜ **Reply**

3. **JEREMY GRIFSKI** SAYS:
June 5, 2019 at 10:19 pm
Great stuff! I've played around with mocking socket connections and whatnot in the past, but it's been awhile. Thanks for the refresher.

⚜ **Reply**

GUTSYTECHSTER SAYS:
June 5, 2019 at 10:54 pm
Thank you so much. It's great to know that it helped. I feel good every time someone gets benefit from what I write and I guess that's what motivates me to keep writing.
🙂

⚜ **Reply**

1. **JEREMY GRIFSKI** SAYS:
June 5, 2019 at 10:56 pm
Oh, for sure! You should keep writing.

⚜ **Reply**