The Patch

The mock module has a neat little function called **patch** that can be used as a function decorator, a class decorator or even a context manager. This will allow you to easily create mock classes or objects in a module that you want to test as it will be replaced by a mock.

Let's start out by creating a simple function for reading web pages. We will call it **webreader.py**. Here's the code:

```
import urllib.request

def read_webpage(url):
    response = urllib.request.urlopen(url)
    return response.read()
```

This code is pretty self-explanatory. All it does is take a URL, opens the page, reads the HTML and returns it. Now in our test environment we don't want to get bogged down reading data from websites especially is our application happens to be a web crawler that downloads gigabytes worth of data every day. Instead, we want to create a mocked version of Python's urllib so that we can call our function above without actually downloading anything.

Let's create a file named **mock_webreader.py** and save it in the same location as the code above. Then put the following code into it:

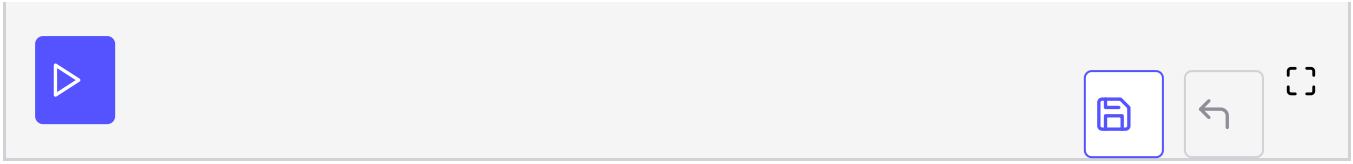
```
# import webreader

from unittest.mock import patch


@patch('urllib.request.urlopen')
def dummy_reader(mock_obj):
    result = read_webpage('https://www.google.com/') #webreader.read_webpage('https://www.goo
    mock_obj.assert_called_with('https://www.google.com/')
    print(result)

if __name__ == '__main__':
```

```
    dummy_reader()
```



Here we just import our previously created module and the **patch** function from the mock module. Then we create a decorator that patches **urllib.request.urlopen**. Inside the function, we call our webreader module's **read_webpage** function with Google's URL and print the result. If you run this code, you will see that instead of getting HTML for our result, we get a MagicMock object instead. This demonstrates the power of patch. We can now prevent the downloading of data while still calling the original function correctly.

The documentation points out that you can stack path decorators just as you can with regular decorators. So if you have a really complex function that accesses databases or writes file or pretty much anything else, you can add multiple patches to prevent side effects from happening.

Wrapping Up

The mock module is quite useful and very powerful. It also takes some time to learn how to use properly and effectively. There are lots of examples in the Python documentation although they are all simple examples with dummy classes. I think you will find this module useful for creating robust tests that can run quickly without having unintentional side effects.