Week 9

Mocking in Python

What is Mocking (Mock Objects)?

When unit testing a piece of code, we want to test it in isolation. However, if the code requires some external dependency to run, e.g., a connection to a database or a webservice, this can be a problem. We do not want to use this external component when we are just testing the piece of code that uses it. How can we do it? We can simulate the external component! For that, we will use mock objects.

When we mock an object, we create a simulation of this object. To handle external dependencies, we mock (simulate) the class in the system that interacts with the dependency. Instead of doing the actual work of the external components, mock objects just return fake, hard-coded results. These return values can be configured inside of the test itself. We will see how it works in practice soon.

The use of mock objects has some advantages. Returning these pre-configured values is way faster than accessing an external component. Simulating objects also gives us a lot more control. If we, for example, want to make sure the system keeps going even if one of its dependencies crashes, we can just tell the simulation to crash; think of how hard would it be to crash a database just to test if your system reacts to that well (although techniques such as chaos monkey have become really popular!).

Mock objects are therefore widely used in software testing, mainly to increase testability. As we have discussed, external systems that the system under test relies on are often mocked to increase controllability and observability. We often mock:

- External components that might be hard to control, or too slow to be used in a unit test, such as databases and web services.
- To simulate exceptions that are hard to trigger in the actual system.
- To control the behaviour of complex third-party libraries.
- To test how the different components interact (i.e., exchange messages) among each other.

Implementation-wise, we follow some steps:

- We create a mock object.

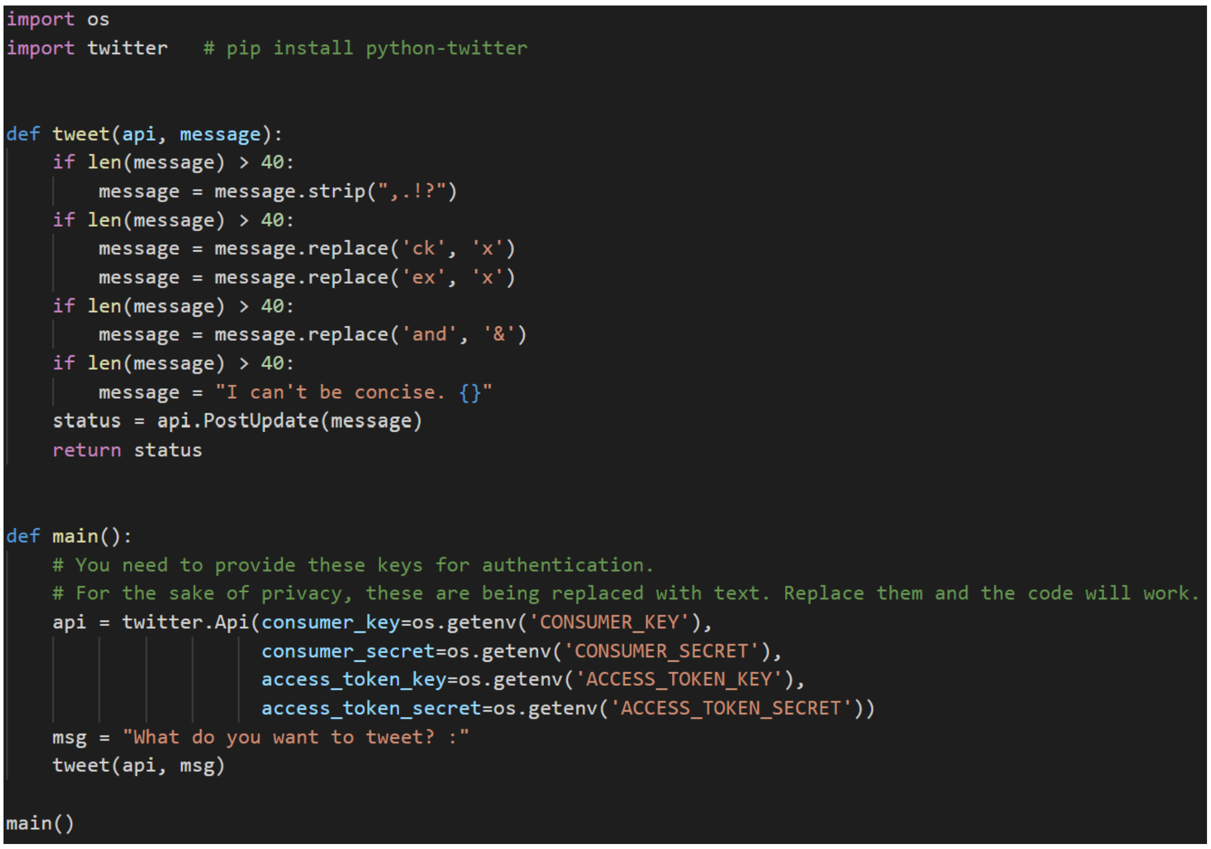
- Once it has been created, we give it to the class that normally uses the concrete implementation of the mocked object. This class is now using the mocked object while the tests are executed.
- At first, the mock does not know how to do anything. So, before running the tests, we have to tell it exactly what to do when a certain method is called.
- We trigger the action on the class/method under test. During its execution, note that the mock replaces the external component.
- We make assertions on the mock object, often related to its execution.

Mocking in Python

Python ships with `unittest.mock`, a powerful part of the standard library for stubbing dependencies and mocking side effects. We use mocking in Python using the `unittest.mock` library. Before going into details let's look at the example.

Example: Mocking Twitter

Let’s look at the code below that connects Twitter API using Python.

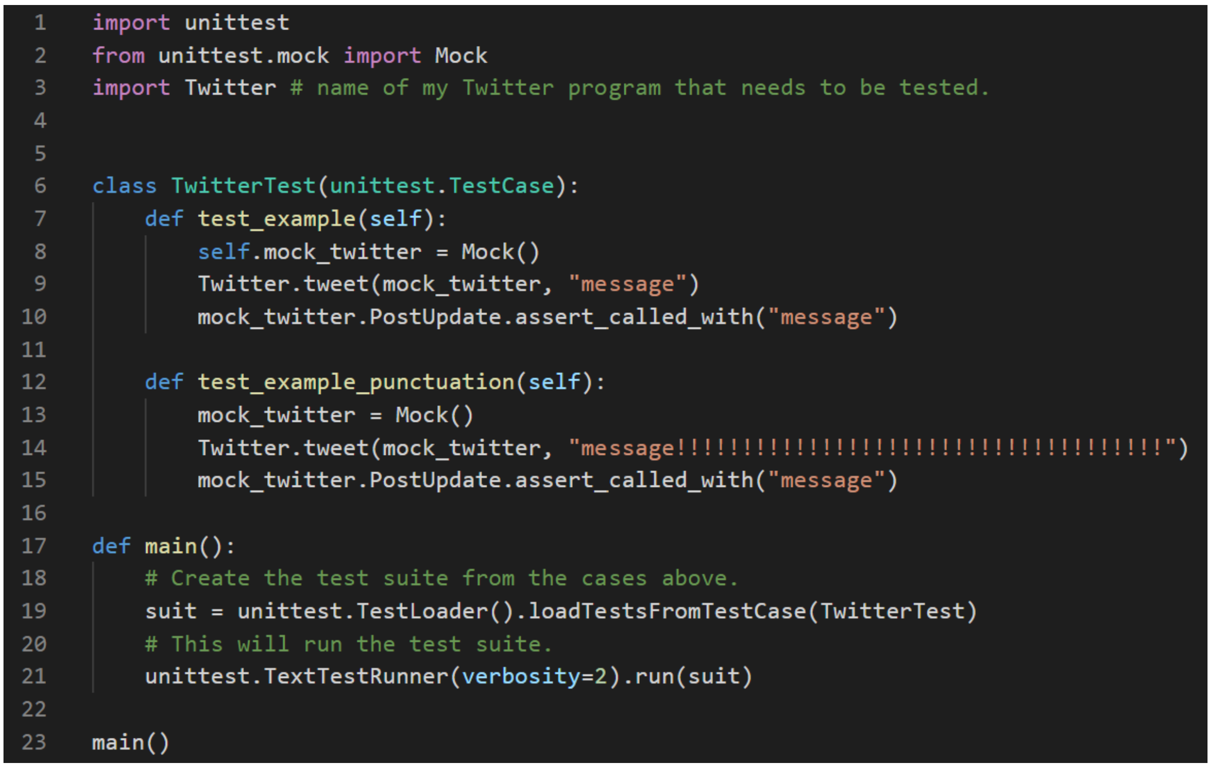


```python
import os
import twitter   # pip install python-twitter


def tweet(api, message):
    if len(message) > 40:
        message = message.strip(",.!?")
    if len(message) > 40:
        message = message.replace('ck', 'x')
        message = message.replace('ex', 'x')
    if len(message) > 40:
        message = message.replace('and', '&')
    if len(message) > 40:
        message = "I can't be concise. {}"
    status = api.PostUpdate(message)
    return status


def main():
    # You need to provide these keys for authentication.
    # For the sake of privacy, these are being replaced with text. Replace them and the code will work.
    api = twitter.Api(consumer_key=os.getenv('CONSUMER_KEY'),
                      consumer_secret=os.getenv('CONSUMER_SECRET'),
                      access_token_key=os.getenv('ACCESS_TOKEN_KEY'),
                      access_token_secret=os.getenv('ACCESS_TOKEN_SECRET'))
    msg = "What do you want to tweet? :"
    tweet(api, msg)

main()
```

Figure 1: Twitter code in Python

Every time we send requests to the twitter API, we are making extra calls to the api which may change, update the data - not the intended behaviour we expect. It may change the values for the client which is not desirable. Call api through mock will create a fake request (a kind of virtual environment), where you can test the intended behaviour before making it live.

Here is the code that tests the functionality presented in the aforementioned code. (Figure 1):



```python
import unittest
from unittest.mock import Mock
import Twitter # name of my Twitter program that needs to be tested.


class TwitterTest(unittest.TestCase):
    def test_example(self):
        self.mock_twitter = Mock()
        Twitter.tweet(mock_twitter, "message")
        mock_twitter.PostUpdate.assert_called_with("message")

    def test_example_punctuation(self):
        mock_twitter = Mock()
        Twitter.tweet(mock_twitter, "message!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!")
        mock_twitter.PostUpdate.assert_called_with("message")

def main():
    # Create the test suite from the cases above.
    suit = unittest.TestLoader().loadTestsFromTestCase(TwitterTest)
    # This will run the test suite.
    unittest.TextTestRunner(verbosity=2).run(suit)

main()
```

Figure 2: Unit Test cases for Twitter.

Development, Staging and Production Environments

In industry, a robust software release management process is vital in ensuring the software being delivered is timely, within budget but most importantly, of high quality (i.e. the software meets all requirements and is thoroughly tested to be as bug-free as possible). A good software release management process usually includes a workflow where software is developed, tested and released in separate environments, namely:

- **Development (dev)**
  - New features are developed, and existing bugs are fixed here.
  - The testing done here is usually focused on testing each component separately, i.e. unit tests.
- **Staging**
  - When enough features have been developed and tested to satisfaction, they will be batched up into a single release and deployed to the staging environment (a software update usually contains several new features and/or bug fixes).

    - This environment should be an identical replica of what you will be deploying to production.
    - The testing done here is mostly focused on testing the entire system, i.e. integration testing, user acceptance testing, performance testing and where feasible, load/stress testing.
- **Production (prod)**
    - Once all tests have been comprehensively passed and the client is happy, the new features/bug fixes will be deployed (pushed) here.
    - Your end users will interact directly with this environment.
    - All data created and stored here are real, actual data.

Why do we need separate environments, and why is this important in the context of software testing?

When we test any piece of software and discover a bug, we want to be sure where it exactly is coming from. Having separate environments allows us to do exactly this, as we will be able to test a newer version of the software <u>in isolation</u> from the current version.

How is this achieved? With separate environments, changes in one environment do not propagate or affect another environment in any way. We can be sure that if a bug was found in production while we were developing a new feature in the development environment, that bug originated from a feature that was previously released, and not from the feature being developed. If we were to use the same environment for development, testing and production, we cannot easily determine if that bug was previously present or newly introduced (that bug could have been located within the new changes itself or even worse, introduced when integrating these new changes into the existing codebase).

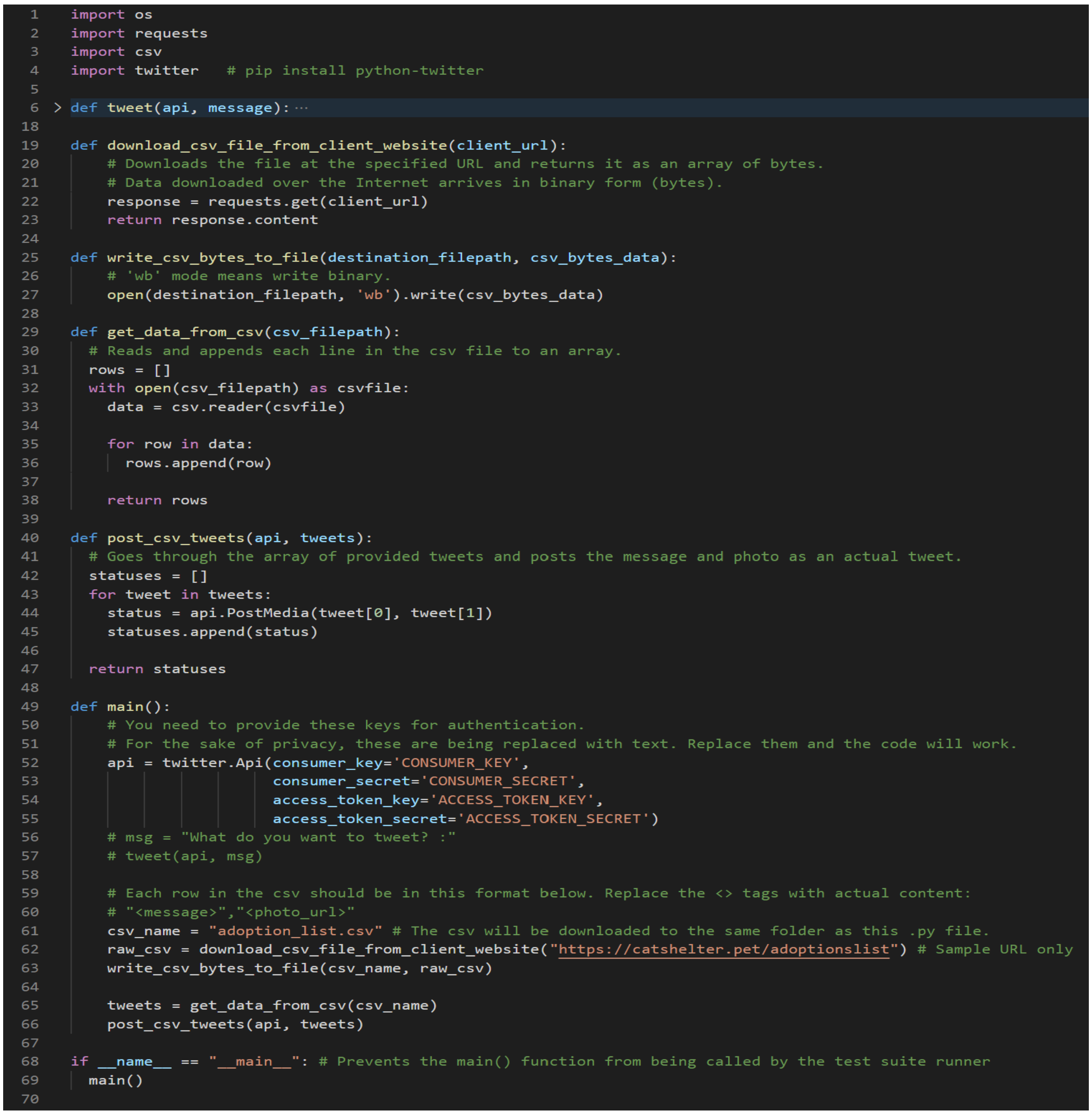
Having isolated environments provides you with a safety net from accidents during software development, and provides you with clearer opportunities to <u>test, discover and fix</u> the problems before it is too late. It is important that you test as much as you can. For example, if you were developing a new database-related functionality in the dev environment and your code accidentally wipes out the entire database, your customers' data in the prod environment will still be safe and unaffected.

Example: A more practical Mocking example using Twitter API

Assume that you have just joined *PyFessional Ltd*, a Python3 software development company who are working on several projects simultaneously, and that one of these projects is the app that helps users post tweets to Twitter which we discussed earlier. The client of this project is a cat adoption centre and wishes to automate the posting of individual tweets for each cat available for adoption. Hence, you have been requested to work together with another colleague to develop a feature that allows the app to be able to read all the messages and image URLs stored in a CSV file and posts them all to Twitter in one go.

Your colleague shares with you that they have actually worked on this project a few years ago until the client decided to temporarily put this project on hold due to a lack of funds. It was only just last week that the client contacted the company to revive the project again. Your colleague also tells you that the *python-twitter* package is being used in this project because the previous team found that it was too error-prone to work with the raw Twitter web API directly, and version 3.3 of the package was in use just before the project was suspended.

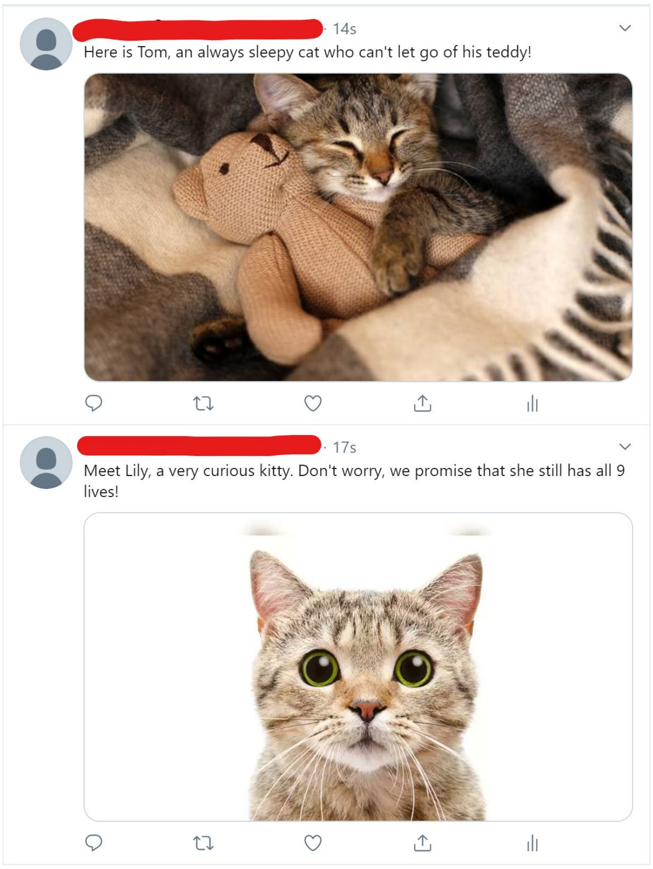
Your colleague has just implemented the requested feature and walks you through the new code that they had written:



```python
import os
import requests
import csv
import twitter   # pip install python-twitter

def tweet(api, message): ···

def download_csv_file_from_client_website(client_url):
    # Downloads the file at the specified URL and returns it as an array of bytes.
    # Data downloaded over the Internet arrives in binary form (bytes).
    response = requests.get(client_url)
    return response.content

def write_csv_bytes_to_file(destination_filepath, csv_bytes_data):
    # 'wb' mode means write binary.
    open(destination_filepath, 'wb').write(csv_bytes_data)

def get_data_from_csv(csv_filepath):
  # Reads and appends each line in the csv file to an array.
  rows = []
  with open(csv_filepath) as csvfile:
    data = csv.reader(csvfile)

    for row in data:
      rows.append(row)

    return rows

def post_csv_tweets(api, tweets):
  # Goes through the array of provided tweets and posts the message and photo as an actual tweet.
  statuses = []
  for tweet in tweets:
    status = api.PostMedia(tweet[0], tweet[1])
    statuses.append(status)

  return statuses

def main():
    # You need to provide these keys for authentication.
    # For the sake of privacy, these are being replaced with text. Replace them and the code will work.
    api = twitter.Api(consumer_key='CONSUMER_KEY',
                      consumer_secret='CONSUMER_SECRET',
                      access_token_key='ACCESS_TOKEN_KEY',
                      access_token_secret='ACCESS_TOKEN_SECRET')
    # msg = "What do you want to tweet? :"
    # tweet(api, msg)

    # Each row in the csv should be in this format below. Replace the <> tags with actual content:
    # "<message>","<photo_url>"
    csv_name = "adoption_list.csv" # The csv will be downloaded to the same folder as this .py file.
    raw_csv = download_csv_file_from_client_website("https://catshelter.pet/adoptionslist") # Sample URL only
    write_csv_bytes_to_file(csv_name, raw_csv)

    tweets = get_data_from_csv(csv_name)
    post_csv_tweets(api, tweets)

if __name__ == "__main__": # Prevents the main() function from being called by the test suite runner
  main()
```

Figure 3: CSV reading test using UnitTest Mock

After the walkthrough, your colleague runs the code and you both see the following output on the test Twitter account, as expected:



[redacted] · 14s

Here is Tom, an always sleepy cat who can't let go of his teddy!

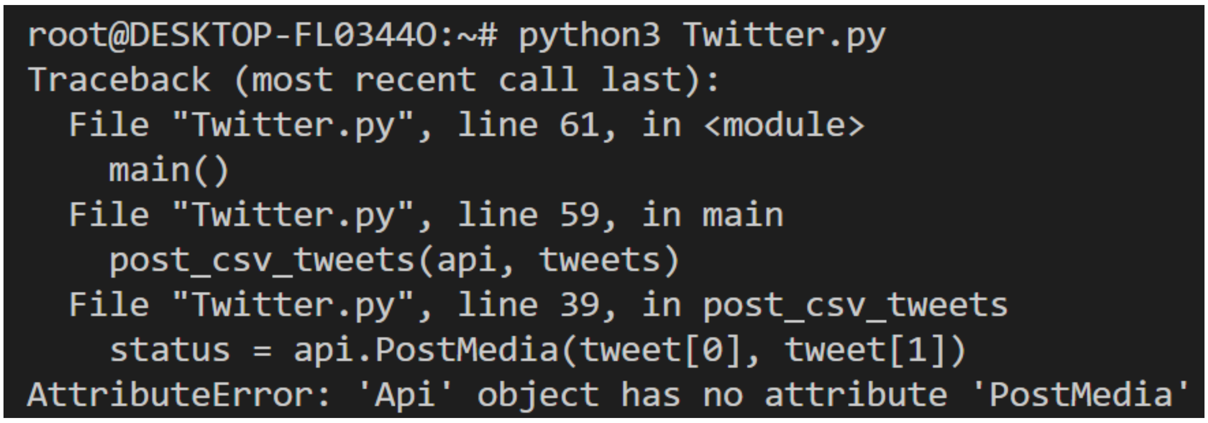
[redacted] · 17s

Meet Lily, a very curious kitty. Don't worry, we promise that she still has all 9 lives!

Figure 4: Output of the code.

Your colleague then tells you to try the code on your own machine. You then clone the project repository from the company's git server, and then proceed to setup the `python-twitter` package on your machine using the `pip install python-twitter` command

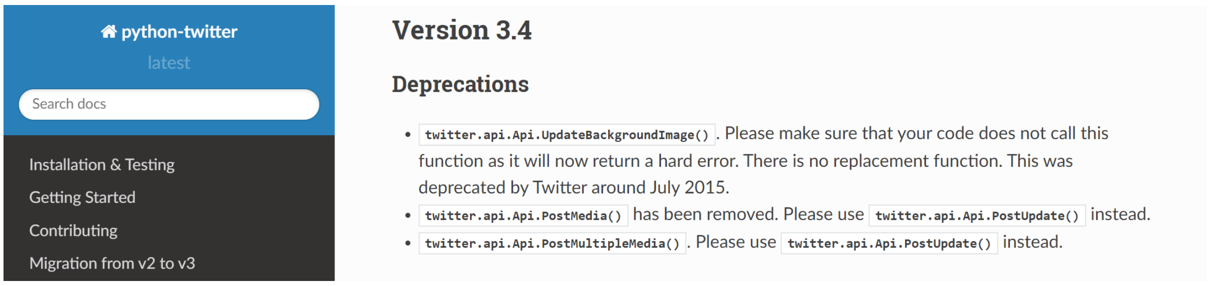
You then try running the app yourself, but you get the following error instead, although you did not touch anything in the code!



```
root@DESKTOP-FL0344O:~# python3 Twitter.py
Traceback (most recent call last):
  File "Twitter.py", line 61, in <module>
    main()
  File "Twitter.py", line 59, in main
    post_csv_tweets(api, tweets)
  File "Twitter.py", line 39, in post_csv_tweets
    status = api.PostMedia(tweet[0], tweet[1])
AttributeError: 'Api' object has no attribute 'PostMedia'
```

Figure 5: Errors? Why?

You quickly check the version of `python-twitter` you have installed using the command `pip` *list* and see that you have version 3.5 (the latest one) installed. You look at `python-twitter's` changelog documentation…



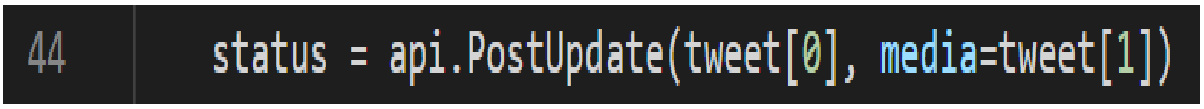
Version 3.4

Deprecations

- `twitter.api.Api.UpdateBackgroundImage()`. Please make sure that your code does not call this function as it will now return a hard error. There is no replacement function. This was deprecated by Twitter around July 2015.
- `twitter.api.Api.PostMedia()` has been removed. Please use `twitter.api.Api.PostUpdate()` instead.
- `twitter.api.Api.PostMultipleMedia()`. Please use `twitter.api.Api.PostUpdate()` instead.

Figure 6: Python Twitter API Version 3.4

...and discover that the `PostMedia()` method your colleague was using has been removed from version 3.4 onwards and replaced with the *PostUpdate()* method. You ask your colleague what version of `python-twitter` they were using, and they confirm it was version 3.3. Fortunately, there are no other breaking changes that are affecting the code given to you, and you do not need to make much changes other than changing your code to use *PostUpdate()* instead of *PostMedia()*, like this:



```
44        status = api.PostUpdate(tweet[0], media=tweet[1])
```

Figure 7: Line 44 of the code.

As you can see, bugs are not just limited to being introduced via the code that you write, but it can also be introduced through the libraries/packages that your application depends on, as well as permissions, configurations and settings on the machine as well. What works on one machine might not work on another machine at all. For example, if you develop on a Windows machine an app that reads from a file, this app might break when you deploy it to a Linux server as file paths are case sensitive in Linux, but not in Windows. Each time you make a change to any part of your software (this includes non-code changes), you must make sure to thoroughly test it. Always read the documentation!

In the next section, you will be shown one way to avoid the issue presented above, and minimise the occurrence and impact of bugs arising from different configurations on different machines, or even conflicting configuration requirements by different applications on the same machine.

How to create an isolated environment for development and testing

Usually, development environments will be hosted/running on separate servers or machines. However, with Python, you can create as many isolated environments as you want on the same machine using *virtualenv*.

To use *virtualenv*, you will need to have `pip`, the Python package installer, already installed on your machine (by default, it should also be installed when you install Python).

Windows

```
python3 -m venv env
```

```
.\env\Scripts\activate
```

Linux/macOS

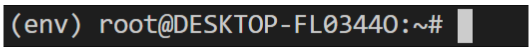
```
python3 -m venv env
```

```
source env/bin/activate
```

If you are unable to get *virtualenv* to run, you might need to run the following command first:

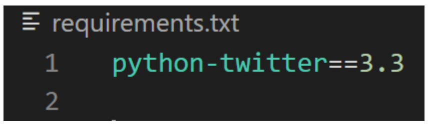
```
sudo apt install python3-venv
```

Once you activate *virtualenv*, you will see "(env)" appended to the start of your command prompt:



```
(env) root@DESKTOP-FL03440:~#
```

The packages that your application needs along with their specific versions can be defined in a *requirements.txt* file, as shown in the image below:



requirements.txt

```
python-twitter==3.3

```

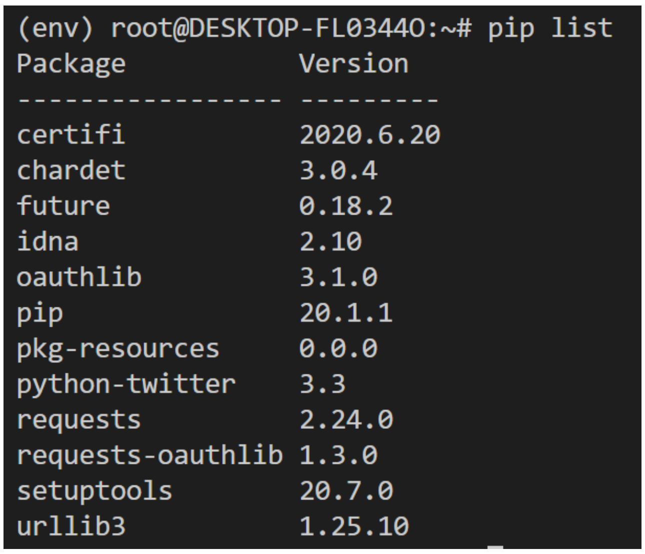
You can then use the following command to install all the dependencies listed in the *requirements.txt* file in one go:

```
pip install -r requirements.txt
```

For the reasons stated in the previous sub-section, it is always recommended to always specify and use a fixed version of any library instead, as `pip` will download the latest versions each time you perform a fresh *pip install* in a new *virtualenv* environment if you do not. You need to test your app whenever you start to use an updated version of any existing dependency.

Dependencies installed within a *virtualenv* environment will not be accessible by other Python applications outside the environment, and any dependency globally installed on your machine will not be visible to your application within the *virtualenv* environment.

This is a list of packages installed within the *virtualenv* environment, using the `pip list` command:



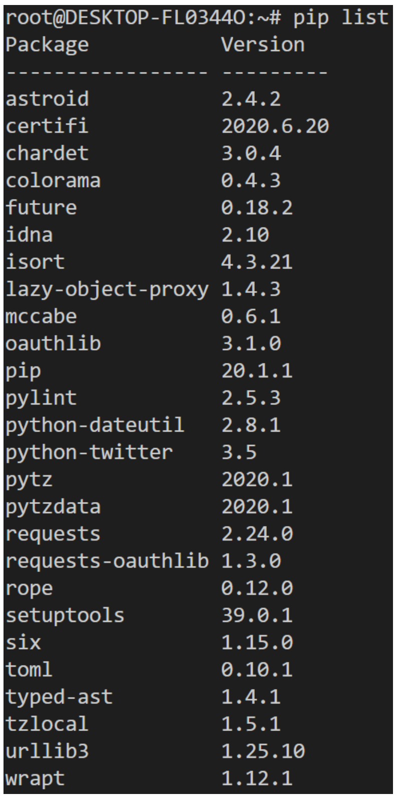
```
(env) root@DESKTOP-FL0344O:~# pip list
Package            Version
------------------ ---------
certifi            2020.6.20
chardet            3.0.4
future             0.18.2
idna               2.10
oauthlib           3.1.0
pip                20.1.1
pkg-resources      0.0.0
python-twitter     3.3
requests           2.24.0
requests-oauthlib  1.3.0
setuptools         20.7.0
urllib3            1.25.10
```

Figure 8: Pip List

To stop *virtualenv* on Windows/Linux/macOS, simply use the command:

```
deactivate
```

Now, let us check the packages that were installed globally and observe the version number of the *python-twitter* package here as well. If we were developing another app on our local machine that also depends on the *python-twitter* package but requires version 3.4 or later, we will be unable to carry out development simultaneously for both apps without breaking one or the other.



```
root@DESKTOP-FL0344O:~# pip list
Package            Version
------------------ ---------
astroid            2.4.2
certifi            2020.6.20
chardet            3.0.4
colorama           0.4.3
future             0.18.2
idna               2.10
isort              4.3.21
lazy-object-proxy  1.4.3
mccabe             0.6.1
oauthlib           3.1.0
pip                20.1.1
pylint             2.5.3
python-dateutil    2.8.1
python-twitter     3.5
pytz               2020.1
pytzdata           2020.1
requests           2.24.0
requests-oauthlib  1.3.0
rope               0.12.0
setuptools         39.0.1
six                1.15.0
toml               0.10.1
typed-ast          1.4.1
tzlocal            1.5.1
urllib3            1.25.10
wrapt              1.12.1
```

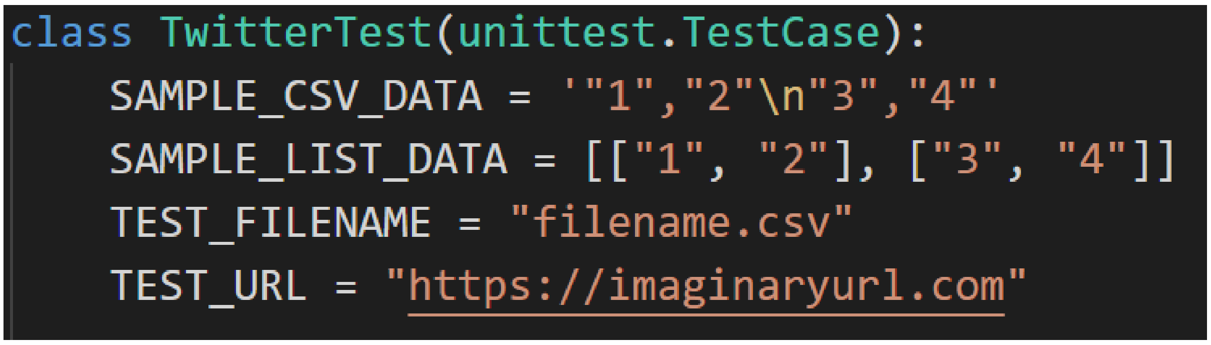
Figure 9: More Pip List

Mock Patching

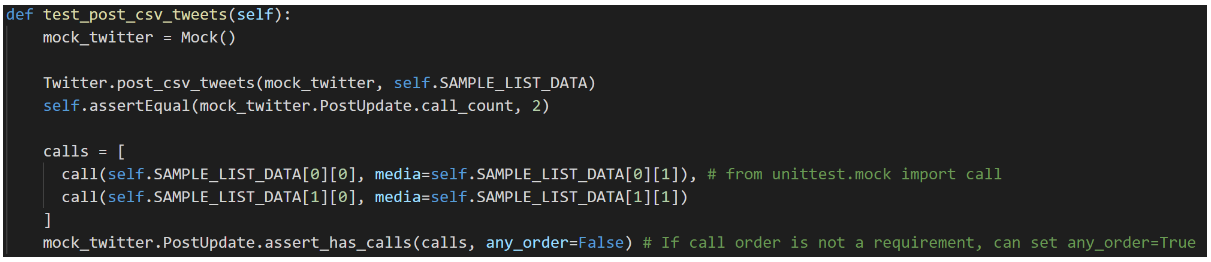
Before we dive into “patching”, let us conceptually revisit what a mock is. As objects and methods take input from their callers and return output, a mock essentially substitutes the object or method being tested. It is a special object which keeps track of how many times you call its methods or accessed its properties/attributes (as well as the order you called them), as well as what arguments you called its methods with. A mock can also be configured to return an output of your choosing, as a response to a method call or property/attribute access. A mock can also return another mock too!

To reiterate, mocks allow for very effective and efficient testing especially when we are testing potentially slow operations such as disk (reading and writing a file) and network I/O (web requests). If we had a test suite with 10 test cases that each make a single web API call that takes 5 seconds per request, we will need to wait at least 50 seconds (assuming no tests fail early) before our test suite finishes running. We also run the risk of not being able to conduct testing whenever an Internet connection is unavailable, or test cases failing due to sudden network disruptions and not because there are bugs in our code. In contrast, if we used mocks to mimic the response returned by that web API, all the tests can be run without an Internet connection and will most likely be fully completed in just a few seconds!

Let us look at more examples of using mocks in test cases. Add the following attributes to the TwitterTest class:


```python
class TwitterTest(unittest.TestCase):
    SAMPLE_CSV_DATA = '"1","2"\n"3","4"'
    SAMPLE_LIST_DATA = [["1", "2"], ["3", "4"]]
    TEST_FILENAME = "filename.csv"
    TEST_URL = "https://imaginaryurl.com"
```

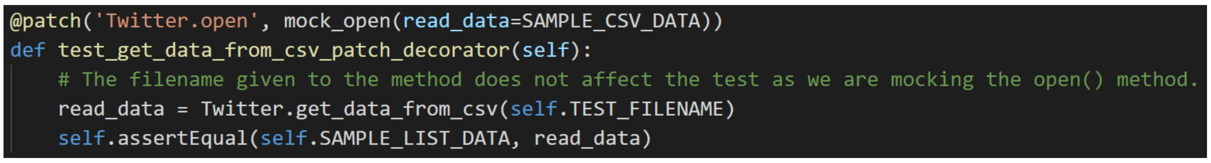
Here is a test case that checks that the app posts all of the tweets as provided CSV file. Note that with mocks, we are even able to check that tweets were made in the correct order. If you run the code, you should notice that we have replaced the API object itself with a mock and hence, we do not need to provide API keys and secrets to verify the correctness of our `post_csv_tweets()` method.


```python
def test_post_csv_tweets(self):
    mock_twitter = Mock()

    Twitter.post_csv_tweets(mock_twitter, self.SAMPLE_LIST_DATA)
    self.assertEqual(mock_twitter.PostUpdate.call_count, 2)

    calls = [
      call(self.SAMPLE_LIST_DATA[0][0], media=self.SAMPLE_LIST_DATA[0][1]), # from unittest.mock import call
      call(self.SAMPLE_LIST_DATA[1][0], media=self.SAMPLE_LIST_DATA[1][1])
    ]
    mock_twitter.PostUpdate.assert_has_calls(calls, any_order=False) # If call order is not a requirement, can set any_order=True
```

Now, let us go back to the main discussion at hand - what do we mean by patching? Patching is essentially the act of substituting a method or object with a mock. In the last three test cases shown to you, we manually patched the API object provided by the `python-twitter` package.

With PyUnit (unittest), we can use the *patch()* method it provides to more effectively and easily patch an object or method under test. The *patch()* method can be used as a decorator, or as a context manager. Let us look into using *patch()* as a decorator first. In Python, a decorator is a specialised method that is attached to another method and adds or modifies its functionality before returning it back to us.

** **Note:** *This example will not work if you are using Python 3.6 or below, as `mock_open()` is buggy in these versions. You will need to use the workaround provided separately.*

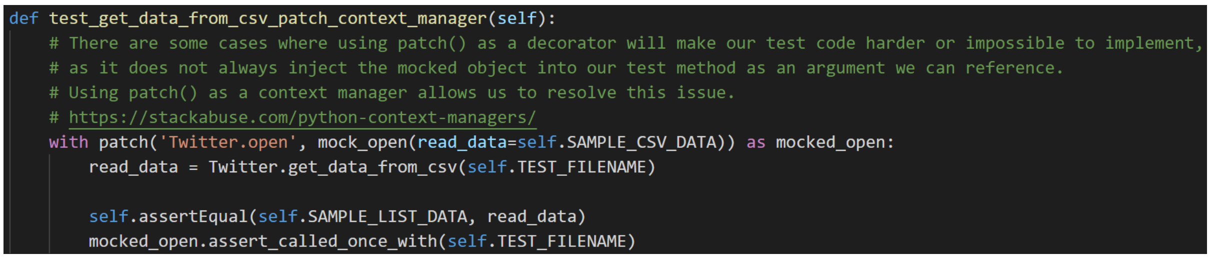

```python
@patch('Twitter.open', mock_open(read_data=SAMPLE_CSV_DATA))
def test_get_data_from_csv_patch_decorator(self):
    # The filename given to the method does not affect the test as we are mocking the open() method.
    read_data = Twitter.get_data_from_csv(self.TEST_FILENAME)
    self.assertEqual(self.SAMPLE_LIST_DATA, read_data)
```

It can become quite complicated to manually mock disk I/O ourselves, so PyUnit offers a special mock called *mock_open* for this purpose. Here, we can tell the *mock_open* object what data to return when a file is opened with it and read from, but do notice that we do not need to provide or create a CSV file of our own when testing at all.

With the `patch()` decorator, we are telling PyUnit to replace all instances of the built-in `open()` method used in our Twitter class with an instance of `mock_open`. A downside of using `patch()` as a decorator is that if you specify the object to patch (the second argument here, i.e. `mock_open()` the target with, it does not provide us with a reference to the mock as a method argument. Hence, in this case, we cannot easily verify that the method under test is only opening just the file we specified, and not reading from other files as well (good code should not have any unexpected behaviour).

We will look at how to mock and test for file writes shortly, but now let us take a look at using *patch()* as a context manager.

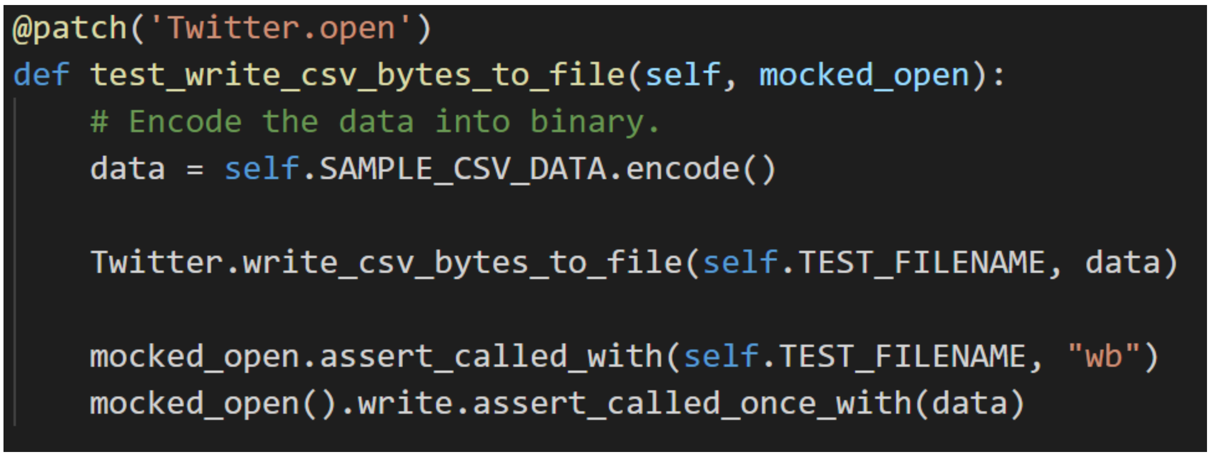
*** Note: Again, this example will not work if you are using Python 3.6 or below, and you will need to use the given workaround code instead.*



```python
def test_get_data_from_csv_patch_context_manager(self):
    # There are some cases where using patch() as a decorator will make our test code harder or impossible to implement,
    # as it does not always inject the mocked object into our test method as an argument we can reference.
    # Using patch() as a context manager allows us to resolve this issue.
    # https://stackabuse.com/python-context-managers/
    with patch('Twitter.open', mock_open(read_data=self.SAMPLE_CSV_DATA)) as mocked_open:
        read_data = Twitter.get_data_from_csv(self.TEST_FILENAME)

        self.assertEqual(self.SAMPLE_LIST_DATA, read_data)
        mocked_open.assert_called_once_with(self.TEST_FILENAME)
```

Here, this test case is functionally equivalent to the one above, except that we can now directly reference the mocked open() object via the local variable mocked_open. There are some scenarios where using patch() as a decorator would be easier and result in cleaner code over using patch() as a context manager, and vice-versa.

Now, let us look at how to mock file writes.



```python
@patch('Twitter.open')
def test_write_csv_bytes_to_file(self, mocked_open):
    # Encode the data into binary.
    data = self.SAMPLE_CSV_DATA.encode()

    Twitter.write_csv_bytes_to_file(self.TEST_FILENAME, data)

    mocked_open.assert_called_with(self.TEST_FILENAME, "wb")
    mocked_open().write.assert_called_once_with(data)
```

In this test case, we are using *patch()* as a decorator again, and now notice that we are mocking the *open()* method with just a regular mock object (not a *mock_open* object) this time. As we did not specify the specific type of mock object to patch the *open()* method with, we now have the mock directly available to us and referenceable in our code via the *mocked_open* argument. We can now test that our *write_csv_bytes_to_file()* method writes correctly (i.e. it writes in binary mode, writes only once and writes to file exactly the data we told it to write). Again, a new CSV file will not be created on our local disk as we are simulating the file writing using mocks.

Mocking Methods, Attributes and Return Values

In the previous section, we have focused mostly on mocking objects. As mentioned earlier, mocks can be told what value it should return when they are accessed and hence, we should be able to specify the return values of its methods and attributes too.

- For methods, this is achieved by assigning the desired value to the *return_value* of the attribute representing the method to mock.
- For attributes, we just assign the value directly to the mock's attribute with the same name, without using the *return_value* property.

Let us look at an example of a test case where we mock an attribute, and run it:

```python
def test_first_attribute_mocking_attempt(self):
    test_object = Mock()
    test_object.content.return_value = "Hi"
    result = test_object.content
    self.assertEqual(result, "Hi")
```

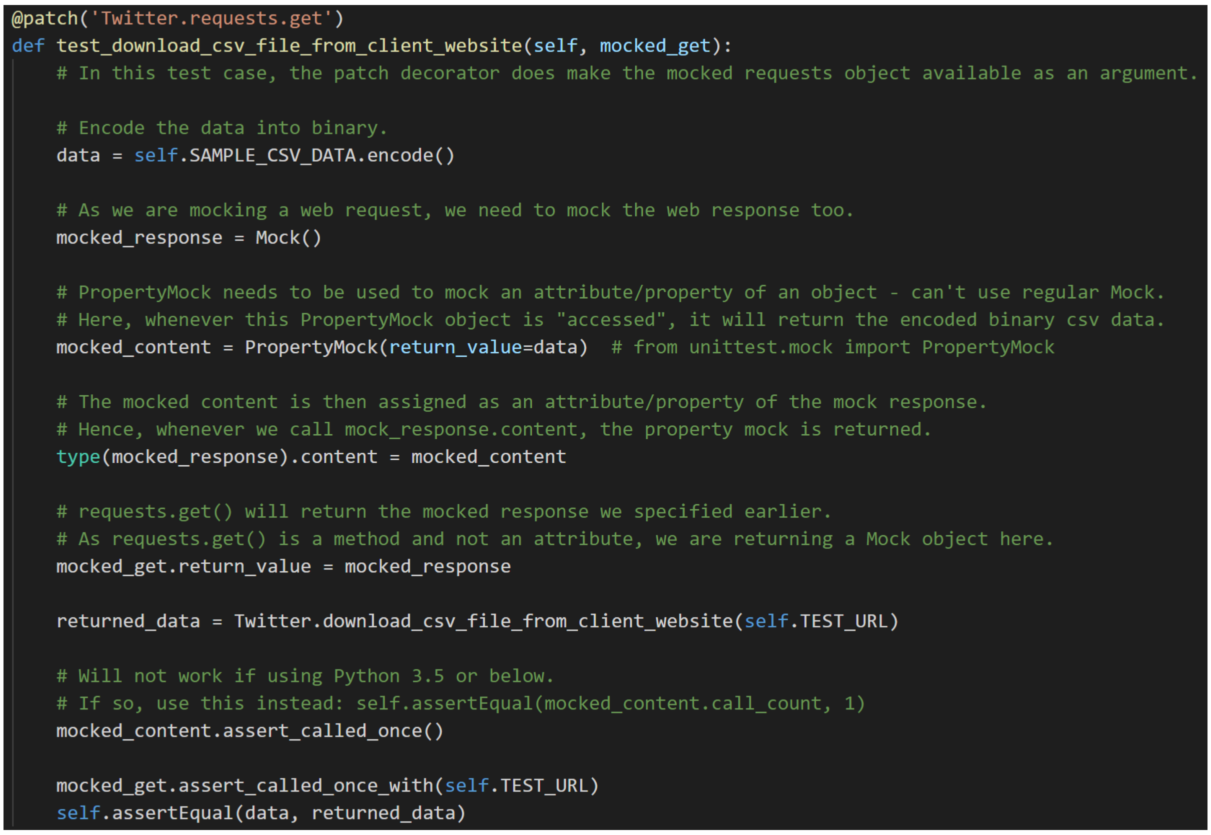
```
======================================================================
FAIL: test_first_attribute_mocking_attempt (__main__.TwitterTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "TwitterTest_workaround_for_Python3.6_and_below.py", line 73, in test_first_attribute_mocking_attempt
    self.assertEqual(result, "Hi")
AssertionError: <Mock name='mock.content' id='140260046379384'> != 'Hi'

----------------------------------------------------------------------
```

The test case fails! So, what happened here? Notice that we are using the *return_value* property here, which means that we are telling the mock object that *content* is a method it has, and not an attribute. To fix this, we just need to remove *return_value* from the second line in code, so that it becomes `test_object.content = "Hi"`.

If we actually intended for *content* to be a method instead, we will need to change the third line of the code above to become `result = test_object.content()` instead, and replace *"Hi"* with a reference to a method. For example, if we have a method called `do_xyz()` in the TwitterTest class, we will write the following: `test_object.content.return_value = self.do_xyz`

A downside to mocking attributes as described above is that the attribute value we provided is not an actual mock itself, so we cannot verify how many times the attribute was accessed or whether it was accessed at all, etc. If you need to make assertions about an attribute in your test case, you will need to use another type of mock called a PropertyMock, instead.

We will now write a test case for our method `download_csv_file_from_client_website()` using a PropertyMock. The result of the web request will be stored in the `content` attribute of the web response object:



```python
@patch('Twitter.requests.get')
def test_download_csv_file_from_client_website(self, mocked_get):
    # In this test case, the patch decorator does make the mocked requests object available as an argument.

    # Encode the data into binary.
    data = self.SAMPLE_CSV_DATA.encode()

    # As we are mocking a web request, we need to mock the web response too.
    mocked_response = Mock()

    # PropertyMock needs to be used to mock an attribute/property of an object - can't use regular Mock.
    # Here, whenever this PropertyMock object is "accessed", it will return the encoded binary csv data.
    mocked_content = PropertyMock(return_value=data)  # from unittest.mock import PropertyMock

    # The mocked content is then assigned as an attribute/property of the mock response.
    # Hence, whenever we call mock_response.content, the property mock is returned.
    type(mocked_response).content = mocked_content

    # requests.get() will return the mocked response we specified earlier.
    # As requests.get() is a method and not an attribute, we are returning a Mock object here.
    mocked_get.return_value = mocked_response

    returned_data = Twitter.download_csv_file_from_client_website(self.TEST_URL)

    # Will not work if using Python 3.5 or below.
    # If so, use this instead: self.assertEqual(mocked_content.call_count, 1)
    mocked_content.assert_called_once()

    mocked_get.assert_called_once_with(self.TEST_URL)
    self.assertEqual(data, returned_data)
```

Mocking Side Effects

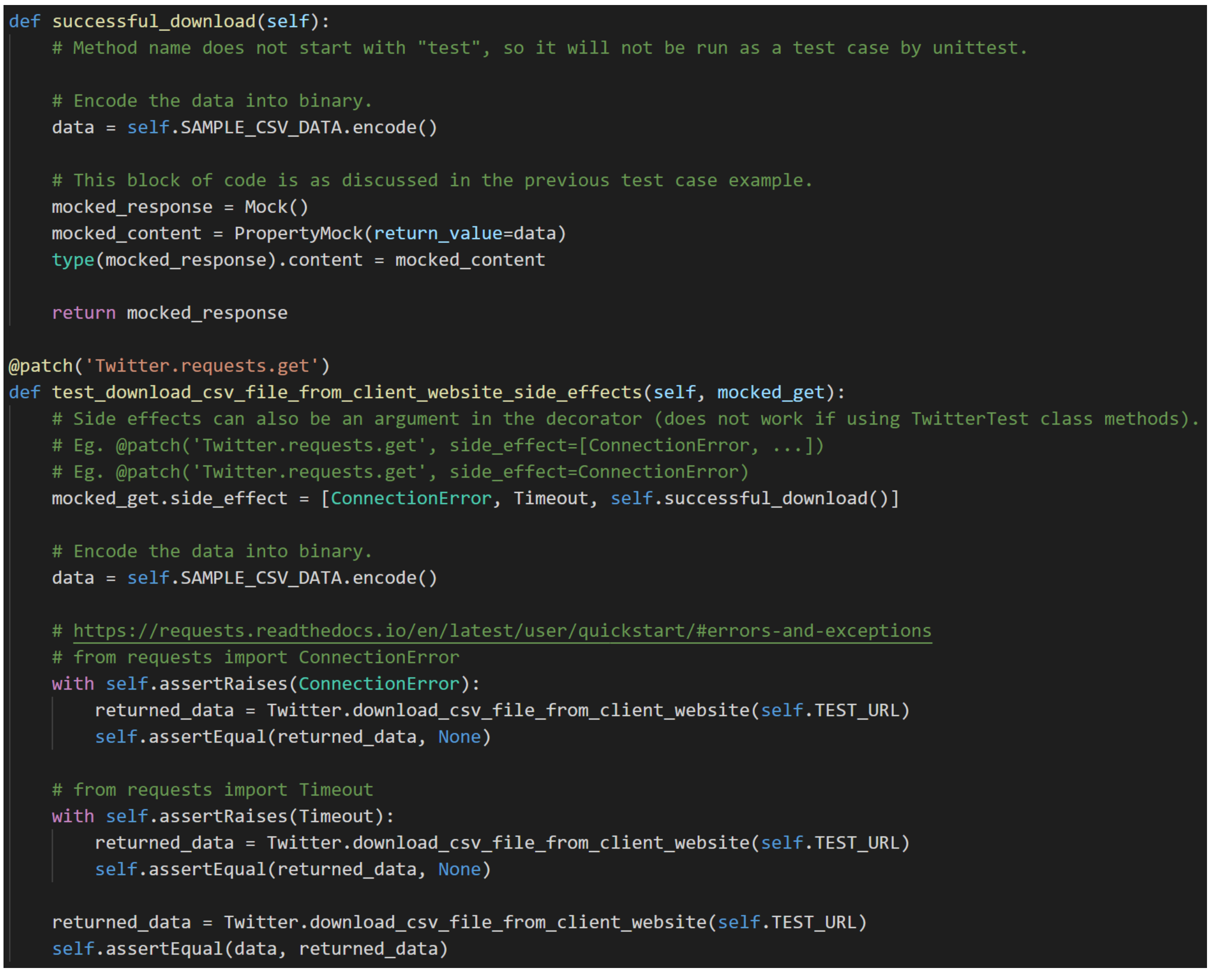
As good programmers, we not only need to test that our code does a task correctly, but also test the failure scenarios. Sometimes, our code might also change the state of something else or produce an unintended result/error – these are what we call *side effects*. For example, if we have an app that hypothetically writes 1GB of data to disk, this write operation might succeed or fail depending on how much remaining disk space the user has. In this case, the side effects of this failure could be that an exception is raised, and the app might log the error to a logfile on our machine, or to the console.

As is the case with return values, we specify side effects a mock should have through its `side_effects` attribute.

In what scenarios should we use side_effects?

- When you want to run another method, each time a mocked object method has been called
  - For example, your original code writes something to a file in addition to its promised functionality, and you would like to test that.
- When you want the return values of the mocked object to be dynamic
  - Side effects can also be specified as an iterable (i.e. a list).
  - Each time the object is called, the next side effect in the list will be run.
- Testing for exceptions
  - Exceptions cannot be realistically raised using *return_value*, as we will be only passing a reference to the exception back to the caller instead of raising it.

As web requests can fail due to several reasons, let us have a look at how using side effects will come handy when testing this:



```python
def successful_download(self):
    # Method name does not start with "test", so it will not be run as a test case by unittest.

    # Encode the data into binary.
    data = self.SAMPLE_CSV_DATA.encode()

    # This block of code is as discussed in the previous test case example.
    mocked_response = Mock()
    mocked_content = PropertyMock(return_value=data)
    type(mocked_response).content = mocked_content

    return mocked_response

@patch('Twitter.requests.get')
def test_download_csv_file_from_client_website_side_effects(self, mocked_get):
    # Side effects can also be an argument in the decorator (does not work if using TwitterTest class methods).
    # Eg. @patch('Twitter.requests.get', side_effect=[ConnectionError, ...])
    # Eg. @patch('Twitter.requests.get', side_effect=ConnectionError)
    mocked_get.side_effect = [ConnectionError, Timeout, self.successful_download()]

    # Encode the data into binary.
    data = self.SAMPLE_CSV_DATA.encode()

    # https://requests.readthedocs.io/en/latest/user/quickstart/#errors-and-exceptions
    # from requests import ConnectionError
    with self.assertRaises(ConnectionError):
        returned_data = Twitter.download_csv_file_from_client_website(self.TEST_URL)
        self.assertEqual(returned_data, None)

    # from requests import Timeout
    with self.assertRaises(Timeout):
        returned_data = Twitter.download_csv_file_from_client_website(self.TEST_URL)
        self.assertEqual(returned_data, None)

    returned_data = Twitter.download_csv_file_from_client_website(self.TEST_URL)
    self.assertEqual(data, returned_data)
```

Python Mock Resources

Official python mock resource is accessible at.

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

Some examples can be found here

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

The Twitter API is accessible at

- https://python-twitter.readthedocs.io/en/latest/changelog.html