3 Ways to Implement the Mock During Python Unit Testing

Aaron Ginder
☁️ Lead Data Engineer at BT

2 articles + Follow

August 3, 2020

For code to be considered complete and reliable, more often than not people will ask "How do you know the function works as expected?" or "Have you tried to break your own code?". Either question, it's clear that unit testing is salient to being confident that your program will execute successfully. The last thing that you want to do is finish writing a program and be stuck fixing code that does not do what it's suppose to.

By the end of this article, you will have an good understanding about what mock objects are and the different ways to mock objects or patch external libraries using Python's unittest library.

What is Unit Testing?

Unit testing is a branch of software testing that assesses if individual blocks of code (called units) can correctly execute to produce an expected result, independent of one another. Unit testing uses the white box testing methodology (or transparent box testing) where the tester has knowledge of the underlying structure of the unit being tested. With this in mind, the tester should only unit test their own code to have confidence in the program.

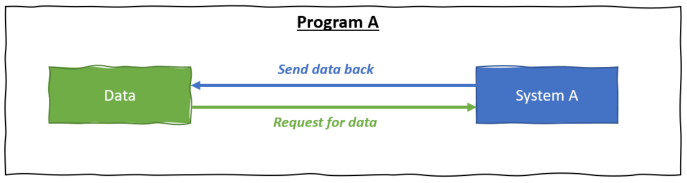
Effective unit tests should also check that the function can handle erroneous data if a wrong input is passed. This is a fundamental step to a robust program, because you will most likely find weird and wonderful data inputs if uncleaned.

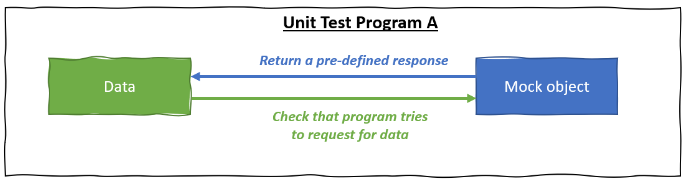
What is the Mock?

As defined in the python unittest documentation...

> **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.

A mock object will allow you to control a component of your program. For example, suppose you have *program A* that retrieves data from *system A,* from a unit testing perspective, your test should not dependent on *system A* working correctly. Rather, your test should check your program attempts to make a data request call to the system and return some data.




Unit Test Program A
Data
Return a pre-defined response
Check that program tries to request for data
Mock object

The way you would essentially switch *System A* with the mock object is using the *patch*. The *patch* temporarily replaces your object with a different object. In the case above, we would us the patch function to replace *system A* with the mock object in program A.

Ways to Patch & Replace an Object with a Mock

In Python 3, there are three common ways to patch an object:

1. Decorator
2. Context Manager
3. Inline

In turn, we will go through the these methods of using the patch with example code. The example we will unit test involves two functions; *get_data()* and *connect_to_db()*.

```
# my_module.py

def get_data():
  db = connect_to_db()
  result = db.query_all_data()
  return result

def connect_to_db():
  ...
```

For unit testing purposes, we do not want to connect to the database and query a table. We want to test that we are calling the database and that some data is returned. By doing it this way, even if the database server crashes, our unit tests will not fail because of an extraneous variable outside the scope of our function declaration.

Decorator

The **decorator**, denoted by the @ symbol above the function definition, is used to patch a callable Python object before the unit test is ran. The way that decorators would work in this context is the database connection will be replaced with a mock object and then the unit test will execute. Therefore, you will not be connecting to the database but rather the configured mock database object.

Below shows a code snippet of using a decorator to unit test the *get_url_html()* function:

```
# test_my_module.py

import unittest
from my_module import get_data
from unittest.mock import Mock, patch

@patch('my_module.connect_to_db')
def test_get_data(self, mock_db):
  mock_db.return_value.query_all_data.return_value =
  result = get_data()

  self.assertEqual(result, 'result data')
  self.assertEqual(mock_db.call_count, 1')
  self.assertEqual(mock_db.query_all_data.call_count
```

We decorate out test function with the patch to the function that we want to replace with a mock object (*connect_to_db*). When patching using a decorator, you must pass an argument of the object as an input to the test function (*mock_db*). Then, we set the *get_query_all_data()* method of the mock_db object to return 'the text result_data'. Now that the *connect_to_db()* function has been replaced with our mock_db object , we execute the *get_data()* function and assign it to the result variable.

AssertEquals is used to for three reasons in our test_get_data function:

1. Check that the result of *get_data()* function returns the result_data as we configured
2. Check that we have tried to connect to our database object. We should only call our database once. Call count is a special attribute of the mock object that totals the number of times the mock object has been called
3. Check that we have executed a query method to retrieve all data. We should only call the *query_all_data()* function once

Context Manager

Context managers can be used to patch objects only within the block of code. In the same way as the decorator example, we patch the database connection with mock_db and check that the respective elements of the function have been called,

```
# test_my_module.py

def test_get_data(self):

  with patch('my_module.connect_to_db', return_value

    mock_db.return_value.query_all_data.return_value
    result = get_data()

    self.assertEqual(result, 'result data')
    self.assertEqual(mock_db.call_count, 1')
    self.assertEqual(mock_db.query_all_data.call_cou
```

The key difference between patching using a decorator compared to a context manager is within the *test_get_data()*, you do not need to pass in mock_db as an argument. If the function was called outside of the context manager, the *connect_to_db* method will not be patched, therefore, you would actually be connecting to the database. Context managers are useful for configuring mock objects in a contained fashion.

Inline

Another alternative to the two patching methods above is inline. If you have wrote unit tests before, you re likely to have seen the setUp() and tearDown() methods of the unittest library. For more informtion about these methods, visit the **Python unittest documentation**.

> The setUp() and tearDown() methods allow you to define instructions that will be executed before and after each test method.

The difference to using inline patching is you must start and stop the patch; often within the setUp and tearDown methods respectively.

```
# test_my_module.py

def test_get_data(self):

  mock_db = patch('my_module.connect_to_db').start()
  mock_db.return_value.query_all_data.return_value =
  result = get_data()

  self.assertEqual(actual, 'result data')
  self.assertEqual(mock_db.call_count, 1)
  self.assertEqual(mock_db.query_all_data.call_count

  mock_db.stop()
```

We patch the connect_to_db and assign it to the mock_db variable. As demonstrated in the code above, the *start()* and *stop()* method is used to to activate (or deactivate) the patching of an object. The only difference in the code demonstration above is these start and stop methods. Remember to start and stop the patch to avoid potential issues with the mock object call_count cumulatively adding for each unit test.

Choosing the Mock Implementation Method

All three methods above are valid and equally as effective to patch a target object with a mock. From my experience, there are two elements to consider when deciding the method to use:

1. Choose the method that makes sense to you
2. Be consistent in your approach to applying the patch

Everyone unit tester has their personal preference on the way to mock an object. Some may find a context manager easier to read, whereas others may prefer to declare all their mock objects via decorators at the top of the function. But try not to use both. When those unit tests are visited in the future, the tester will see the approach you have taken across which makes it easier to amend tests if or when they fail.

Conclusion

Now that you know the wonderful ways to patch an object with a mock, experiment and have fun unit testing your programs!

- Here is a **useful resource** to learning how to patch an object successfully.
- Write functions with complete unit tests by testing different scenarios to see if you function can pass different tests using **Hypothesis**.

Report this

Published by

Aaron Ginder **2 articles** + Follow

Lead Data Engineer at BT
Published • 2y

3 Ways to Implement the Mock During Python Unit Testing

Like Comment Share 15

Reactions

+3

0 Comments

Add a comment…

Aaron Ginder

Lead Data Engineer at BT

+ Follow

More from Aaron Ginder

Collecting Wine Reviews Data Using Apache Airflow & Cloud Composer

Aaron Ginder on LinkedIn