Test Mocking

Contents

Overview

Mocking is the process of faking the output of a function while you are trying to test an application.

This is useful for either:

- When you have some functionality that you either want to **test under specific conditions**.

  For example: a conditional statement that depends on a random value.

- When a part of your application **cannot produce an intended value during a test**.

  For example: making a request to a separate API that is unreachable during testing.

`unittest.mock` and `patch()`

`unittest.mock` is a built-in Python library that allows you to mock the return value from a function. It does this using a function called **patch**.

To import the patch function we need to import it from the library `unittest.mock`.

```
from unittest.mock import patch
```

In this module we are going to use `patch()` as a **function decorator**. Function decorators change how a function behaves without changing the function's code.

This allows us to alter the output of a function while testing without having to change the function itself.

`with`

Since we only want to alter the functionality of a function for one test, we use the `with` statement.

A `with` statement creates a context block, within which our function will simply return a value we define rather than running the actual function's code. This will only be true within the `with` block.

```
with patch('function_we_want_to_patch') as p:
    # change the functionality of the function here
```

Return Value

All python functions have a return value.

We can change the functionality of a function using `patch()`. The simplest way to do this is to change the value which is returned.

For example, the following function either returns `'heads'` or `'tails'` depending on whether the output of `random.randint(0,1)` is `0` (heads) or `1` (tails):

```
import random

def coin_flip():
    if random.randint(0,1) == 0:
        return "heads"
    else:
        return "tails"
```

We could write a test that will assert that the output is either `'heads'` or `'tails'`:

```
import pytest

def test_coin_flip():
    assert coin_flip == "heads" or coin_flip == "tails"
```

But we aren't testing that the number `0` will result in `'heads'` being returned, and that `1` results in `'tails'` being returned.

So we can **mock** the response from `random.randint()` to specifically return either `0` or `1` and assert that the function being tested should return `'heads'` or `'tails'` respectively:

```
import pytest
from unittest.mock import patch

def test_coin_flip_heads():
    with patch('random.randint') as p:
        p.return_value = 0
        assert coin_flip == "heads"

def test_coin_flip_tails():
    with patch('random.randint') as p:
        p.return_value = 1
        assert coin_flip == "tails"
```

Mocking HTTP Requests

HTTP requests in python are done using functions from the `requests` library.

This means that we can mock the responses from these functions so that the requests do not need to be made for us to test an application.

Using `patch()`

Consider this function:

```
import requests

def get_number():
    response = requests.get('http://api:5000/get/num')
    print(response.text)
```

This function makes a request to a theoretical API running on port 5000 that responds with a random number between 1 and 10 as `text`. It then prints the value of the text property to the console.

To mock the response for this request we can patch the `requests.get()` function and give it the mock object as the return value.

Since the python script (above) is looking for a `text` property from the request, we need to ensure that we are assigning a value to this:

```
from unittest.mock import patch

def test_get_number():
    with patch('requests.get') as g:
        g.return_value.text = '1'
```

Now if we are testing the application which makes the HTTP request, we can include the patched response to ensure that we always get the text value of 1 as the response.

Limitations of `patch()`

`patch()` can only mock the return value of a function once for the duration of unit test.

In most cases this does not pose a problem, but causes issues when the same function is invoked multiple times and requires different outputs.

For example, consider a function that retrieved the current temperature in both Celcius and Fahrenheit as two separate calls to a weather API and returns them as a dictionary:

```
import requests

def get_temperature():
    celcius = requests.get('http://weather_api:5000/get/temperature/c')
    fahrenheit = requests.get('http://weather_api:5000/get/temperature/f')
    return {
        "celcius" : f"{celcius.text}°C",
        "fahrenheit" : f"{fahrenheit.text}°F"
    }
```

With `patch()`, we can only mock the return value of `requests.get()` once, meaning the value of `celcius` and `fahrenheit` would be the same.

Using `requests_mock`

`requests_mock` is a separate Python package designed for use with the `requests` module. It is designed to circumvent `unittest.mock`'s limitations to allow you to mock multiple HTTP requests within the same function.

It needs to be installed with pip:

```
pip3 install requests_mock
```

We can then mock our HTTP requests to the weather API from before using our `with` statement, much like with `patch()`:

```
import requests_mock

def test_get_temperature():
    with requests_mock.Mocker() as m:
        m.get('http://weather_api:5000/get/temperature/c', text='15')
        m.get('http://weather_api:5000/get/temperature/f', text='59')
        assert get_temperature()["celcius"] == "15°C"
        assert get_temperature()["fahrenheit"] == "59°F"
```

Despite being the same function, `requests_mock.Mocker()` can distingusih between two separate invocations of `requests.get()` based on the URL they are sending their HTTP requests to.

The mocked return value is provided by the `text=<value>` argument. You can also mock JSON responses by replacing the `text` argument with `json={"some":"dictionary}`.

Tutorial

In this tutorial we will make a simple flask application which makes an HTTP request to an API, we will then test the application using a mocked response from the API.

Prerequisites

For this tutorial, you will need a Linux machine/virtual machine with the following installed:

- python3
- python3-pip
- python3-venv

Setup

Create a new directory called `mock-testing` and change directory to it.

```
mkdir mock-testing && cd $_
```

Create a virtual environment and install `flask`, `flask-testing`, `pytest`, `pytest-cov`, `requests`.

```
python3 -m venv venv
. ./venv/bin/activate

pip3 install flask flask-testing pytest pytest-cov requests
```

Create a directory called `application` and one called `testing`.

```
mkdir application testing
```

Create the following files:

- `app.py`
- `application/__init__.py`
- `application/routes.py`
- `testing/__init__.py`
- `testing/test_mock.py`

```
touch app.py application/{__init__.py,routes.py}
testing/{__init__.py,test_mock.py}
```

Inside `application/__init__.py`, we will create the app object use to run the application.

```
from flask import Flask
app = Flask(__name__)
from application import routes
```

Inside `app.py` we're going to put the code to run the application using the app object.

```
from application import app

if __name__ == '__main__':
    app.run(port=5000, host='0.0.0.0')
```

Inside `application/routes.py` we will create one route which will make a request for a random number and either return a sport depending on which number is in the response.

```
from application import app
import requests

@app.route('/get/sport', methods=['GET'])
def sport():
    response = requests.get('http://api:5000/get/number')
    if response.text == "1":
        return "Football"
    elif response.text == "2":
        return "Badminton"
    elif response.text == "3":
        return "Hockey"
    else:
        return "Boxing"
```

Inside `testing/test_mock.py` we will write a test which uses a mocked response and show that we get the required value returned.

```
from unittest.mock import patch
from flask import url_for
from flask_testing import TestCase

from application import app

class TestBase(TestCase):
    def create_app(self):
        return app

class TestResponse(TestBase):

    def test_football(self):
    # We will mock a response of 1 and test that we get football returned.
        with patch('requests.get') as g:
            g.return_value.text = "1"

            response = self.client.get(url_for('sport'))
            self.assertIn(b'Football', response.data)
```

Ensuring you are in the directory `mock-testing` then running the command

```
pytest
```

you should see that we have one test and it passed.

Exercises

1. In the directory `mock-testing` run the command:

   ```
   pytest --cov=application
   ```

   You should see that we have `64% coverage`.Try and improve this to reach `100% coverage`.

   ▶ Hint

2. Replace the route in the tutorial with:

   ```
   @app.route('/get/sport', methods=['GET'])
   def sport():
       number = requests.get('http://api:5000/get/number').text
       letter = requests.get('http://api:5000/get/letter').text
       if number == "1" and letter == "a":
           return "Football"
       elif number == "1" and letter == "b":
           return "Badminton"
       elif number == "1" and letter == "c":
           return "Hockey"
       else:
           return "Boxing"
   ```

Update your unit test to get 100% coverage for this new function.

▶ Hint