```
        blog.posts.assert_called_with() # We called the posts method with no ar

        blog.posts.assert_called_once_with() # We called the posts method once

        # blog.posts.assert_called_with(1, 2, 3) - This assertion is False and

        blog.reset_mock() # Reset the mock object

        blog.posts.assert_not_called() # After resetting, posts has not been ca
```

As stated earlier, the mock object allows us to test how it was used by checking the way it was called and which arguments were passed, not just the return value.

Mock objects can also be reset to a pristine state i.e. the mock object has not been called yet. This is especially useful when you want to make multiple calls to your mock and want each one to run on a fresh instance of the mock.

Side Effects

These are the things that you want to happen when your mock function is called. Common examples are calling another function or raising exceptions.

Let us revisit our `sum` function. What if, instead of hard coding a return value, we wanted to run a custom `sum` function instead? Our custom function will mock out the undesired long running `time.sleep` call and only remain with the actual summing functionality we want to test. We can simply define a `side_effect` in our test.

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

def mock_sum(a, b):
    # mock sum function without the long running time.sleep
    return a + b

class TestCalculator(TestCase):
    @patch('main.Calculator.sum', side_effect=mock_sum)
    def test_sum(self, sum):
        self.assertEqual(sum(2,3), 5)
        self.assertEqual(sum(7,3), 10)
```