Python Mock Tutorial Documentation

Release 0.1

Javier Collado

Nov 14, 2017

Contents

Contents:

CHAPTER 1

Introduction

mock is a python library than can be used together with unittest write better tests for your code. This document's goal is to get the reader started quickly using mock. None of the examples below will be difficult to follow, but a basic knowledge of python and unittest library is recommended.

CHAPTER 2

Mock

2.1 What is a mock object?

A mock object is an instance of the `Mock` class or any of its subclasses:

```
>>> from mock import Mock
>>> my_mock = Mock()
```

It supports attribute access and method calls:

```
>>> my_mock.attribute
<Mock name='mock.attribute' id='...'>
>>> my_mock.method()
<Mock name='mock.method()' id='...'>
```

Index access is no supported by default because magic methods, those whose names are surrounded by double underscores, require a special configuration because of the way they are looked up internally in python. However, there is a special subclass of `Mock` called `MagicMock` that provides a default implementation for most magic methods including *__getitem__*:

```
>>> from mock import MagicMock
>>> my_mock = MagicMock()
>>> my_mock['index']
<MagicMock name='mock.__getitem__()' id='...'>
```

Given that index access is usually required in test cases and that `MagicMock` provides that functionality out of the box, it's common to use `MagicMock` as an alias of `Mock` by default using `import ... as`:

```
>>> from mock import MagicMock as Mock
>>> my_mock.attribute
<MagicMock name='mock.attribute' id='...'>
>>> my_mock.method()
<MagicMock name='mock.method()' id='...'>
```

```
>>> my_mock['index']
<MagicMock name='mock.__getitem__()' id='...'>
```

2.2 What makes mock objects useful for testing?

As you may have noticed from the examples above, when an attribute is accessed (or any other of the supported operations), another mock object is returned. What you may have not noticed,though, is that always the same mock object is returned, that is, on the first call a new mock object is created that is cached and returned on any subsequent calls:

```
>>> mock_attribute_1 = my_mock.attribute
>>> mock_attribute_2 = my_mock.attribute
>>> mock_attribute_1 is mock_attribute_2
True
```

This is a useful property because it makes possible to get the mock objects that are going to be used in the unit under test in advance to configure them properly depending on the need of each test case.

2.3 How are methods mocked?

2.3.1 Setting return values

Sometimes when a call is made on a mock object that pretends to be a method, the desired return value is not another mock object, but a python object that makes sense for a given test case. One way to set a return value, is to set the `return_value` attribute of a mock object to the desired value.

For example:

```
>>> my_mock.answer.return_value = 42
>>> my_mock.answer()
42
```

2.3.2 Setting side effects

Some other times, when a method is called, an exception is supposed to be raised to simulate an error situation that the code being tested is expected to handle. In that case, the *side_effect* attribute provide the expected behavior:

```
>>> my_mock.error.side_effect = ValueError('Error message')
>>> my_mock.error()
Traceback (most recent call last):
...
ValueError: Error message
```

Additionally, the *side_effect* attribute can be used when different values are expected to be returned for each method call using an iterable:

```
>>> my_mock.get_next.side_effect = [1, 2, 3]
>>> my_mock.get_next()
1
>>> my_mock.get_next()
```

```
2
>>> my_mock.get_next()
3
```

When a more advanced behavior is needed, instead of an iterable, a callable can be used to return whatever is needed. However, this is not commonly used.

2.3.3 Assertions

Given that they are used for testing, mock objects are usually involved in assertions as well. In particular, they are commonly used to make sure that a method from an external dependency was called.

One simple way to do this is just look at the `called` attribute:

```
>>> my_mock.method()
<MagicMock name='mock.method()' id='...'>
>>> my_mock.method.called
True
```

However, that's not usually enough, since we need to figure out not only if a method was called, but also if it was called with the right arguments. In such a case, `assert_called_with` is a great helper method:

```
>>> my_mock.method(1, 2, 3, a=4, b=5, c=6)
<MagicMock name='mock.method()' id='...'>
>>> my_mock.method.assert_called_with(1, 2, 3, a=4, b=5, c=6)
```

Of course, if the method wasn't called with the expected arguments an `AssertionError` will be raised:

```
>>> my_mock.method.assert_called_with('some', 'other', 'arguments')
Traceback (most recent call last):
  ...
AssertionError: Expected call: method('some', 'other', 'arguments')
Actual call: method(1, 2, 3, a=4, c=6, b=5)
```

There are other helper methods that I recommend that can be used and are well described in the documentation. One that is particularly useful is `assert_called_once_with` that works exactly in the same way, but will fail if the method has been called more than once.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search