Python Mock Object Library

Common Pitfalls and Best Practices

Sunaina Pai

PyCon UK 2019, Cardiff City Hall, Cardiff, UK

15 Sep 2019

who am i

Sunaina Pai

Software Developer from Bangalore, India.

Author of `makesite.py`, a simple, lightweight, and magic-free static site/blog generator for Python coders. <https://git.io/makesite>

https://twitter.com/sunainapai

https://github.com/sunainapai

https://sunainapai.in/

Introduction to Mock

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.foo()
<Mock name='mock.foo()' id='4425638744'>
```

Introduction to Mock

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.foo()
<Mock name='mock.foo()' id='4425638744'>
>>> m.foo().bar().baz()
<Mock name='mock.foo().bar().baz()' id='4428396584'>
```

Introduction to Mock

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.foo()
<Mock name='mock.foo()' id='4425638744'>
>>> m.foo().bar().baz()
<Mock name='mock.foo().bar().baz()' id='4428396584'>
>>> m.a.b.c
<Mock name='mock.a.b.c' id='4428394568'>
```

Introduction to Mock

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.foo()
<Mock name='mock.foo()' id='4425638744'>
>>> m.foo().bar().baz()
<Mock name='mock.foo().bar().baz()' id='4428396584'>
>>> m.a.b.c
<Mock name='mock.a.b.c' id='4428394568'>
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'Mock' object is not subscriptable
```

Introduction to Mock

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.foo()
<Mock name='mock.foo()' id='4425638744'>
>>> m.foo().bar().baz()
<Mock name='mock.foo().bar().baz()' id='4428396584'>
>>> m.a.b.c
<Mock name='mock.a.b.c' id='4428394568'>
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'Mock' object is not subscriptable
>>> 'foo' in m
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: argument of type 'Mock' is not iterable
```

Introduction to Mock

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.foo()
<Mock name='mock.foo()' id='4425638744'>
>>> m.foo().bar().baz()
<Mock name='mock.foo().bar().baz()' id='4428396584'>
>>> m.a.b.c
<Mock name='mock.a.b.c' id='4428394568'>
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'Mock' object is not subscriptable
>>> 'foo' in m
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: argument of type 'Mock' is not iterable
```

`Mock` does not implement `__getitem__`, `__contains__`, `__len__`, etc.

Introduction to MagicMock

```
>>> from unittest import mock
>>> m = mock.MagicMock()
>>> m.foo()
<MagicMock name='mock.foo()' id='4428595096'>
```

Introduction to MagicMock

```
>>> from unittest import mock
>>> m = mock.MagicMock()
>>> m.foo()
<MagicMock name='mock.foo()' id='4428595096'>
>>> m.foo().bar().baz()
<MagicMock name='mock.foo().bar().baz()' id='4428661760'>
```

Introduction to MagicMock

```
>>> from unittest import mock
>>> m = mock.MagicMock()
>>> m.foo()
<MagicMock name='mock.foo()' id='4428595096'>
>>> m.foo().bar().baz()
<MagicMock name='mock.foo().bar().baz()' id='4428661760'>
>>> m.a.b.c
<MagicMock name='mock.a.b.c' id='4428707432'>
```

Introduction to MagicMock

```
>>> from unittest import mock
>>> m = mock.MagicMock()
>>> m.foo()
<MagicMock name='mock.foo()' id='4428595096'>
>>> m.foo().bar().baz()
<MagicMock name='mock.foo().bar().baz()' id='4428661760'>
>>> m.a.b.c
<MagicMock name='mock.a.b.c' id='4428707432'>
>>> m[0]
<MagicMock name='mock.__getitem__()' id='4428740704'>
```

Introduction to MagicMock

```
>>> from unittest import mock
>>> m = mock.MagicMock()
>>> m.foo()
<MagicMock name='mock.foo()' id='4428595096'>
>>> m.foo().bar().baz()
<MagicMock name='mock.foo().bar().baz()' id='4428661760'>
>>> m.a.b.c
<MagicMock name='mock.a.b.c' id='4428707432'>
>>> m[0]
<MagicMock name='mock.__getitem__()' id='4428740704'>
>>> 'foo' in m
False
```

Introduction to MagicMock

```
>>> from unittest import mock
>>> m = mock.MagicMock()
>>> m.foo()
<MagicMock name='mock.foo()' id='4428595096'>
>>> m.foo().bar().baz()
<MagicMock name='mock.foo().bar().baz()' id='4428661760'>
>>> m.a.b.c
<MagicMock name='mock.a.b.c' id='4428707432'>
>>> m[0]
<MagicMock name='mock.__getitem__()' id='4428740704'>
>>> 'foo' in m
False
```

`MagicMock` implements most magic methods such as `__getitem__`, `__contains__`, `__len__`, etc.

Introduction to Mock().return_value

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m()
<Mock name='mock()' id='4497061008'>
>>> m.return_value
<Mock name='mock()' id='4497061008'>
```

Introduction to `Mock().return_value`

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m()
<Mock name='mock()' id='4497061008'>
>>> m.return_value
<Mock name='mock()' id='4497061008'>
```

The default return value is a new `Mock` object. It is created the first time the return value is accessed, either explicity, e.g., `m.return_value`, or by calling the mock, e.g., `m()`.

Introduction to MagicMock().return_value

```
>>> from unittest import mock
>>> m = mock.MagicMock()
>>> m()
<MagicMock name='mock()' id='4323535632'>
>>> m.return_value
<MagicMock name='mock()' id='4323535632'>
```

Introduction to MagicMock().return_value

```
>>> from unittest import mock
>>> m = mock.MagicMock()
>>> m()
<MagicMock name='mock()' id='4323535632'>
>>> m.return_value
<MagicMock name='mock()' id='4323535632'>
```

In case of MagicMock, the default return value is a new MagicMock object.

Introduction to mock.patch(): Basics

```
>>> import os.path
>>> os.path.getsize('/etc/hosts')
259
```

Introduction to mock.patch(): Basics

```
>>> import os.path
>>> os.path.getsize('/etc/hosts')
259
>>> from unittest import mock
>>> patcher = mock.patch('os.path.getsize')
```

Introduction to mock.patch(): Basics

```
>>> import os.path
>>> os.path.getsize('/etc/hosts')
259
>>> from unittest import mock
>>> patcher = mock.patch('os.path.getsize')
>>> mock_getsize = patcher.start()
>>> mock_getsize
<MagicMock name='getsize' id='4348922256'>
>>> os.path.getsize
<MagicMock name='getsize' id='4348922256'>
```

Introduction to mock.patch(): Basics

```
>>> import os.path
>>> os.path.getsize('/etc/hosts')
259
>>> from unittest import mock
>>> patcher = mock.patch('os.path.getsize')
>>> mock_getsize = patcher.start()
>>> mock_getsize
<MagicMock name='getsize' id='4348922256'>
>>> os.path.getsize
<MagicMock name='getsize' id='4348922256'>
>>> mock_getsize.return_value = 10
>>> os.path.getsize('/etc/hosts')
10
```

Introduction to mock.patch(): Basics

```
>>> import os.path
>>> os.path.getsize('/etc/hosts')
259
>>> from unittest import mock
>>> patcher = mock.patch('os.path.getsize')
>>> mock_getsize = patcher.start()
>>> mock_getsize
<MagicMock name='getsize' id='4348922256'>
>>> os.path.getsize
<MagicMock name='getsize' id='4348922256'>
>>> mock_getsize.return_value = 10
>>> os.path.getsize('/etc/hosts')
10
>>> patcher.stop()
>>> os.path.getsize('/etc/hosts')
259
```

Introduction to mock.patch(): Basics

```
>>> import os.path
>>> os.path.getsize('/etc/hosts')
259
>>> from unittest import mock
>>> patcher = mock.patch('os.path.getsize')
>>> mock_getsize = patcher.start()
>>> mock_getsize
<MagicMock name='getsize' id='4348922256'>
>>> os.path.getsize
<MagicMock name='getsize' id='4348922256'>
>>> mock_getsize.return_value = 10
>>> os.path.getsize('/etc/hosts')
10
>>> patcher.stop()
>>> os.path.getsize('/etc/hosts')
259
```

`start()` patches the target. `stop()` undoes the patch.

Introduction to mock.patch(): Context Manager

```
>>> import os.path
>>> os.path.getsize('/etc/hosts')
259
```

Introduction to mock.patch(): Context Manager

```
>>> import os.path
>>> os.path.getsize('/etc/hosts')
259
>>> from unittest import mock
>>> with mock.patch('os.path.getsize') as mock_getsize:
...     mock_getsize.return_value = 10
...     os.path.getsize('/etc/hosts')
...
10
```

Introduction to mock.patch(): Context Manager

```
>>> import os.path
>>> os.path.getsize('/etc/hosts')
259
>>> from unittest import mock
>>> with mock.patch('os.path.getsize') as mock_getsize:
...     mock_getsize.return_value = 10
...     os.path.getsize('/etc/hosts')
...
10
>>> os.path.getsize('/etc/hosts')
259
```

Introduction to mock.patch(): Context Manager

```
>>> import os.path
>>> os.path.getsize('/etc/hosts')
259
>>> from unittest import mock
>>> with mock.patch('os.path.getsize') as mock_getsize:
...     mock_getsize.return_value = 10
...     os.path.getsize('/etc/hosts')
...
10
>>> os.path.getsize('/etc/hosts')
259
```

Inside the body of the `with` statement, the target is patched.

When the `with` statement exits, the patch is undone.

Introduction to mock.patch(): Function Decorator

```
>>> import os.path
>>> from unittest import mock
>>> @mock.patch('os.path.getsize')
... def f(mock_getsize):
...     mock_getsize.return_value = 10
...     return os.path.getsize('/etc/hosts')
...
```

Introduction to mock.patch(): Function Decorator

```
>>> import os.path
>>> from unittest import mock
>>> @mock.patch('os.path.getsize')
... def f(mock_getsize):
...     mock_getsize.return_value = 10
...     return os.path.getsize('/etc/hosts')
...
>>> os.path.getsize('/etc/hosts')
259
```

Introduction to mock.patch(): Function Decorator

```
>>> import os.path
>>> from unittest import mock
>>> @mock.patch('os.path.getsize')
... def f(mock_getsize):
...     mock_getsize.return_value = 10
...     return os.path.getsize('/etc/hosts')
...
>>> os.path.getsize('/etc/hosts')
259
>>> f()
10
```

Introduction to mock.patch(): Function Decorator

```
>>> import os.path
>>> from unittest import mock
>>> @mock.patch('os.path.getsize')
... def f(mock_getsize):
...     mock_getsize.return_value = 10
...     return os.path.getsize('/etc/hosts')
...
>>> os.path.getsize('/etc/hosts')
259
>>> f()
10
>>> os.path.getsize('/etc/hosts')
259
```

Introduction to mock.patch(): Function Decorator

```
>>> import os.path
>>> from unittest import mock
>>> @mock.patch('os.path.getsize')
... def f(mock_getsize):
...     mock_getsize.return_value = 10
...     return os.path.getsize('/etc/hosts')
...
>>> os.path.getsize('/etc/hosts')
259
>>> f()
10
>>> os.path.getsize('/etc/hosts')
259
```

Inside the body of the decorated function, the target is patched.

When the function returns, the patch is undone.

Introduction to Mock Assertions

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.foo(10, 20)
<Mock name='mock.foo()' id='4371682128'>
```

Introduction to Mock Assertions

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.foo(10, 20)
<Mock name='mock.foo()' id='4371682128'>
>>> m.assert_called()
>>> m.foo.assert_called()
>>> m.foo.assert_called_once()
>>> m.foo.assert_called_with(10, 20)
>>> m.foo.assert_called_once_with(10, 20)
```

Introduction to Mock Assertions

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.foo(10, 20)
<Mock name='mock.foo()' id='4371682128'>
>>> m.assert_called()
>>> m.foo.assert_called()
>>> m.foo.assert_called_once()
>>> m.foo.assert_called_with(10, 20)
>>> m.foo.assert_called_once_with(10, 20)
>>> m.foo.assert_called_with(10, 30)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "unittest/mock.py", line 834, in assert_called_with
    raise AssertionError(_error_message()) from cause
AssertionError: Expected call: foo(10, 30)
Actual call: foo(10, 20)
```

Introduction to Mock Assertions

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.foo(10, 20)
<Mock name='mock.foo()' id='4371682128'>
>>> m.assert_called()
>>> m.foo.assert_called()
>>> m.foo.assert_called_once()
>>> m.foo.assert_called_with(10, 20)
>>> m.foo.assert_called_once_with(10, 20)
>>> m.foo.assert_called_with(10, 30)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "unittest/mock.py", line 834, in assert_called_with
    raise AssertionError(_error_message()) from cause
AssertionError: Expected call: foo(10, 30)
Actual call: foo(10, 20)
```

An assertion failure leads to `AssertionError`.

Introduction to Autospeccing

```
>>> from unittest import mock
>>> with mock.patch('os.path.getsize') as mock_getsize:
...     mock_getsize()
...     mock_getsize.assrt_called()
...
<MagicMock name='getsize()' id='4373267664'>
<MagicMock name='getsize.assrt_called()' id='4373339664'>
```

Introduction to Autospeccing

```
>>> from unittest import mock
>>> with mock.patch('os.path.getsize') as mock_getsize:
...     mock_getsize()
...     mock_getsize.assrt_called()
...
<MagicMock name='getsize()' id='4373267664'>
<MagicMock name='getsize.assrt_called()' id='4373339664'>
>>> with mock.patch('os.path.getsize', autospec=True) as mock_getsize:
...     mock_getsize()
...
TypeError: missing a required argument: 'filename'
```

Introduction to Autospeccing

```
>>> from unittest import mock
>>> with mock.patch('os.path.getsize') as mock_getsize:
...     mock_getsize()
...     mock_getsize.assrt_called()
...
<MagicMock name='getsize()' id='4373267664'>
<MagicMock name='getsize.assrt_called()' id='4373339664'>
>>> with mock.patch('os.path.getsize', autospec=True) as mock_getsize:
...     mock_getsize()
...
TypeError: missing a required argument: 'filename'
>>> with mock.patch('os.path.getsize', autospec=True) as mock_getsize:
...     mock_getsize.assrt_called()
...
AttributeError: 'function' object has no attribute 'assrt_called'
```

Introduction to Autospeccing

```
>>> from unittest import mock
>>> with mock.patch('os.path.getsize') as mock_getsize:
...     mock_getsize()
...     mock_getsize.assrt_called()
...
<MagicMock name='getsize()' id='4373267664'>
<MagicMock name='getsize.assrt_called()' id='4373339664'>
>>> with mock.patch('os.path.getsize', autospec=True) as mock_getsize:
...     mock_getsize()
...
TypeError: missing a required argument: 'filename'
>>> with mock.patch('os.path.getsize', autospec=True) as mock_getsize:
...     mock_getsize.assrt_called()
...
AttributeError: 'function' object has no attribute 'assrt_called'
```

Autospeccing creates mocks that have the same API as the objects they are replacing.

Introduction to Mock().mock_calls

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.foo().bar(10, 20).baz()
<Mock name='mock.foo().bar().baz()' id='4561016592'>
```

Introduction to Mock().mock_calls

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.foo().bar(10, 20).baz()
<Mock name='mock.foo().bar().baz()' id='4561016592'>
>>> m.mock_calls
[call.foo(), call.foo().bar(10, 20), call.foo().bar().baz()]
```

Introduction to Mock().mock_calls

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.foo().bar(10, 20).baz()
<Mock name='mock.foo().bar().baz()' id='4561016592'>
>>> m.mock_calls
[call.foo(), call.foo().bar(10, 20), call.foo().bar().baz()]
>>> m.mock_calls[1] == mock.call.foo().bar(10, 20)
True
```

Introduction to Mock().mock_calls

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.foo().bar(10, 20).baz()
<Mock name='mock.foo().bar().baz()' id='4561016592'>
>>> m.mock_calls
[call.foo(), call.foo().bar(10, 20), call.foo().bar().baz()]
>>> m.mock_calls[1] == mock.call.foo().bar(10, 20)
True
>>> m.mock_calls == [mock.call.foo(), mock.call.foo().bar(10, 20),
...                  mock.call.foo().bar().baz()]
True
```

Introduction to Mock().mock_calls

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.foo().bar(10, 20).baz()
<Mock name='mock.foo().bar().baz()' id='4561016592'>
>>> m.mock_calls
[call.foo(), call.foo().bar(10, 20), call.foo().bar().baz()]
>>> m.mock_calls[1] == mock.call.foo().bar(10, 20)
True
>>> m.mock_calls == [mock.call.foo(), mock.call.foo().bar(10, 20),
...                  mock.call.foo().bar().baz()]
True
```

`mock_calls` returns a list of call objects.

`mock.call` lets us create call objects.

Pitfall #1: Where to Patch: App Code

```
# app.py

import sys
from os.path import getsize


def get_total_size(filenames):
    total = 0
    for f in filenames:
        total += getsize(f)
    return total


if __name__ == '__main__':
    args = sys.argv[1:]
    print(get_total_size(args))
```

Pitfall #1: Where to Patch: False Start

```
# testapp.py

import unittest
from unittest import mock

import app


class AppTest(unittest.TestCase):

    @mock.patch('os.path.getsize')
    def test_total_size(self, mock_getsize):
        mock_getsize.return_value = 10
        total = app.get_total_size(['', ''])
        self.assertEqual(total, 20)
```

Pitfall #1: Where to Patch: False Start

```python
# app.py

import sys
from os.path import getsize


def get_total_size(filenames):
    total = 0
    for f in filenames:
        total += getsize(f)
    return total


if __name__ == '__main__':
    args = sys.argv[1:]
    print(get_total_size(args))
```

```python
# testapp.py

import unittest
from unittest import mock

import app


class AppTest(unittest.TestCase):

    @mock.patch('os.path.getsize')
    def test_total_size(self, mock_getsize):
        mock_getsize.return_value = 10
        total = app.get_total_size(['', ''])
        self.assertEqual(total, 20)
```

Pitfall #1: Where to Patch: False Start

```python
# app.py

import sys
from os.path import getsize


def get_total_size(filenames):
    total = 0
    for f in filenames:
        total += getsize(f)
    return total


if __name__ == '__main__':
    args = sys.argv[1:]
    print(get_total_size(args))
```

```python
# testapp.py

import unittest
from unittest import mock

import app


class AppTest(unittest.TestCase):

    @mock.patch('os.path.getsize')
    def test_total_size(self, mock_getsize):
        mock_getsize.return_value = 10
        total = app.get_total_size(['', ''])
        self.assertEqual(total, 20)
```

```
FileNotFoundError: [Errno 2] No such file or directory: ''
```

Pitfall #1: Where to Patch: False Start

```python
# app.py

import sys
from os.path import getsize


def get_total_size(filenames):
    total = 0
    for f in filenames:
        total += getsize(f)
    return total


if __name__ == '__main__':
    args = sys.argv[1:]
    print(get_total_size(args))
```

```python
# testapp.py

import unittest
from unittest import mock

import app


class AppTest(unittest.TestCase):

    @mock.patch('os.path.getsize')
    def test_total_size(self, mock_getsize):
        mock_getsize.return_value = 10
        total = app.get_total_size(['', ''])
        self.assertEqual(total, 20)
```

```
FileNotFoundError: [Errno 2] No such file or directory: ''
```

- app is imported first.

Pitfall #1: Where to Patch: False Start

```python
# app.py

import sys
from os.path import getsize


def get_total_size(filenames):
    total = 0
    for f in filenames:
        total += getsize(f)
    return total


if __name__ == '__main__':
    args = sys.argv[1:]
    print(get_total_size(args))
```

```python
# testapp.py

import unittest
from unittest import mock

import app


class AppTest(unittest.TestCase):

    @mock.patch('os.path.getsize')
    def test_total_size(self, mock_getsize):
        mock_getsize.return_value = 10
        total = app.get_total_size(['', ''])
        self.assertEqual(total, 20)
```

```
FileNotFoundError: [Errno 2] No such file or directory: ''
```

- app is imported first.
- getsize is imported into app.

Pitfall #1: Where to Patch: False Start

```python
# app.py

import sys
from os.path import getsize


def get_total_size(filenames):
    total = 0
    for f in filenames:
        total += getsize(f)
    return total


if __name__ == '__main__':
    args = sys.argv[1:]
    print(get_total_size(args))
```

```python
# testapp.py

import unittest
from unittest import mock

import app


class AppTest(unittest.TestCase):

    @mock.patch('os.path.getsize')
    def test_total_size(self, mock_getsize):
        mock_getsize.return_value = 10
        total = app.get_total_size(['', ''])
        self.assertEqual(total, 20)
```

```
FileNotFoundError: [Errno 2] No such file or directory: ''
```

- app is imported first.
- getsize is imported into app.
- getsize in os.path is patched then.

Pitfall #1: Where to Patch: False Start

```python
# app.py

import sys
from os.path import getsize


def get_total_size(filenames):
    total = 0
    for f in filenames:
        total += getsize(f)
    return total


if __name__ == '__main__':
    args = sys.argv[1:]
    print(get_total_size(args))
```

```python
# testapp.py

import unittest
from unittest import mock

import app


class AppTest(unittest.TestCase):

    @mock.patch('os.path.getsize')
    def test_total_size(self, mock_getsize):
        mock_getsize.return_value = 10
        total = app.get_total_size(['', ''])
        self.assertEqual(total, 20)
```

```
FileNotFoundError: [Errno 2] No such file or directory: ''
```

- app is imported first.
- getsize is imported into app.
- getsize in os.path is patched then.
- getsize in app remains unaffected.

Pitfall #1: Where to Patch: Solution

```python
# app.py

import sys
from os.path import getsize


def get_total_size(filenames):
    total = 0
    for f in filenames:
        total += getsize(f)
    return total


if __name__ == '__main__':
    args = sys.argv[1:]
    print(get_total_size(args))
```

```python
# testapp.py

import unittest
from unittest import mock

import app


class AppTest(unittest.TestCase):

    @mock.patch('app.getsize')
    def test_total_size(self, mock_getsize):
        mock_getsize.return_value = 10
        total = app.get_total_size(['', ''])
        self.assertEqual(total, 20)
```

Pitfall #1: Where to Patch: Solution

```python
# app.py

import sys
from os.path import getsize


def get_total_size(filenames):
    total = 0
    for f in filenames:
        total += getsize(f)
    return total


if __name__ == '__main__':
    args = sys.argv[1:]
    print(get_total_size(args))
```

```python
# testapp.py

import unittest
from unittest import mock

import app


class AppTest(unittest.TestCase):

    @mock.patch('app.getsize')
    def test_total_size(self, mock_getsize):
        mock_getsize.return_value = 10
        total = app.get_total_size(['', ''])
        self.assertEqual(total, 20)
```

- app is imported first.

Pitfall #1: Where to Patch: Solution

```python
# app.py

import sys
from os.path import getsize


def get_total_size(filenames):
    total = 0
    for f in filenames:
        total += getsize(f)
    return total


if __name__ == '__main__':
    args = sys.argv[1:]
    print(get_total_size(args))
```

```python
# testapp.py

import unittest
from unittest import mock

import app


class AppTest(unittest.TestCase):

    @mock.patch('app.getsize')
    def test_total_size(self, mock_getsize):
        mock_getsize.return_value = 10
        total = app.get_total_size(['', ''])
        self.assertEqual(total, 20)
```

- app is imported first.
- getsize is imported into app.

Pitfall #1: Where to Patch: Solution

```python
# app.py

import sys
from os.path import getsize


def get_total_size(filenames):
    total = 0
    for f in filenames:
        total += getsize(f)
    return total


if __name__ == '__main__':
    args = sys.argv[1:]
    print(get_total_size(args))
```

```python
# testapp.py

import unittest
from unittest import mock

import app


class AppTest(unittest.TestCase):

    @mock.patch('app.getsize')
    def test_total_size(self, mock_getsize):
        mock_getsize.return_value = 10
        total = app.get_total_size(['', ''])
        self.assertEqual(total, 20)
```

- app is imported first.
- getsize is imported into app.
- getsize in app is patched then.

Pitfall #1: Where to Patch: Solution

```python
# app.py

import sys
from os.path import getsize


def get_total_size(filenames):
    total = 0
    for f in filenames:
        total += getsize(f)
    return total


if __name__ == '__main__':
    args = sys.argv[1:]
    print(get_total_size(args))
```

```python
# testapp.py

import unittest
from unittest import mock

import app


class AppTest(unittest.TestCase):

    @mock.patch('app.getsize')
    def test_total_size(self, mock_getsize):
        mock_getsize.return_value = 10
        total = app.get_total_size(['', ''])
        self.assertEqual(total, 20)
```

- app is imported first.
- getsize is imported into app.
- getsize in app is patched then.
- getsize in app is a mock now.

Pitfall #1: Where to Patch: Solution

```python
# app.py

import sys
from os.path import getsize


def get_total_size(filenames):
    total = 0
    for f in filenames:
        total += getsize(f)
    return total


if __name__ == '__main__':
    args = sys.argv[1:]
    print(get_total_size(args))
```

```python
# testapp.py

import unittest
from unittest import mock

import app


class AppTest(unittest.TestCase):

    @mock.patch('app.getsize')
    def test_total_size(self, mock_getsize):
        mock_getsize.return_value = 10
        total = app.get_total_size(['', ''])
        self.assertEqual(total, 20)
```

```
test_get_total (testfoo.FooTest) ... ok
```

- app is imported first.
- getsize is imported into app.
- getsize in app is patched then.
- getsize in app is a mock now.

Pitfall #2: Mocking Chained Calls: App Code

```python
# app.py

import sys

import requests


def get_stars(owner, repo):
    url = f'https://api.github.com/repos/{owner}/{repo}'
    stars = requests.get(url).json()['stargazers_count']
    return stars


if __name__ == '__main__':
    print(get_stars(sys.argv[1], sys.argv[2]))
```

Pitfall #2: Mocking Chained Calls: Ugly Test Code

```
# testapp.py

import unittest
from unittest import mock

import app


class AppTest(unittest.TestCase):

    @mock.patch('requests.get')
    def test_get_stars(self, mock_get):
        mock_response = mock.Mock()
        mock_response.json.return_value = {'stargazers_count': 5}
        mock_get.return_value = mock_response
        stars = app.get_stars('', '')
        self.assertEqual(stars, 5)
```

Pitfall #2: Mocking Chained Calls: Ugly Test Code

```
# app.py

def get_stars(owner, repo):
    url = f'https://api.github.com/repos/{owner}/{repo}'
    stars = requests.get(url).json()['stargazers_count']
    return stars
```

```
# testapp.py

class AppTest(unittest.TestCase):

    @mock.patch('requests.get')
    def test_get_stars(self, mock_get):
        mock_response = mock.Mock()
        mock_response.json.return_value = {'stargazers_count': 5}
        mock_get.return_value = mock_response
        stars = app.get_stars('', '')
        self.assertEqual(stars, 5)
```

Pitfall #2: Mocking Chained Calls: Ugly Test Code

```
# app.py

def get_stars(owner, repo):
    url = f'https://api.github.com/repos/{owner}/{repo}'
    stars = requests.get(url).json()['stargazers_count']
    return stars
```

```
# testapp.py

class AppTest(unittest.TestCase):

    @mock.patch('requests.get')
    def test_get_stars(self, mock_get):
        mock_response = mock.Mock()
        mock_response.json.return_value = {'stargazers_count': 5}
        mock_get.return_value = mock_response
        stars = app.get_stars('', '')
        self.assertEqual(stars, 5)
```

- We don't need to create a new mock to assign to `mock_get.return_value`.
- `mock_get.return_value` gives us a `MagicMock` object. We can use that.

Pitfall #2: Mocking Chained Calls: Good Test Code

```python
# app.py

def get_stars(owner, repo):
    url = f'https://api.github.com/repos/{owner}/{repo}'
    stars = requests.get(url).json()['stargazers_count']
    return stars
```

```python
# testapp.py

class AppTest(unittest.TestCase):

    @mock.patch('requests.get')
    def test_get_stars_good(self, mock_get):
        mock_get.return_value.json.return_value = {'stargazers_count': 5}
        stars = app.get_stars('', '')
        self.assertEqual(stars, 5)
```

Pitfall #2: Mocking Chained Calls: Good Test Code

```
# app.py

def get_stars(owner, repo):
    url = f'https://api.github.com/repos/{owner}/{repo}'
    stars = requests.get(url).json()['stargazers_count']
    return stars
```

```
# testapp.py

class AppTest(unittest.TestCase):

    @mock.patch('requests.get')
    def test_get_stars_good(self, mock_get):
        mock_get.return_value.json.return_value = {'stargazers_count': 5}
        stars = app.get_stars('', '')
        self.assertEqual(stars, 5)
```

- Accessing `mock_get.return_value` gives us a `MagicMock` object.
- We access its `json` attribute to get another `MagicMock` object.
- We then set the `return_value` of this new `MagicMock` object.

Pitfall #3: Redundant `return_value`

```python
# app.py

def get_stars(owner, repo):
    url = f'https://api.github.com/repos/{owner}/{repo}'
    stars = requests.get(url).json()['stargazers_count']
    return stars
```

```python
# testapp.py

class AppTest(unittest.TestCase):

    @mock.patch('requests.get')
    def test_get_stars(self, mock_get):
        owner, repo = 'foo', 'bar'
        url = f'https://api.github.com/repos/{owner}/{repo}'
        mock_get.return_value.json.return_value = {'stargazers_count': 5}
        app.get_stars(owner, repo)
        mock_get.assert_called_once_with(url)
```

Pitfall #3: Redundant `return_value`

```
# app.py

def get_stars(owner, repo):
    url = f'https://api.github.com/repos/{owner}/{repo}'
    stars = requests.get(url).json()['stargazers_count']
    return stars
```

```
# testapp.py

class AppTest(unittest.TestCase):

    @mock.patch('requests.get')
    def test_get_stars_good(self, mock_get):
        owner, repo = 'foo', 'bar'
        url = f'https://api.github.com/repos/{owner}/{repo}'
        mock_get.return_value.json.return_value = {'stargazers_count': 5}
        app.get_stars(owner, repo)
        mock_get.assert_called_once_with(url)
```

- `mock_get.return_value.json.return_value` is a `MagicMock` object too.
- It is subscriptable. There is no need to set it to a `dict` value.

Pitfall #4: Perplexing Build Errors: App Code

```
# app.py

import sys

import requests


def get_stars(owner, repo):
    url = 'https://api.github.com/repos/' + owner + '/' + repo
    stars = requests.get(url).json()['stargazers_count']
    return stars


if __name__ == '__main__':
    print(get_stars(sys.argv[1], sys.argv[2]))
```

Pitfall #4: Perplexing Build Errors: Test Code

```
# testapp.py

import unittest
from unittest import mock

import app


class AppTest(unittest.TestCase):

    @mock.patch('requests.get')
    def test_get_stars(self, mock_get):
        app.get_stars('', '')
        mock_get.assert_called()
```

Pitfall #4: Perplexing Build Errors: Build Config

```yaml
# .travis.yml

language: python

python:
  - "3.4"
  - "3.5"
  - "3.6"
  - "3.7"

install:
  - pip3 install requests

script:
  - python3 -m unittest -v
```

Pitfall #4: Perplexing Build Errors: `AttributeError`

✓	# 1.1	Python: 3.4	no environment variables set	27 sec	
✗	# 1.2	Python: 3.5	no environment variables set	29 sec	
✓	# 1.3	Python: 3.6	no environment variables set	15 sec	
✓	# 1.4	Python: 3.7	no environment variables set	14 sec	

Python 3.4: `test_get_stars (testapp.FooTest) ... ok`

Python 3.5: `AttributeError: assert_called`

Python 3.6: `test_get_stars (testapp.FooTest) ... ok`

Python 3.7: `test_get_stars (testapp.FooTest) ... ok`

Pitfall #4: Perplexing Build Errors: Missing Method

✓ # 1.1	Python: 3.4	no environment variables set	27 sec	
✗ # 1.2	Python: 3.5	no environment variables set	29 sec	
✓ # 1.3	Python: 3.6	no environment variables set	15 sec	
✓ # 1.4	Python: 3.7	no environment variables set	14 sec	

The `assert_called()` method is new in Python 3.6.

---

assert_called()

Assert that the mock was called at least once.

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

New in version 3.6.

See https://docs.python.org/3.7/library/unittest.mock.html#unittest.mock.Mock.assert_called

Pitfall #4: Perplexing Build Errors: CPython Commit

`cpython/Lib/unittest/mock.py > class NonCallableMock(Base):`

```
      def __getattr__(self, name):
-         if name == '_mock_methods':
+         if name in {'_mock_methods', '_mock_unsafe'}:
              raise AttributeError(name)
          elif self._mock_methods is not None:
              if name not in self._mock_methods or name in _all_magics:
                  raise AttributeError("Mock object has no attribute %r" % name)
          elif _is_magic(name):
              raise AttributeError(name)
+         if not self._mock_unsafe:
+             if name.startswith(('assert', 'assret')):
+                 raise AttributeError(name)
```

See https://github.com/python/cpython/commit/8c14534

This change in Python 3.5 prevents tests from silently passing when we think we have called an assertion method but it does not exist.

Pitfall #4: Perplexing Build Errors: `Mock` Documentation

The `assert_called_once()` method is also new in Python 3.6.

It could also lead to perplexing build errors.

`assert_called_once()`

Assert that the mock was called exactly once.

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
Traceback (most recent call last):
...
AssertionError: Expected 'method' to have been called
```

New in version 3.6.

See https://docs.python.org/3.7/library/unittest.mock.html#unittest.mock.Mock.assert_called_once

Pitfall #4: Perplexing Build Errors: Autospeccing

```
# testapp.py

import unittest
from unittest import mock

import app


class AppTest(unittest.TestCase):

    @mock.patch('requests.get', autospec=True)
    def test_get_stars(self, mock_get):
        app.get_stars('', '')
        mock_get.assert_called()
```

The `assert_called()` method no longer works silently in Python versions 3.4 and 3.5 due to autospeccing.

Pitfall #4: Perplexing Build Errors: Autospeccing

✗	# 2.1	Python: 3.4	no environment variables set	23 sec	
✗	# 2.2	Python: 3.5	no environment variables set	26 sec	
✓	# 2.3	Python: 3.6	no environment variables set	16 sec	
✓	# 2.4	Python: 3.7	no environment variables set	14 sec	

Python 3.4: `AttributeError: 'function' object has no attribute 'assert_called'`

Python 3.5: `AttributeError: 'function' object has no attribute 'assert_called'`

Python 3.6: `test_get_stars (testapp.FooTest) ... ok`

Python 3.7: `test_get_stars (testapp.FooTest) ... ok`

Pitfall #4: Perplexing Build Errors: Solution

```
# testapp.py

import unittest
from unittest import mock

import app


class AppTest(unittest.TestCase):

    @mock.patch('requests.get')
    def test_get_stars(self, mock_get):
        owner, repo = 'foo', 'bar'
        url = 'https://api.github.com/repos/' + owner + '/' + repo
        app.get_stars(owner, repo)
        mock_get.assert_called_once_with(url)
```

The `assert_called_once_with()` method is available since Python 3.3.

Pitfall #5: Asserting Chained Calls: App Code

```python
# app.py

import sys

import requests


def get_stars(owner, repo):
    url = f'https://api.github.com/repos/{owner}/{repo}'
    stars = requests.get(url).json().get('stargazers_count')
    return stars


if __name__ == '__main__':
    print(get_stars(sys.argv[1], sys.argv[2]))
```

Pitfall #5: Asserting Chained Calls: Unwieldy Test Code

```
# testapp.py

import unittest
from unittest import mock

import app


class AppTest(unittest.TestCase):

    @mock.patch('requests.get')
    def test_get_stars(self, mock_get):
        owner, repo = 'foo', 'bar'
        url = f'https://api.github.com/repos/{owner}/{repo}'
        key = 'stargazers_count'
        app.get_stars(owner, repo)
        expected = [mock.call(url), mock.call().json(),
                    mock.call().json().get(key)]
        self.assertEqual(mock_get.mock_calls, expected)
```

Pitfall #5: Asserting Chained Calls: Unwieldy Test Code

```
# app.py

def get_stars(owner, repo):
    url = f'https://api.github.com/repos/{owner}/{repo}'
    stars = requests.get(url).json().get('stargazers_count')
    return stars
```

```
# testapp.py

class AppTest(unittest.TestCase):

    @mock.patch('requests.get')
    def test_get_stars(self, mock_get):
        owner, repo = 'foo', 'bar'
        url = f'https://api.github.com/repos/{owner}/{repo}'
        key = 'stargazers_count'
        app.get_stars(owner, repo)
        expected = [mock.call(url), mock.call().json(),
                    mock.call().json().get(key)]
        self.assertEqual(mock_get.mock_calls, expected)
```

Pitfall #5: Asserting Chained Calls: Unwieldy Test Code

```
# app.py

def get_stars(owner, repo):
    url = f'https://api.github.com/repos/{owner}/{repo}'
    stars = requests.get(url).json().get('stargazers_count')
    return stars
```

```
# testapp.py

class AppTest(unittest.TestCase):

    @mock.patch('requests.get')
    def test_get_stars(self, mock_get):
        owner, repo = 'foo', 'bar'
        url = f'https://api.github.com/repos/{owner}/{repo}'
        key = 'stargazers_count'
        app.get_stars(owner, repo)
        expected = [mock.call(url), mock.call().json(),
                    mock.call().json().get(key)]
        self.assertEqual(mock_get.mock_calls, expected)
```

We don't need to create the complete list of call objects.

Pitfall #5: Asserting Chained Calls: Good Test Code

```
# app.py

def get_stars(owner, repo):
    url = f'https://api.github.com/repos/{owner}/{repo}'
    stars = requests.get(url).json().get('stargazers_count')
    return stars
```

```
# testapp.py

class AppTest(unittest.TestCase):

    @mock.patch('requests.get')
    def test_get_stars(self, mock_get):
        owner, repo = 'foo', 'bar'
        url = f'https://api.github.com/repos/{owner}/{repo}'
        key = 'stargazers_count'
        app.get_stars(owner, repo)
        expected = mock.call(url).json().get(key).call_list()
        self.assertEqual(mock_get.mock_calls, expected)
```

Pitfall #5: Asserting Chained Calls: Good Test Code

```
# app.py

def get_stars(owner, repo):
    url = f'https://api.github.com/repos/{owner}/{repo}'
    stars = requests.get(url).json().get('stargazers_count')
    return stars
```

```
# testapp.py

class AppTest(unittest.TestCase):

    @mock.patch('requests.get')
    def test_get_stars(self, mock_get):
        owner, repo = 'foo', 'bar'
        url = f'https://api.github.com/repos/{owner}/{repo}'
        key = 'stargazers_count'
        app.get_stars(owner, repo)
        expected = mock.call(url).json().get(key).call_list()
        self.assertEqual(mock_get.mock_calls, expected)
```

A call object's `call_list()` method can create the complete list of call objects for us.

Summary

#1

Patch where an object is looked up, which is not necessarily the same place as where it is defined.

Summary

#1

Patch where an object is looked up, which is not necessarily the same place as where it is defined.

#2

Do not create a new mock to assign to a mock's `return_value`. A mock's `return_value` is already a mock by default. Just use it and configure it.

Summary

#1

Patch where an object is looked up, which is not necessarily the same place as where it is defined.

#2

Do not create a new mock to assign to a mock's `return_value`. A mock's `return_value` is already a mock by default. Just use it and configure it.

#3

Do not set a `tuple`, `list`, or `dict` as the `return_value` of a `MagicMock` object only to make it subscriptable. A `MagicMock` object is already subscriptable.

Summary

#4

Do not use `assert_called()` or `assert_called_once()` if you are still supporting Python 3.4. Use `assert_called_once_with()` instead.

Summary

#4

Do not use `assert_called()` or `assert_called_once()` if you are still supporting Python 3.4. Use `assert_called_once_with()` instead.

#5

Do not create a long list of `mock.call` objects to mimic `mock_calls` of a mock's chained calls. Use `call_list()` to generate the list for you.

Thank You