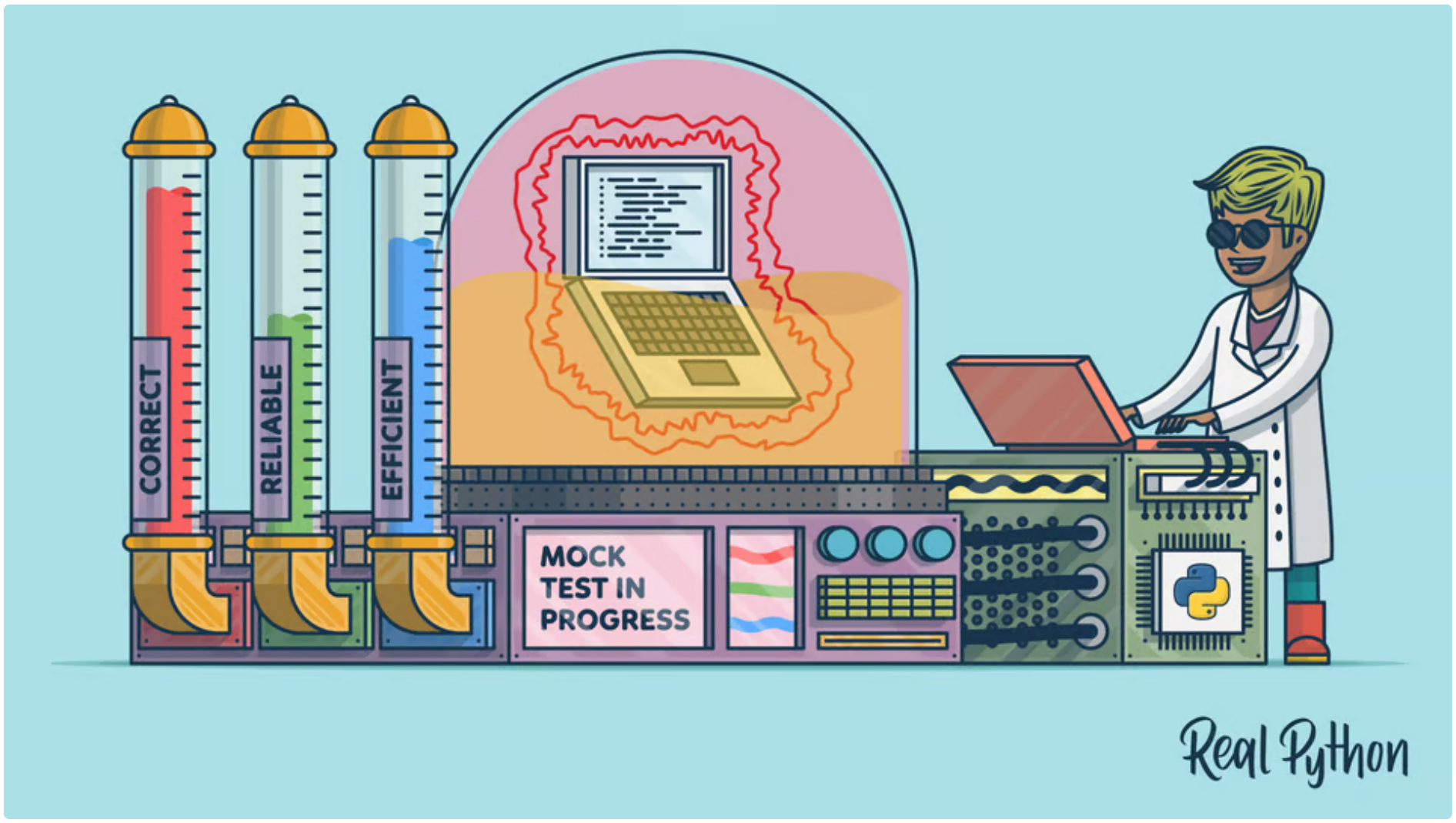

Understanding the Python Mock Object Library

by Alex Ronquillo Mar 13, 2019 19 Comments intermediate testing

Mark as Completed

Share Share Email

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Improve Your Tests With the Python Mock Object Library**

When you're writing robust code, tests are essential for verifying that your application logic is correct, reliable, and efficient. However, the value of your tests depends on how well they demonstrate these criteria. Obstacles such as complex logic and unpredictable dependencies make writing valuable tests difficult. The Python mock object library, `unittest.mock`, can help you overcome these obstacles.

By the end of this article, you'll be able to:

- Create Python mock objects using `Mock`
- Assert you're using objects as you intended
- Inspect usage data stored on your Python mocks
- Configure certain aspects of your Python mock objects
- Substitute your mocks for real objects using `patch()`
- Avoid common problems inherent in Python mocking

You'll begin by seeing what mocking is and how it will improve your tests.

What Is Mocking?

A mock object substitutes and imitates a real object within a testing environment. It is a versatile and powerful tool for improving the quality of your tests.

One reason to use Python mock objects is to control your code's behavior during testing.

For example, if your code makes HTTP requests to external services, then your tests execute predictably only so far as the services are behaving as you expected. Sometimes, a temporary change in the behavior of these external services can cause intermittent failures within your test suite.

Because of this, it would be better for you to test your code in a controlled environment. Replacing the actual request with a mock object would allow you to simulate external service outages and successful responses in a predictable way.

Sometimes, it is difficult to test certain areas of your codebase. Such areas include `except` blocks and `if` statements that are hard to satisfy. Using Python mock objects can help you control the execution path of your code to reach these areas and improve your code coverage.

Another reason to use mock objects is to better understand how you're using their real counterparts in your code. A Python mock object contains data about its usage that you can inspect such as:

- If you called a method
- How you called the method
- How often you called the method

Understanding what a mock object does is the first step to learning how to use one.

Now, you'll see how to use Python mock objects.

The Python Mock Library

The Python mock object library is `unittest.mock`. It provides an easy way to introduce mocks into your tests.

Note: The standard library includes `unittest.mock` in Python 3.3 and later. If you're using an older version of Python, you'll need to install the official backport of the library. To do so, install `mock` from PyPI:

Shell

```
$ pip install mock
```

`unittest.mock` provides a class called `Mock` which you will use to imitate real objects in your codebase. `Mock` offers incredible flexibility and insightful data. This, along with its subclasses, will meet most Python mocking needs that you will face in your tests.

The library also provides a function, called `patch()`, which replaces the real objects in your code with `Mock` instances. You can use `patch()` as either a decorator or a context manager, giving you control over the scope in which the object will be mocked. Once the designated scope exits, `patch()` will clean up your code by replacing the mocked objects with their original counterparts.

Finally, `unittest.mock` provides solutions for some of the issues inherent in mocking objects.

Now, you have a better understanding of what mocking is and the library you'll be using to do it. Let's dive in and explore what features and functionalities `unittest.mock` offers.

The `Mock` Object

`unittest.mock` offers a base class for mocking objects called `Mock`. The use cases for `Mock` are practically limitless because `Mock` is so flexible.

Begin by instantiating a new `Mock` instance:

Python

```
>>> from unittest.mock import Mock
>>> mock = Mock()
>>> mock
<Mock id='4561344720'>
```

Now, you are able to substitute an object in your code with your new `Mock`. You can do this by passing it as an argument to a function or by redefining another object:

Python

```
# Pass mock as an argument to do_something()
do_something(mock)

# Patch the json library
json = mock
```

When you substitute an object in your code, the `Mock` must look like the real object it is replacing. Otherwise, your code will not be able to use the `Mock` in place of the original object.

For example, if you are mocking the `json` library and your program calls `dumps()`, then your Python mock object must also contain `dumps()`.

Next, you'll see how `Mock` deals with this challenge.

Lazy Attributes and Methods

A `Mock` must simulate any object that it replaces. To achieve such flexibility, it creates its attributes when you access them:

Python

```
>>> mock.some_attribute
<Mock name='mock.some_attribute' id='4394778696'>
>>> mock.do_something()
<Mock name='mock.do_something()' id='4394778920'>
```

Since `Mock` can create arbitrary attributes on the fly, it is suitable to replace any object.

Using an example from earlier, if you're mocking the `json` library and you call `dumps()`, the Python mock object will create the method so that its interface can match the library's interface:

Python

```
>>> json = Mock()
>>> json.dumps()
<Mock name='mock.dumps()' id='4392249776'>
```

Notice two key characteristics of this mocked version of `dumps()`:

1. Unlike the real `dumps()`, this mocked method requires no arguments. In fact, it will accept any arguments that you pass to it.

2. The return value of `dumps()` is also a `Mock`. The capability of `Mock` to recursively define other mocks allows for you to use mocks in complex situations:

Python

```
>>> json = Mock()
>>> json.loads('{"k": "v"}').get('k')
<Mock name='mock.loads().get()' id='4379599424'>
```

Because the return value of each mocked method is also a `Mock`, you can use your mocks in a multitude of ways.

Mocks are flexible, but they're also informative. Next, you'll learn how you can use mocks to understand your code better.

Assertions and Inspection

`Mock` instances store data on how you used them. For instance, you can see if you called a method, how you called the method, and so on. There are two main ways to use this information.

First, you can assert that your program used an object as you expected:

Python

```
>>> from unittest.mock import Mock

>>> # Create a mock object
... json = Mock()

>>> json.loads('{"key": "value"}')
<Mock name='mock.loads()' id='4550144184'>

>>> # You know that you called loads() so you can
>>> # make assertions to test that expectation
... json.loads.assert_called()
>>> json.loads.assert_called_once()
>>> json.loads.assert_called_with('{"key": "value"}')
>>> json.loads.assert_called_once_with('{"key": "value"}')

>>> json.loads('{"key": "value"}')
<Mock name='mock.loads()' id='4550144184'>

>>> # If an assertion fails, the mock will raise an AssertionError
... json.loads.assert_called_once()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/Cellar/python/3.6.5/Frameworks/Python.framework/Versions/3.6/lib/python3.6/unittest/mock.py", line 79
    raise AssertionError(msg)
AssertionError: Expected 'loads' to have been called once. Called 2 times.

>>> json.loads.assert_called_once_with('{"key": "value"}')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/Cellar/python/3.6.5/Frameworks/Python.framework/Versions/3.6/lib/python3.6/unittest/mock.py", line 82
    raise AssertionError(msg)
AssertionError: Expected 'loads' to be called once. Called 2 times.

>>> json.loads.assert_not_called()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/Cellar/python/3.6.5/Frameworks/Python.framework/Versions/3.6/lib/python3.6/unittest/mock.py", line 77
    raise AssertionError(msg)
AssertionError: Expected 'loads' to not have been called. Called 2 times.
```

`.assert_called()` ensures you called the mocked method while `.assert_called_once()` checks that you called the method exactly one time.

Both assertion functions have variants that let you inspect the arguments passed to the mocked method:

- `.assert_called_with(*args, **kwargs)`
- `.assert_called_once_with(*args, **kwargs)`

To pass these assertions, you must call the mocked method with the same arguments that you pass to the actual method:

Python

```
>>> json = Mock()
>>> json.loads(s='{"key": "value"}')
>>> json.loads.assert_called_with('{"key": "value"}')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/Cellar/python/3.6.5/Frameworks/Python.framework/Versions/3.6/lib/python3.6/unittest/mock.py", line 81
    raise AssertionError(_error_message()) from cause
AssertionError: Expected call: loads('{"key": "value"}')
Actual call: loads(s='{"key": "value"}')
>>> json.loads.assert_called_with(s='{"key": "value"}')
```

`json.loads.assert_called_with('{"key": "value"}')` raised an `AssertionError` because it expected you to call `loads()` with a positional argument, but you actually called it with a keyword argument. `json.loads.assert_called_with(s='{"key": "value"}')` gets this assertion correct.

Second, you can view special attributes to understand how your application used an object:

Python

```
>>> from unittest.mock import Mock

>>> # Create a mock object
... json = Mock()
>>> json.loads('{"key": "value"}')
<Mock name='mock.loads()' id='4391026640'>

>>> # Number of times you called loads():
... json.loads.call_count
1
>>> # The last loads() call:
... json.loads.call_args
call('{"key": "value"}')
>>> # List of loads() calls:
... json.loads.call_args_list
[call('{"key": "value"}')]
>>> # List of calls to json's methods (recursively):
... json.method_calls
[call.loads('{"key": "value"}')]
```

You can write tests using these attributes to make sure that your objects behave as you intended.

Now, you can create mocks and inspect their usage data. Next, you'll see how to customize mocked methods so that they become more useful in your testing environment.

Managing a Mock's Return Value

One reason to use mocks is to control your code's behavior during tests. One way to do this is to specify a function's return value. Let's use an example to see how this works.

First, create a file called `my_calendar.py`. Add `is_weekday()`, a function that uses Python's `datetime` library to determine whether or not today is a week day. Finally, write a test that asserts that the function works as expected:

Python

```
from datetime import datetime

def is_weekday():
    today = datetime.today()
    # Python's datetime library treats Monday as 0 and Sunday as 6
    return (0 <= today.weekday() < 5)

# Test if today is a weekday
assert is_weekday()
```

Since you're testing if today is a weekday, the result depends on the day you run your test:

Shell

```
$ python my_calendar.py
```

If this command produces no output, the assertion was successful. Unfortunately, if you run the command on a weekend, you'll get an `AssertionError`:

Shell

```
$ python my_calendar.py
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    assert is_weekday()
AssertionError
```

When writing tests, it is important to ensure that the results are predictable. You can use `Mock` to eliminate uncertainty from your code during testing. In this case, you can mock `datetime` and set the `.return_value` for `.today()` to a day that you choose:

Python

```
import datetime
from unittest.mock import Mock

# Save a couple of test days
tuesday = datetime.datetime(year=2019, month=1, day=1)
saturday = datetime.datetime(year=2019, month=1, day=5)

# Mock datetime to control today's date
datetime = Mock()

def is_weekday():
    today = datetime.datetime.today()
    # Python's datetime library treats Monday as 0 and Sunday as 6
    return (0 <= today.weekday() < 5)

# Mock .today() to return Tuesday
datetime.datetime.today.return_value = tuesday
# Test Tuesday is a weekday
assert is_weekday()
# Mock .today() to return Saturday
datetime.datetime.today.return_value = saturday
# Test Saturday is not a weekday
assert not is_weekday()
```

In the example, `.today()` is a mocked method. You've removed the inconsistency by assigning a specific day to the mock's `.return_value`. That way, when you call `.today()`, it returns the `datetime` that you specified.

In the first test, you ensure `tuesday` is a weekday. In the second test, you verify that `saturday` is not a weekday. Now, it doesn't matter what day you run your tests on because you've mocked `datetime` and have control over the object's behavior.

> **Further Reading:** Though mocking `datetime` like this is a good practice example for using `Mock`, a fantastic library already exists for mocking `datetime` called `freezegun`.

When building your tests, you will likely come across cases where mocking a function's return value will not be enough. This is because functions are often more complicated than a simple one-way flow of logic.

Sometimes, you'll want to make functions return different values when you call them more than once or even raise exceptions. You can do this using `.side_effect`.

Managing a Mock's Side Effects

You can control your code's behavior by specifying a mocked function's side effects. A `.side_effect` defines what happens when you call the mocked function.

To test how this works, add a new function to `my_calendar.py`:

Python

```
import requests

def get_holidays():
    r = requests.get('http://localhost/api/holidays')
    if r.status_code == 200:
        return r.json()
    return None
```

`get_holidays()` makes a request to the `localhost` server for a set of holidays. If the server responds successfully, `get_holidays()` will return a dictionary. Otherwise, the method will return `None`.

You can test how `get_holidays()` will respond to a connection timeout by setting `requests.get.side_effect`.

For this example, you'll only see the relevant code from `my_calendar.py`. You'll build a test case using Python's `unittest` library:

Python

```
import unittest
from requests.exceptions import Timeout
from unittest.mock import Mock

# Mock requests to control its behavior
requests = Mock()

def get_holidays():
    r = requests.get('http://localhost/api/holidays')
    if r.status_code == 200:
        return r.json()
    return None

class TestCalendar(unittest.TestCase):
    def test_get_holidays_timeout(self):
        # Test a connection timeout
        requests.get.side_effect = Timeout
        with self.assertRaises(Timeout):
            get_holidays()

if __name__ == '__main__':
    unittest.main()
```

You use `.assertRaises()` to verify that `get_holidays()` raises an exception given the new side effect of `get()`.

Run this test to see the result of your test:

Shell

```
$ python my_calendar.py
.
----------------------------------------------------------
Ran 1 test in 0.000s

OK
```

If you want to be a little more dynamic, you can set `.side_effect` to a function that `Mock` will invoke when you call your mocked method. The mock shares the arguments and return value of the `.side_effect` function:

Python

```
import requests
import unittest
from unittest.mock import Mock

# Mock requests to control its behavior
requests = Mock()

def get_holidays():
    r = requests.get('http://localhost/api/holidays')
    if r.status_code == 200:
        return r.json()
    return None

class TestCalendar(unittest.TestCase):
    def log_request(self, url):
        # Log a fake request for test output purposes
        print(f'Making a request to {url}.')
        print('Request received!')

        # Create a new Mock to imitate a Response
        response_mock = Mock()
        response_mock.status_code = 200
        response_mock.json.return_value = {
            '12/25': 'Christmas',
            '7/4': 'Independence Day',
        }
        return response_mock

    def test_get_holidays_logging(self):
        # Test a successful, logged request
        requests.get.side_effect = self.log_request
        assert get_holidays()['12/25'] == 'Christmas'

if __name__ == '__main__':
    unittest.main()
```

First, you created `.log_request()`, which takes a URL, logs some output using `print()`, then returns a `Mock` response. Next, you set the `.side_effect` of `get()` to `.log_request()`, which you'll use when you call `get_holidays()`. When you run your test, you'll see that `get()` forwards its arguments to `.log_request()` then accepts the return value and returns it as well:

Shell

```
$ python my_calendar.py
Making a request to http://localhost/api/holidays.
Request received!
.
----------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Great! The `print()` statements logged the correct values. Also, `get_holidays()` returned the holidays dictionary.

`.side_effect` can also be an iterable. The iterable must consist of return values, exceptions, or a mixture of both. The iterable will produce its next value every time you call your mocked method. For example, you can test that a retry after a `Timeout` returns a successful response:

Python

```
import unittest
from requests.exceptions import Timeout
from unittest.mock import Mock

# Mock requests to control its behavior
requests = Mock()

def get_holidays():
    r = requests.get('http://localhost/api/holidays')
    if r.status_code == 200:
        return r.json()
    return None

class TestCalendar(unittest.TestCase):
    def test_get_holidays_retry(self):
        # Create a new Mock to imitate a Response
        response_mock = Mock()
        response_mock.status_code = 200
        response_mock.json.return_value = {
            '12/25': 'Christmas',
            '7/4': 'Independence Day',
        }
        # Set the side effect of .get()
        requests.get.side_effect = [Timeout, response_mock]
        # Test that the first request raises a Timeout
        with self.assertRaises(Timeout):
            get_holidays()
        # Now retry, expecting a successful response
        assert get_holidays()['12/25'] == 'Christmas'
        # Finally, assert .get() was called twice
        assert requests.get.call_count == 2

if __name__ == '__main__':
    unittest.main()
```

The first time you call `get_holidays()`, `get()` raises a `Timeout`. The second time, the method returns a valid holidays dictionary. These side effects match the order they appear in the list passed to `.side_effect`.

You can set `.return_value` and `.side_effect` on a `Mock` directly. However, because a Python mock object needs to be flexible in creating its attributes, there is a better way to configure these and other settings.

Configuring Your Mock

You can configure a `Mock` to set up some of the object's behaviors. Some configurable members include `.side_effect`, `.return_value`, and `.name`. You configure a `Mock` when you create one or when you use `.configure_mock()`.

You can configure a `Mock` by specifying certain attributes when you initialize an object:

Python

```
>>> mock = Mock(side_effect=Exception)
>>> mock()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/Cellar/python/3.6.5/Frameworks/Python.framework/Versions/3.6/lib/python3.6/unittest/mock.py", line 93
    return _mock_self._mock_call(*args, **kwargs)
  File "/usr/local/Cellar/python/3.6.5/Frameworks/Python.framework/Versions/3.6/lib/python3.6/unittest/mock.py", line 99
    raise effect
Exception

>>> mock = Mock(name='Real Python Mock')
>>> mock
<Mock name='Real Python Mock' id='4434041432'>

>>> mock = Mock(return_value=True)
>>> mock()
True
```

While `.side_effect` and `.return_value` can be set on the `Mock` instance, itself, other attributes like `.name` can only be set through `.__init__()` or `.configure_mock()`. If you try to set the `.name` of the `Mock` on the instance, you will get a different result:

Python

```
>>> mock = Mock(name='Real Python Mock')
>>> mock.name
<Mock name='Real Python Mock.name' id='4434041544'>

>>> mock = Mock()
>>> mock.name = 'Real Python Mock'
>>> mock.name
'Real Python Mock'
```

`.name` is a common attribute for objects to use. So, `Mock` doesn't let you set that value on the instance in the same way you can with `.return_value` or `.side_effect`. If you access `mock.name` you will create a `.name` attribute instead of configuring your mock.

You can configure an existing `Mock` using `.configure_mock()`:

Python

```
>>> mock = Mock()
>>> mock.configure_mock(return_value=True)
>>> mock()
True
```

By unpacking a dictionary into either `.configure_mock()` or `Mock.__init__()`, you can even configure your Python mock object's attributes. Using `Mock` configurations, you could simplify a previous example:

Python

```
# Verbose, old Mock
response_mock = Mock()
response_mock.json.return_value = {
    '12/25': 'Christmas',
    '7/4': 'Independence Day',
}

# Shiny, new .configure_mock()
holidays = {'12/25': 'Christmas', '7/4': 'Independence Day'}
response_mock = Mock(**{'json.return_value': holidays})
```

Now, you can create and configure Python mock objects. You can also use mocks to control the behavior of your application. So far, you've used mocks as arguments to functions or patching objects in the same module as your tests.

Next, you'll learn how to substitute your mocks for real objects in other modules.

patch()

`unittest.mock` provides a powerful mechanism for mocking objects, called `patch()`, which looks up an object in a given module and replaces that object with a `Mock`.

Usually, you use `patch()` as a decorator or a context manager to provide a scope in which you will mock the target object.

patch() as a Decorator

If you want to mock an object for the duration of your entire test function, you can use `patch()` as a function decorator.

To see how this works, reorganize your `my_calendar.py` file by putting the logic and tests into separate files:

Python

```
import requests
from datetime import datetime

def is_weekday():
    today = datetime.today()
    # Python's datetime library treats Monday as 0 and Sunday as 6
    return (0 <= today.weekday() < 5)

def get_holidays():
    r = requests.get('http://localhost/api/holidays')
    if r.status_code == 200:
        return r.json()
    return None
```

These functions are now in their own file, separate from their tests. Next, you'll re-create your tests in a file called `tests.py`.

Up to this point, you've monkey patched objects in the file in which they exist. Monkey patching is the replacement of one object with another at runtime. Now, you'll use `patch()` to replace your objects in `my_calendar.py`:

Python

```
import unittest
from my_calendar import get_holidays
from requests.exceptions import Timeout
from unittest.mock import patch

class TestCalendar(unittest.TestCase):
    @patch('my_calendar.requests')
    def test_get_holidays_timeout(self, mock_requests):
            mock_requests.get.side_effect = Timeout
            with self.assertRaises(Timeout):
                get_holidays()
                mock_requests.get.assert_called_once()

if __name__ == '__main__':
    unittest.main()
```

Originally, you created a `Mock` and patched `requests` in the local scope. Now, you need to access the `requests` library in `my_calendar.py` from `tests.py`.

For this case, you used `patch()` as a decorator and passed the target object's path. The target path was `'my_calendar.requests'` which consists of the module name and the object.

You also defined a new parameter for the test function. `patch()` uses this parameter to pass the mocked object into your test. From there, you can modify the mock or make assertions as necessary.

You can execute this test module to ensure it's working as expected:

Shell

```
$ python tests.py
.
----------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Technical Detail: `patch()` returns an instance of `MagicMock`, which is a `Mock` subclass. `MagicMock` is useful because it implements most magic methods for you, such as `.__len__()`, `.__str__()`, and `.__iter__()`, with reasonable defaults.

Using `patch()` as a decorator worked well in this example. In some cases, it is more readable, more effective, or easier to use `patch()` as a context manager.

`patch()` as a Context Manager

Sometimes, you'll want to use `patch()` as a context manager rather than a decorator. Some reasons why you might prefer a context manager include the following:

- You only want to mock an object for a part of the test scope.
- You are already using too many decorators or parameters, which hurts your test's readability.

To use `patch()` as a context manager, you use Python's `with` statement:

Python

```
import unittest
from my_calendar import get_holidays
from requests.exceptions import Timeout
from unittest.mock import patch

class TestCalendar(unittest.TestCase):
    def test_get_holidays_timeout(self):
        with patch('my_calendar.requests') as mock_requests:
            mock_requests.get.side_effect = Timeout
            with self.assertRaises(Timeout):
                get_holidays()
                mock_requests.get.assert_called_once()

if __name__ == '__main__':
    unittest.main()
```

When the test exits the `with` statement, `patch()` replaces the mocked object with the original.

Until now, you've mocked complete objects, but sometimes you'll only want to mock a part of an object.

Patching an Object's Attributes

Let's say you only want to mock one method of an object instead of the entire object. You can do so by using `patch.object()`.

For example, `.test_get_holidays_timeout()` really only needs to mock `requests.get()` and set its `.side_effect` to `Timeout`:

Python

```
import unittest
from my_calendar import requests, get_holidays
from unittest.mock import patch

class TestCalendar(unittest.TestCase):
    @patch.object(requests, 'get', side_effect=requests.exceptions.Timeout)
    def test_get_holidays_timeout(self, mock_requests):
            with self.assertRaises(requests.exceptions.Timeout):
                get_holidays()

if __name__ == '__main__':
    unittest.main()
```

In this example, you've mocked only `get()` rather than all of `requests`. Every other attribute remains the same.

`object()` takes the same configuration parameters that `patch()` does. But instead of passing the target's path, you provide the target object, itself, as the first parameter. The second parameter is the attribute of the target object that you are trying to mock. You can also use `object()` as a context manager like `patch()`.

> **Further Reading:** Besides objects and attributes, you can also `patch()` dictionaries with `patch.dict()`.

Learning how to use `patch()` is critical to mocking objects in other modules. However, sometimes it's not obvious what the target object's path is.

Where to Patch

Knowing where to tell `patch()` to look for the object you want mocked is important because if you choose the wrong target location, the result of `patch()` could be something you didn't expect.

Let's say you are mocking `is_weekday()` in `my_calendar.py` using `patch()`:

Python

```
>>> import my_calendar
>>> from unittest.mock import patch

>>> with patch('my_calendar.is_weekday'):
...     my_calendar.is_weekday()
...
<MagicMock name='is_weekday()' id='4336501256'>
```

First, you import `my_calendar.py`. Then you patch `is_weekday()`, replacing it with a `Mock`. Great! This is working as expected.

Now, let's change this example slightly and import the function directly:

Python

```
>>> from my_calendar import is_weekday
>>> from unittest.mock import patch

>>> with patch('my_calendar.is_weekday'):
...     is_weekday()
...
False
```

> **Note:** Depending on what day you are reading this tutorial, your console output may read `True` or `False`. The important thing is that the output is not a `Mock` like before.

Notice that even though the target location you passed to `patch()` did not change, the result of calling `is_weekday()` is different. The difference is due to the change in how you imported the function.

`from my_calendar import is_weekday` binds the real function to the local scope. So, even though you `patch()` the function later, you ignore the mock because you already have a local reference to the un-mocked function.

A good rule of thumb is to `patch()` the object where it is looked up.

In the first example, mocking `'my_calendar.is_weekday()'` works because you look up the function in the `my_calendar` module. In the second example, you have a local reference to `is_weekday()`. Since you use the function found in the local scope, you should mock the local function:

Python

```
>>> from unittest.mock import patch
>>> from my_calendar import is_weekday

>>> with patch('__main__.is_weekday'):
...     is_weekday()
...
<MagicMock name='is_weekday()' id='4502362992'>
```

Now, you have a firm grasp on the power of `patch()`. You've seen how to `patch()` objects and attributes as well as where to patch them.

Next, you'll see some common problems inherent in object mocking and the solutions that `unittest.mock` provides.

Common Mocking Problems

Mocking objects can introduce several problems into your tests. Some problems are inherent in mocking while others are specific to `unittest.mock`. Keep in mind that there are other issues with mocking that are not mentioned in this tutorial.

The ones covered here are similar to each other in that the problem they cause is fundamentally the same. In each case, the test assertions are irrelevant. Though the intention of each mock is valid, the mocks themselves are not.

Changes to Object Interfaces and Misspellings

Classes and function definitions change all the time. When the interface of an object changes, any tests relying on a `Mock` of that object may become irrelevant.

For example, you rename a method but forget that a test mocks that method and invokes `.assert_not_called()`. After the change, `.assert_not_called()` is still `True`. The assertion is not useful, though, because the method no longer exists.

Irrelevant tests may not sound critical, but if they are your only tests and you assume that they work properly, the situation could be disastrous for your application.

A problem specific to `Mock` is that a misspelling can break a test. Recall that a `Mock` creates its interface when you access its members. So, you will inadvertently create a new attribute if you misspell its name.

If you call `.asert_called()` instead of `.assert_called()`, your test will not raise an `AssertionError`. This is because you've created a new method on the Python mock object named `.asert_called()` instead of evaluating an actual assertion.

> **Technical Detail:** Interestingly, `assret` is a special misspelling of `assert`. If you try to access an attribute that starts with `assret` (or `assert`), `Mock` will automatically raise an `AttributeError`.

These problems occur when you mock objects within your own codebase. A different problem arises when you mock objects interacting with external codebases.

Changes to External Dependencies

Imagine again that your code makes a request to an external API. In this case, the external dependency is the API which is susceptible to change without your consent.

On one hand, unit tests test isolated components of code. So, mocking the code that makes the request helps you to test your isolated components under controlled conditions. However, it also presents a potential problem.

If an external dependency changes its interface, your Python mock objects will become invalid. If this happens (and the interface change is a breaking one), your tests will pass because your mock objects have masked the change, but your production code will fail.

Unfortunately, this is not a problem that `unittest.mock` provides a solution for. You must exercise judgment when mocking external dependencies.

All three of these issues can cause test irrelevancy and potentially costly issues because they threaten the integrity of your mocks. `unittest.mock` gives you some tools for dealing with these problems.

Avoiding Common Problems Using Specifications

As mentioned before, if you change a class or function definition or you misspell a Python mock object's attribute, you can cause problems with your tests.

These problems occur because `Mock` creates attributes and methods when you access them. The answer to these issues is to prevent `Mock` from creating attributes that don't conform to the object you're trying to mock.

When configuring a `Mock`, you can pass an object specification to the `spec` parameter. The `spec` parameter accepts a list of names or another object and defines the mock's interface. If you attempt to access an attribute that does not belong to the specification, `Mock` will raise an `AttributeError`:

Python

```
>>> from unittest.mock import Mock
>>> calendar = Mock(spec=['is_weekday', 'get_holidays'])

>>> calendar.is_weekday()
<Mock name='mock.is_weekday()' id='4569015856'>
>>> calendar.create_event()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/Cellar/python/3.6.5/Frameworks/Python.framework/Versions/3.6/lib/python3.6/unittest/mock.py", line 58
    raise AttributeError("Mock object has no attribute %r" % name)
AttributeError: Mock object has no attribute 'create_event'
```

Here, you've specified that `calendar` has methods called `.is_weekday()` and `.get_holidays()`. When you access `.is_weekday()`, it returns a `Mock`. When you access `.create_event()`, a method that does not match the specification, `Mock` raises an `AttributeError`.

Specifications work the same way if you configure the `Mock` with an object:

Python

```
>>> import my_calendar
>>> from unittest.mock import Mock

>>> calendar = Mock(spec=my_calendar)
>>> calendar.is_weekday()
<Mock name='mock.is_weekday()' id='4569435216'>
>>> calendar.create_event()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/Cellar/python/3.6.5/Frameworks/Python.framework/Versions/3.6/lib/python3.6/unittest/mock.py", line 58
    raise AttributeError("Mock object has no attribute %r" % name)
AttributeError: Mock object has no attribute 'create_event'
```

`.is_weekday()` is available to `calendar` because you configured `calendar` to match the `my_calendar` module's interface.

Furthermore, `unittest.mock` provides convenient methods of automatically specifying a `Mock` instance's interface.

One way to implement automatic specifications is `create_autospec`:

Python

```
>>> import my_calendar
>>> from unittest.mock import create_autospec

>>> calendar = create_autospec(my_calendar)
>>> calendar.is_weekday()
<MagicMock name='mock.is_weekday()' id='4579049424'>
>>> calendar.create_event()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/Cellar/python/3.6.5/Frameworks/Python.framework/Versions/3.6/lib/python3.6/unittest/mock.py", line 58
    raise AttributeError("Mock object has no attribute %r" % name)
AttributeError: Mock object has no attribute 'create_event'
```

Like before, `calendar` is a `Mock` instance whose interface matches `my_calendar`. If you're using `patch()`, you can send an argument to the `autospec` parameter to achieve the same result:

Python

```
>>> import my_calendar
>>> from unittest.mock import patch

>>> with patch('__main__.my_calendar', autospec=True) as calendar:
...     calendar.is_weekday()
...     calendar.create_event()
...
<MagicMock name='my_calendar.is_weekday()' id='4579094312'>
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/Cellar/python/3.6.5/Frameworks/Python.framework/Versions/3.6/lib/python3.6/unittest/mock.py", line 58
    raise AttributeError("Mock object has no attribute %r" % name)
AttributeError: Mock object has no attribute 'create_event'
```


Conclusion

You've learned so much about mocking objects using `unittest.mock`!

Now, you're able to:

- Use `Mock` to imitate objects in your tests
- Check usage data to understand how you use your objects
- Customize your mock objects' return values and side effects
- `patch()` objects throughout your codebase
- See and avoid problems with using Python mock objects

You have built a foundation of understanding that will help you build better tests. You can use mocks to gain insights into your code that you would not have been able to get otherwise.

I leave you with one final disclaimer. *Beware of overusing mock objects!*

It's easy to take advantage of the power of Python mock objects and mock so much that you actually decrease the value of your tests.