Testing With Mocks

Interacting Units

Unit tests are ideally **pinpointed**. They test a specific class, function, or other component in isolation.

But what if the test target depends on other components… and without them, won't work at all?

Sensor

```
class Sensor:
    def __init__(self, name, datastore):
        self.name = name
        self.datastore = datastore
    def receive(self, datum):
        self.datastore.record_datapoint(datum)
    def recent(self, timepoints):
        return self.datastore.fetch(timepoints)
```

```
class Datastore:
    def __init__(self, address, username, password):
        'Initialize a live connection to the database.'
        self.conn = dbconnect(address, username, password)
    def record_datapoint(self, datum):
        'Add a datapoint to the database.'
        # ...
    def fetch(self, num_records):
        'Get the most recently recorded data points.'
        # ...
```

How To Test

We want to test the behavior of `Sensor`. That also depends on `Datastore`, a complex object to set up.

If we don't want to configure and maintain a database server just so we can run unit tests, what are our options?

Stubs

A **stub** is a custom-made object whose methods return predefined data.

The idea: whip up a quick stub that mimics `Datastore`, close enough for the purpose of the test.

```
class StubDatastore:
    def __init__(self, fetched_data):
        self.fetched_data = fetched_data
    def record_datapoint(self, datum):
        pass
    def fetch(self, num_records):
        return self.fetched_data
```

Stub test

```
import unittest
import sensors

class StubDatastore:
    # As defined on the previous slide.

class TestSensor(unittest.TestCase):
    def test_record_data(self):
        # recent 5
        dstore = StubDatastore([5, 6, 7, 8, 9])
        sensor = sensors.Sensor('Temperature', dstore)
        for x in range(10):
            sensor.receive(x)
        self.assertEqual([5, 6, 7, 8, 9], sensor.recent(5))
```

Another option: Fakes

A **fake** is an object with a close-enough working implementation... but not the same as the real one you use in production.

Again, you create this just to run the test.

```
class FakeDatastore:
    def __init__(self):
        self.data = []
    def record_datapoint(self, datum):
        self.data.append(datum)
    def fetch(self, num_records):
        return self.data[-num_records:]
```

Fake test

```
import unittest
import sensors

class FakeDatastore:
    # As defined on the previous slide.

class TestSensor(unittest.TestCase):
    def test_record_data(self):
        dstore = FakeDatastore()
        sensor = sensors.Sensor('Temperature', dstore)
        for x in range(10):
            sensor.receive(x)
        self.assertEqual([5, 6, 7, 8, 9], sensor.recent(5))
```

Downsides

Stubs and fakes have pros and cons:

Upside: Can be quick and easy to set up, and can create quite understandable test code.

Con: You can't easily make assertions on how many times methods are called, whether they are passed the right arguments, etc. **Verification.**

Also: Fragile. Implementation completely separate from the actual code. Can easily diverge, leading to tests passing when they should fail.

And: Only usable when you have the ability to inject the fake or stub object. (Sensor uses *constructor injection*.)

Alternative: Mocks

A **mock** is an object which registers its methods calls.

Your test can then verify expected actions are performed.

Typically, you'll use a mocking library. In Python, you use `unittest.mock`.

(In 2.x, you must install the separate `mock` library.)

Mock Functions

Python has a `Mock` class. Mocks are *callable*: you can call them like a function. And you can set their return value:

```
>>> from unittest.mock import Mock
>>> roll_dice = Mock()
>>> roll_dice.return_value = 7
>>> roll_dice()
7
```

`return_value` is a shortcut. For more complexity, use `side_effect`:

```
>>> def rolling_dice_effect():
...     print("ROLLING...")
...     return 9
>>> roll_dice.side_effect = rolling_dice_effect
>>> roll_dice()
ROLLING...
9
```

Mock Objects

You can also treat a mock like a mock *object*, instead of a function. This lets you create mock *methods* on that object, just by assigning to them.

```
>>> from unittest.mock import Mock
>>> meal = Mock()
>>> meal.total_calories.return_value = 750
>>> meal.total_calories()
750
```

This mock could be used to model a class like this:

```
class Meal:
    def __init__(self, food_items):
        self.food_items = food_items
    def total_calories(self):
        return sum(food_item.calories
                   for food_item in self.food_items)
```

To review...

A *mock* is a programmable, callable object. It's an instance of the *Mock* class.

Every mock can be invoked... treated like a function or method.

Or: you can treat that same mock like an **object**, and create additional mocks as attributes - which are callable, and thus act like methods. These are *also* instances of `Mock`.

To review...

There's only one `Mock` type. Not separate "mock object" and "mock method" types:

```
>>> foo = Mock()
>>> foo.bar.return_value = 1
>>> type(foo)
<class 'unittest.mock.Mock'>
>>> type(foo.bar)
<class 'unittest.mock.Mock'>
```

You create new mock attributes automatically, just by referencing them on your mock object:

```
>>> 'baz' in dir(foo)
False
>>> foo.baz
<Mock name='mock.baz' id='4524451208'>
>>> 'baz' in dir(foo)
True
```

Checking Mock Behavior

Mocks keep track of when they're called.

```
>>> foo = Mock()
>>> foo.called
False
>>> foo.call_count
0
>>> x = foo("Calling ONCE!")
>>> foo.called
True
>>> foo.call_count
1
>>> foo.call_args
call('Calling ONCE!')
>>> x = foo("Calling TWICE!")
>>> foo.called
True
>>> foo.call_count
2
>>> foo.call_args
call('Calling TWICE!')
```

Mock Call Assertions

Mock objects have assertion methods. Useful in tests.

Like any assertion, they do nothing if true, and raise `AssertionError` if false.

```
>>> foo = Mock()
>>> foo.assert_not_called()

>>> x = foo("Calling ONCE!")
>>> foo.assert_called_once()

>>> x = foo("Calling TWICE!")
>>> foo.assert_called_once()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mock.py", line 795, in assert_called_once
    raise AssertionError(msg)
AssertionError: Expected 'mock' to have been called once. Called 2 times.
```

Mocking the sensor

How can we use mocks with the sensor example?
We want to mock `Datastore`.

```
class Sensor:
    def __init__(self, name, datastore):
        self.name = name
        self.datastore = datastore
    def receive(self, datum):
        self.datastore.record_datapoint(datum)
    def recent(self, timepoints):
        return self.datastore.fetch(timepoints)
class Datastore:
    def __init__(self, address, username, password):
        'Initialize a live connection to the database.'
        self.conn = dbconnect(address, username, password)
    def record_datapoint(self, datum):
        'Add a datapoint to the database.'
        # ...
    def fetch(self, num_records):
        'Get the most recently recorded data points.'
        # ...
```

The mock spec

We can pass a class to the `Mock` constructor, as a *spec*. The mock will have the same methods.

```
>>> from unittest.mock import Mock
>>> mock_datastore = Mock(Datastore)
>>> x = mock_datastore.fetch(5)
```

Referencing an attribute not on the spec triggers an error:

```
>>> x = mock_datastore.notamethod()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mock.py", line 582, in __getattr__
    raise AttributeError("Mock object has no attribute %r" % name)
AttributeError: Mock object has no attribute 'notamethod'
```

Without a spec, `notamethod()` would be automatically created. This makes your test code more strict (in a good way).

Using Mock in a Test

```
import unittest
from unittest.mock import Mock
import sensors

class TestSensor(unittest.TestCase):
    def test_record_data(self):
        # set up mock
        dstore = Mock(sensors.Datastore)
        dstore.fetch.return_value = [5, 6, 7, 8, 9]

        sensor = sensors.Sensor('Temperature', dstore)
        for x in range(10):
            sensor.receive(x)
        self.assertEqual([5, 6, 7, 8, 9], sensor.recent(5))

        # assert expected mock behavior
        self.assertEqual(10, dstore.record_datapoint.call_count)
        dstore.fetch.assert_called_once()
```

Notice we don't need to configure `record_datapoint()`.

Another Example

Imagine your game program uses randomness in its combat.

```
# combat.py
import random
def attack():
    if random.random() > 0.5:
        return 'HIT'
    else:
        return 'MISS'
```

`random()` gives a random number between 0.0 and 1.0.

How do you write sane tests for this? You could make a mock of `random()`, but how will you get that mock inside the `attack()` function?

Patching

Solution: you can *patch* the `random` module.

```
>>> # Create a mock random() function:
... original_random = random.random
>>> random.random = Mock()
>>> random.random.return_value = 0.75
>>>
>>> # Now exercise it...
... for n in range(10):
...     print(attack(), end=', ')
...
>>> print('\n100% hits!')
HIT, HIT, HIT, HIT, HIT, HIT, HIT, HIT, HIT, HIT,
100% hits!
>>>
>>> # IMPORTANT: Restore the original when you're done!
... random.random = original_random
```

unittest.mock.patch()

Python lets you *inject* mocks by *patching*.

`patch()` works by temporarily changing the object or function a name points to. *It changes it to a new mock object.*

This works even INSIDE a function using the patched object!

```
>>> from unittest.mock import patch
>>> with patch('random.random') as mock_random:
...     mock_random.return_value = 0.75
...     for x in range(10):
...         print(attack(), end=', ')
...     print('\n100% hits!')
...
HIT, HIT, HIT, HIT, HIT, HIT, HIT, HIT, HIT, HIT,
100% hits!
```

Patching In Unit Tests

You'll normally use `patch()` in unit tests.

```
# test_combat.py
import unittest
from unittest.mock import patch

from combat import attack

class TestCombat(unittest.TestCase):
    def test_attack(self):
        with patch('random.random') as mock_random:
            mock_random.return_value = 0.75
            self.assertEqual('HIT', attack())
            mock_random.assert_called_once()
```

@patch

`patch()` can be used as a decorator on the test method.

This seems to be the most common way it's used in practice.

```
# test_combat.py
import unittest
from unittest.mock import patch

from combat import attack

class TestCombat(unittest.TestCase):
    @patch('random.random')
    def test_attack(self, mock_random):
        mock_random.return_value = 0.75
        self.assertEqual('HIT', attack())
        mock_random.assert_called_once()
```

@patch

Break it down:

```
@patch('random.random')
def test_attack(self, mock_random):
    mock_random.return_value = 0.75
```

- `@patch()` takes a string argument. Module and function (or class, etc.)
- When you use `@patch()`, your test function now takes an argument: the **mocked object**.
- You can program the behavior of mock objects, and make assertions on their activity.
- `patch()` is able to (temporarily) modify the module containing the patched object directly, so you don't have to find a way to inject the mock deep into your code. It's effectively already in there.

Every day patching

`patch()` can be convenient to use, even when not strictly necessary.

```
class TestSensor(unittest.TestCase):
    @patch('sensors.Datastore')
    def test_record_data(self, MockDatastore):
        # set up mock expecations
        dstore = MockDatastore()
        dstore.fetch.return_value = [5, 6, 7, 8, 9]
        sensor = sensors.Sensor('Temperature', dstore)
        for x in range(10):
            sensor.receive(x)
        self.assertTrue(dstore.record_datapoint.called)
        self.assertEqual([5, 6, 7, 8, 9], sensor.recent(5))
        self.assertTrue(dstore.fetch.called)

        # reset_mock() resets the mock's state, for new tests.
        dstore.reset_mock()
        dstore.fetch.return_value = [7, 8, 9]
        self.assertFalse(dstore.fetch.called)
        self.assertEqual([7, 8, 9], sensor.recent(3))
        self.assertTrue(dstore.fetch.called)
```

Why Use Mocks?

Mocks improve *isolation*, by letting you override the normal behavior of What that gives you:

- Run code that depends on other, complex objects, without actually setting up those objects
- Simulate error conditions (without jumping through hoops)
- Simplify the testing environment, and reduce setup work. Sometimes by a lot
- Better control starting state
- Write unit tests for one component before the components they depend on are written. Just create stubs, and mock their behavior
- Avoid undesired side effects (e.g. network API calls)
- Increase speed (by avoiding running expensive code)

And some downsides too.

Downsides of Mocks

- Can create hard-to-change and hard-to-maintain tests
- Can create fragile tests, tied to details of current implementation
- Mocking tends to work by leveraging more "magic" features. So they sometimes conflict with your own advanced code
- Can make tests harder to understand
- Can get out of sync with the object it's mocking

Given this, avoid mocks when they are not needed. **But don't hesitate to use them when they are.**