Mock Patching

Another Example

Imagine your game program uses randomness in its combat.

```python
# 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.

```python
# 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.

```python
# 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.

Patching and specs

Remember with Mock, you can pass in a spec:

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

This pre-defines the methods from `Datastore`, and triggers an error for any others:

```
>>> 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'
```

Turns out, `@patch()` does NOT do this by default.

Patching in mock objects

```
with patch('sensors.Datastore', autospec=True) as mock_datastore:
    # Datastore.fetch() is supposed to take 1 argument
    mock_datastore.fetch(1, 2, 3, 4, 5)
```

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
  File
"/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/unittest/moc
k.py", line 950, in __call__
    _mock_self._mock_check_sig(*args, **kwargs)
  File
"/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/unittest/moc
k.py", line 102, in checksig
    sig.bind(*args, **kwargs)
  File
"/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/inspect.py",
line 3002, in bind
    return args[0]._bind(args[1:], kwargs)
  File
"/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/inspect.py",
line 2923, in _bind
    raise TypeError('too many positional arguments') from None
```

Patching in mock objects

```python
import unittest
from unittest.mock import patch
class TestSensor(unittest.TestCase):
    # Could also say "spec=True", but autospec is more powerful.
    @patch('sensors.Datastore', autospec=True)
    def test_record_data(self, dstore):
        # set up mock expecations
        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)
```

Autospec-ing

```
from unittest.mock import patch
with patch('sensors.Datastore', autospec=True) as dstore:
    # Can use strict mock object. Datastore.fetch() takes one arg
    val = dstore.fetch(5)
    # This will raise TypeError:
    val = dstore.fetch(1, 2, 3)

# Alternatively:
from unittest.mock import create_autospec
dstore = create_autospec(sensors.Datastore)

# This does NOT currently work. But it might in the future:
from unittest.mock import Mock
dstore = Mock(autospec=sensors.Datastore)
# (Be careful. If you do this, there's no error, but
# the mock you get is NOT strict at all.)
```