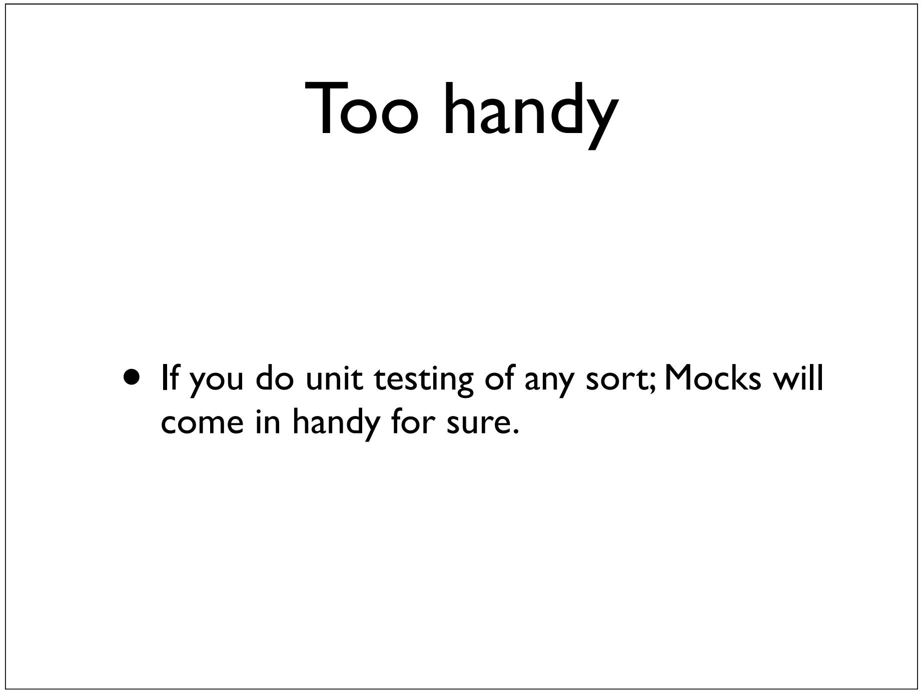
Mock Testing
in the language we love

PUN lightning talk - 28-08-2008

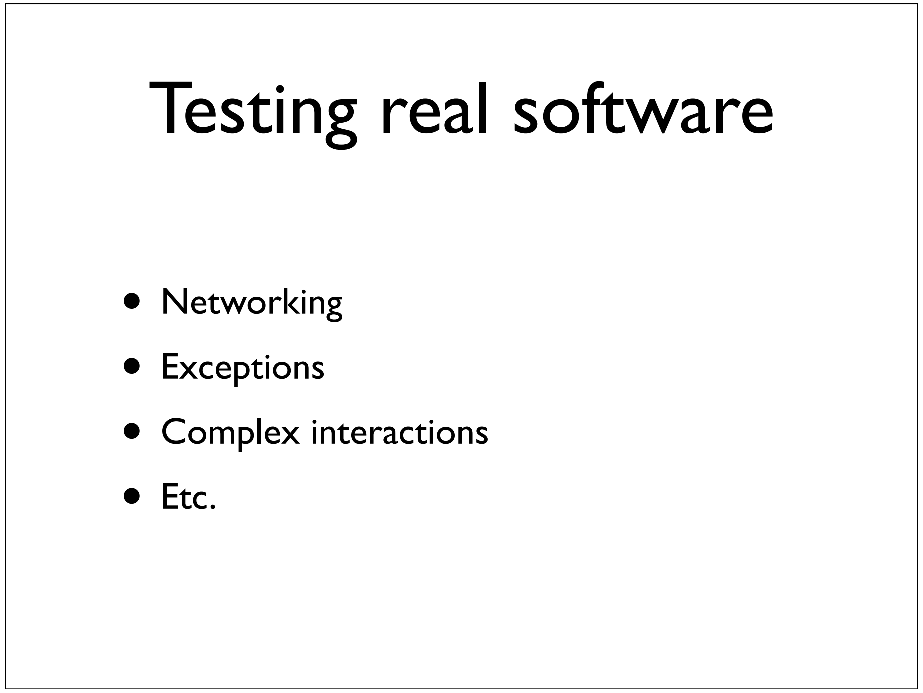
by Remco Wendt - Maykin Media
remco@maykinmedia.nl



Too handy

- If you do unit testing of any sort; Mocks will come in handy for sure.

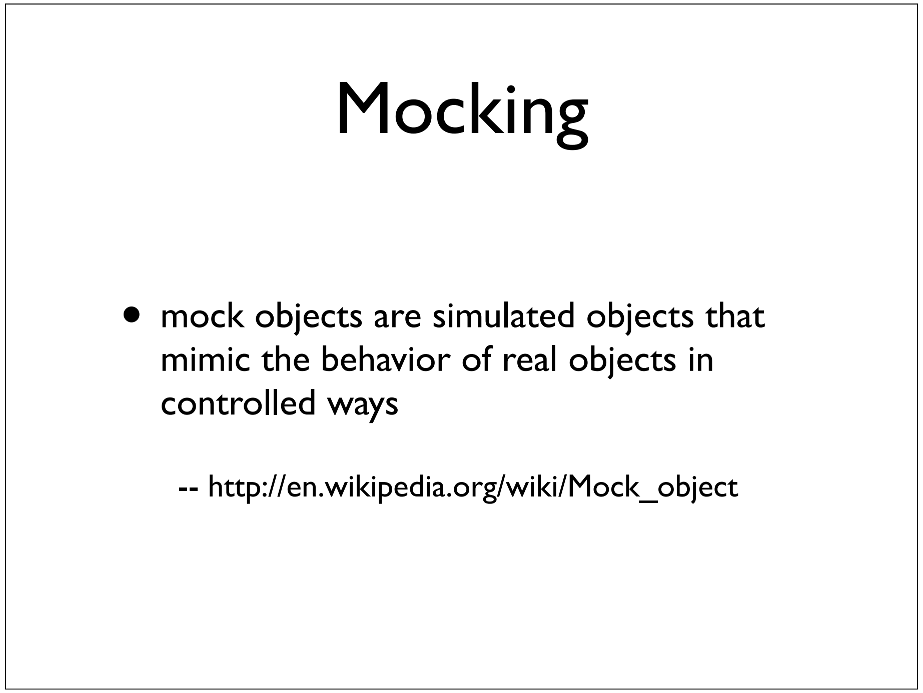
A lot of people doing unit testing will already know about this. But for those of you that haven’t used Mocking, it’s too handy a practice not to use.



Testing real software

- Networking
- Exceptions
- Complex interactions
- Etc.

When testing your software you will run into various conditions that makes it harder to test your software:
- Testing a networking client when you have no network connectivity
- Testing certain exceptional conditions (for example what happens with your software if network connectivity is lost, does it handle the exception well?)
- Complex interactions: the whole idea of UNIT testing is to focus your testing effort.
- Some operations maybe slow, if you have to wait minutes for a test run to finish. Then in practice you stop testing your code.

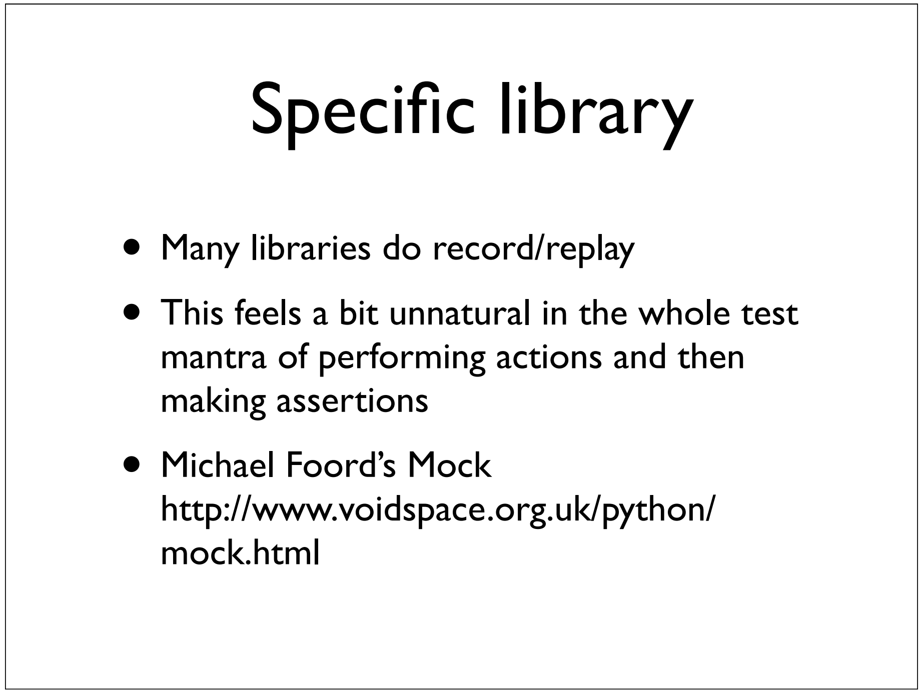


Mocking

- mock objects are simulated objects that mimic the behavior of real objects in controlled ways

-- http://en.wikipedia.org/wiki/Mock_object

The solution is Mocking, which is relatively easy to implement in a dynamic language like Python. You simulate your real objects short cutting a lot of (already tested) application logic and having a lot of control on what is returned from these simulated objects and being able to check if these objects where called the way you expected them to be called.

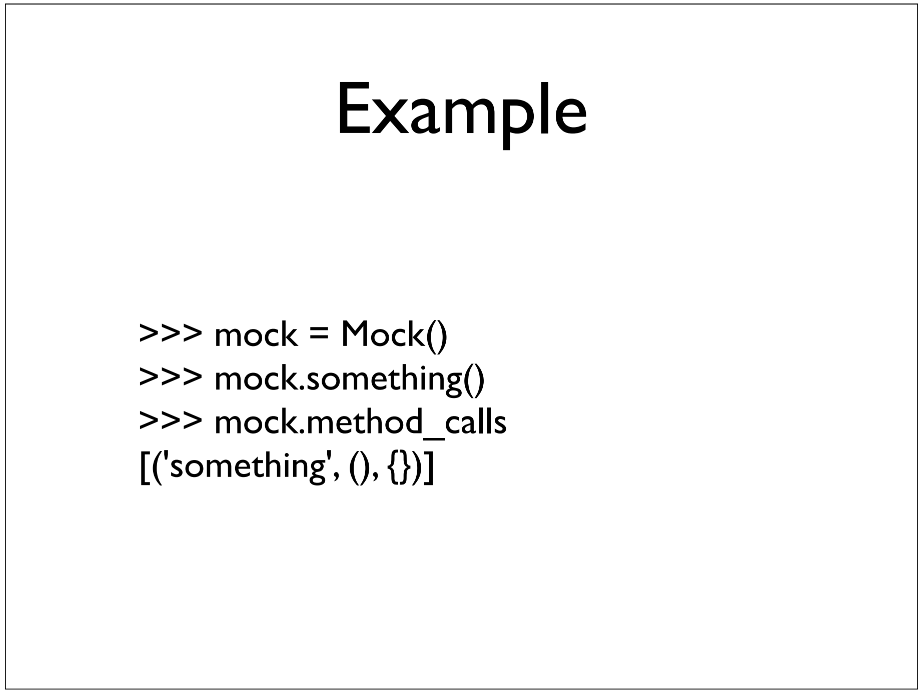


Specific library

- Many libraries do record/replay
- This feels a bit unnatural in the whole test mantra of performing actions and then making assertions
- Michael Foord's Mock http://www.voidspace.org.uk/python/mock.html

Many mocking libraries use the record/replay model:
- you first record what you expect that the code being tested will do
- you then stop the recording with a replay action priming the mock for use by the code being tested
- then you check whether the expectations where met

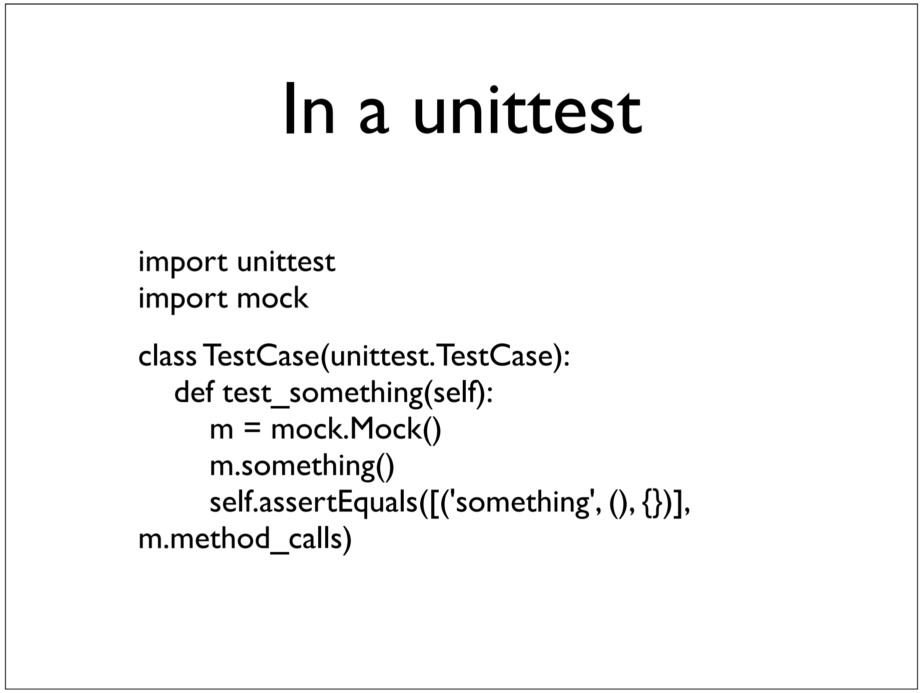
Unit test wise we (well I) rather first make calls to the code being tested (actions) and then check whether what happened is what we expected (assertions). This is the way you would normally do unit testing



Example

```
>>> mock = Mock()
>>> mock.something()
>>> mock.method_calls
[('something', (), {})]
```

You create a mock, then a call to this mock is registered and afterwards you can check what happened

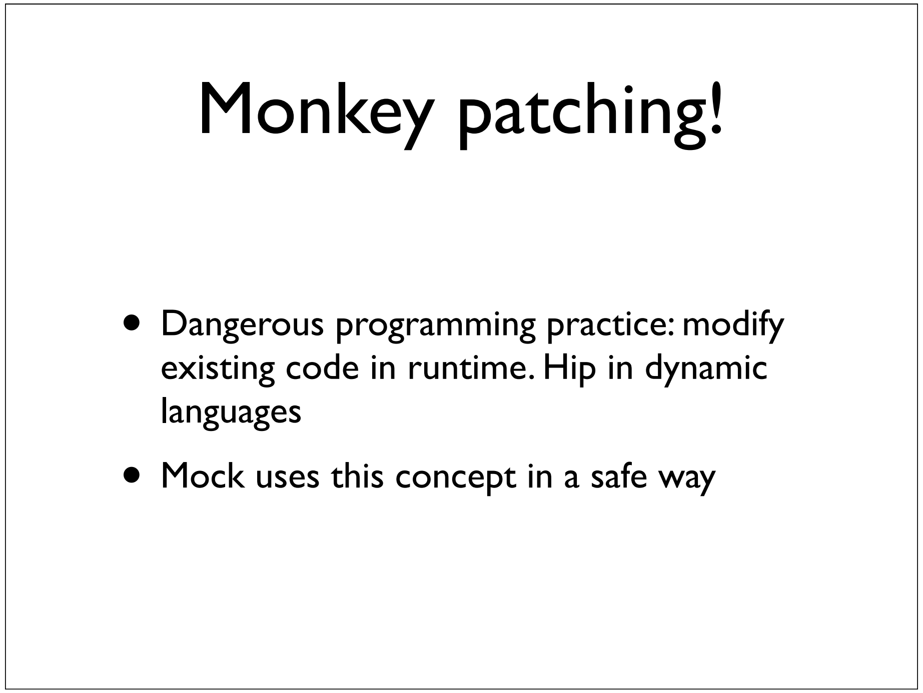


In a unittest

```
import unittest
import mock

class TestCase(unittest.TestCase):
    def test_something(self):
        m = mock.Mock()
        m.something()
        self.assertEquals([('something', (), {})],
m.method_calls)
```

This is how a simple mock test may look in a unit test.



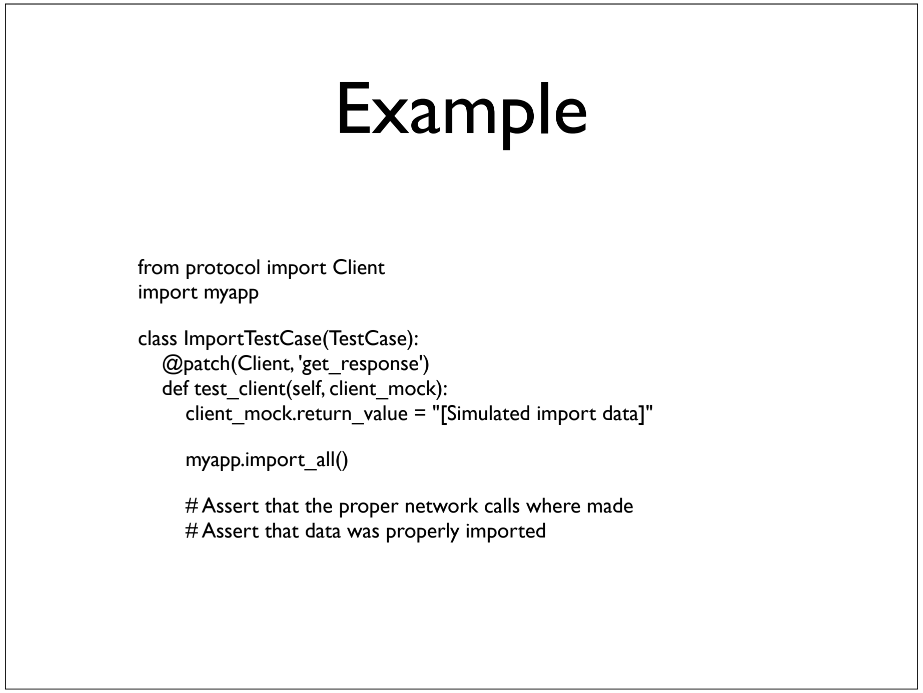
Monkey patching!

- Dangerous programming practice: modify existing code in runtime. Hip in dynamic languages
- Mock uses this concept in a safe way

A better way of doing mock testing is through using monkey patches.

Monkey patching is modifying existing code at runtime (for example overwriting an existing function in the python core lib). This is of course a dangerous programming practice. But mock has a very nice way of handling this:

You declare patches using decorators per unittest function. This patch will only exist in the scope of that specific unittest. When the unittest is done the monkey patch is no longer in place



Example

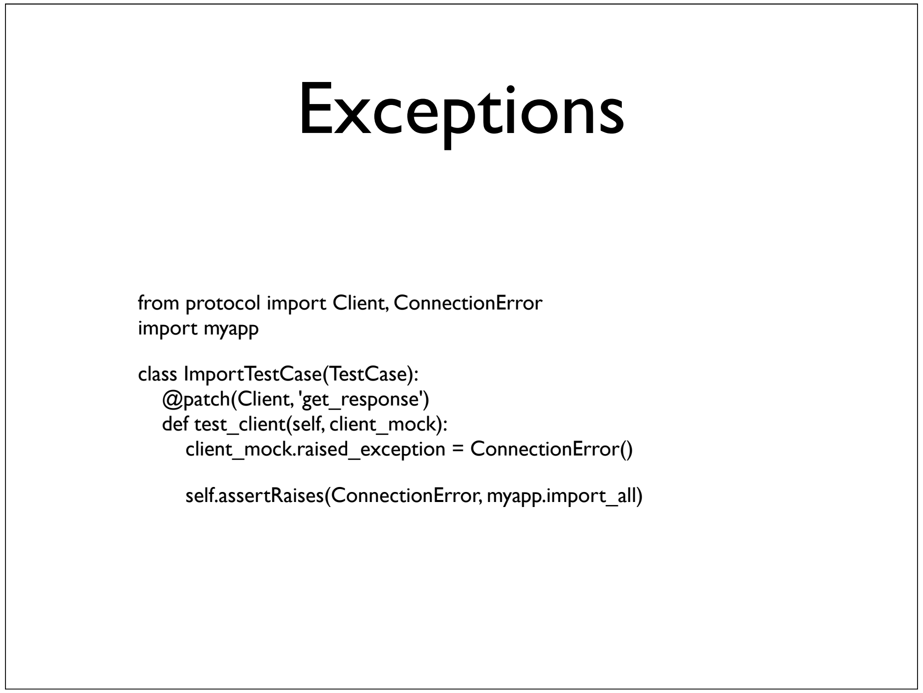
```
from protocol import Client
import myapp

class ImportTestCase(TestCase):
    @patch(Client, 'get_response')
    def test_client(self, client_mock):
        client_mock.return_value = "[Simulated import data]"

        myapp.import_all()

        # Assert that the proper network calls where made
        # Assert that data was properly imported
```

This example shows an app which imports data that comes from a remote server into a local database. The mock shortcuts the network functionality here, patching our client class, deciding what data will be returned, allowing our importer to run with the mock and then afterwards checking if the right network calls where made and if the simulated import data is properly imported



Exceptions

```
from protocol import Client, ConnectionError
import myapp

class ImportTestCase(TestCase):
    @patch(Client, 'get_response')
    def test_client(self, client_mock):
        client_mock.raised_exception = ConnectionError()

        self.assertRaises(ConnectionError, myapp.import_all)
```

This is of course a very simple test but using these ingredients you can do pretty advanced unit and integration tests thanks!