Mock Documentation

Release 0.7.2

Michael Foord

May 30, 2011

CONTENTS

1 API Documentation **3**
- 1.1 The Mock Class . . . 3
- 1.2 Patch Decorators . . . 9
- 1.3 Sentinel . . . 15
- 1.4 Mocking Magic Methods . . . 16
- 1.5 Magic Mock . . . 18
- 1.6 mocksignature . . . 20

2 User Guide **25**
- 2.1 Getting Started with Mock . . . 25
- 2.2 Further Examples . . . 34
- 2.3 Mock Library Comparison . . . 53
- 2.4 TODO and Limitations . . . 63
- 2.5 CHANGELOG . . . 64

3 Installing **73**

4 Quick Guide **75**

5 References **79**

6 Tests **81**

7 Older Versions **83**

8 Terminology **85**

Index **87**

Author Michael Foord

Co-maintainer Konrad Delong

Version 0.7.2

Date 2011/05/30

Homepage Mock Homepage

Download Mock on PyPI

Documentation PDF Documentation

License BSD License

Support Mailing list (testing-in-python@lists.idyll.org)

Issue tracker Google code project

mock is a Python module that provides a core `Mock` class. It removes the need to create a host of stubs throughout your test suite. After performing an action, you can make assertions about which methods / attributes were used and arguments they were called with. You can also specify return values and set needed attributes in the normal way.

The mock module also provides a `patch()` decorator that handles patching module and class level attributes within the scope of a test, along with `sentinel` for creating unique objects. See the quick guide for some examples of how to use `Mock`, `MagicMock` and `patch()`.

Mock is very easy to use and is designed for use with unittest. Mock is based on the 'action -> assertion' pattern instead of *'record -> replay'* used by many mocking frameworks.

mock is tested on Python versions 2.4-2.7 and Python 3.

CHAPTER

ONE

API DOCUMENTATION

1.1 The Mock Class

`Mock` is a flexible mock object intended to replace the use of stubs and test doubles throughout your code. Mocks are callable and create attributes as new mocks when you access them [1]. Accessing the same attribute will always return the same mock. Mocks record how you use them, allowing you to make assertions about what your code has done to them.

The `mock.patch()` decorators makes it easy to temporarily replace classes in a particular module with a Mock object.

class Mock(*spec=None*, *side_effect=None*, *return_value=<SentinelObject "DEFAULT">*, *wraps=None*, *name=None*, *spec_set=None*, *parent=None*)

Create a new `Mock` object. `Mock` takes several optional arguments that specify the behaviour of the Mock object:

- `spec`: This can be either a list of strings or an existing object (a class or instance) that acts as the specification for the mock object. If you pass in an object then a list of strings is formed by calling dir on the object (excluding unsupported magic attributes and methods). Accessing any attribute not in this list will raise an `AttributeError`.

  If `spec` is an object (rather than a list of strings) then *mock.__class__* returns the class of the spec object. This allows mocks to pass *isinstance* tests.

- `spec_set`: A stricter variant of `spec`. If used, attempting to *set* or get an attribute on the mock that isn't on the object passed as `spec_set` will raise an `AttributeError`.

- `side_effect`: A function to be called whenever the Mock is called. See the `Mock.side_effect` attribute. Useful for raising exceptions or dynamically changing return values. The function is called with the same arguments as the mock, and

[1] The only exceptions are magic methods and attributes (those that have leading and trailing double underscores). Mock doesn't create these but instead of raises an `AttributeError`. This is because the interpreter will often implicitly request these methods, and gets *very* confused to get a new Mock object when it expects a magic method. If you need magic method support see *magic methods*.

unless it returns DEFAULT, the return value of this function is used as the return value.

Alternatively side_effect can be an exception class or instance. In this case the exception will be raised when the mock is called.

- return_value: The value returned when the mock is called. By default this is a new Mock (created on first access). See the Mock.return_value attribute.
- wraps: Item for the mock object to wrap. If wraps is not None then calling the Mock will pass the call through to the wrapped object (returning the real result and ignoring return_value). Attribute access on the mock will return a Mock object that wraps the corresponding attribute of the wrapped object (so attempting to access an attribute that doesn't exist will raise an AttributeError).

  If the mock has an explicit return_value set then calls are not passed to the wrapped object and the return_value is returned instead.
- name: If the mock has a name then it will be used in the repr of the mock. This can be useful for debugging. The name is propagated to child mocks.

Mock objects that use a class or an instance as a *spec* or *spec_set* are able to pass *isintance* tests:

```
>>> mock = Mock(spec=SomeClass)
>>> isinstance(mock, SomeClass)
True
>>> mock = Mock(spec_set=SomeClass())
>>> isinstance(mock, SomeClass)
True
```

1.1.1 Methods

Mock.**assert_called_with**(**args*, ***kwargs*)

This method is a convenient way of asserting that calls are made in a particular way:

```
>>> mock = Mock()
>>> mock.method(1, 2, 3, test='wow')
<mock.Mock object at 0x...>
>>> mock.method.assert_called_with(1, 2, 3, test='wow')
```

Mock.**assert_called_once_with**(**args*, ***kwargs*)

Assert that the mock was called exactly once and with the specified arguments.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar='baz')
>>> mock.assert_called_once_with('foo', bar='baz')
>>> mock('foo', bar='baz')
>>> mock.assert_called_once_with('foo', bar='baz')
Traceback (most recent call last):
```

```
    ...
AssertionError: Expected to be called once. Called 2 times.
```

Mock.**reset_mock**()

The reset_mock method resets all the call attributes on a mock object:

```
>>> mock = Mock(return_value=None)
>>> mock('hello')
>>> mock.called
True
>>> mock.reset_mock()
>>> mock.called
False
```

This can be useful where you want to make a series of assertions that reuse the same object. Note that `reset` *doesn't* clear the return value, `side_effect` or any child attributes. Attributes you have set using normal assignment are also left in place. Child mocks and the return value mock (if any) are reset as well.

1.1.2 Calling

Mock objects are callable. The call will return the value set as the `Mock.return_value` attribute. The default return value is a new Mock object; it is created the first time the return value is accessed (either explicitly or by calling the Mock) - but it is stored and the same one returned each time.

Calls made to the object will be recorded in the attributes.

If `Mock.side_effect` is set then it will be called after the call has been recorded but before any value is returned.

1.1.3 Attributes

Mock.**called**

A boolean representing whether or not the mock object has been called:

```
>>> mock = Mock(return_value=None)
>>> mock.called
False
>>> mock()
>>> mock.called
True
```

Mock.**call_count**

An integer telling you how many times the mock object has been called:

```
>>> mock = Mock(return_value=None)
>>> mock.call_count
0
>>> mock()
>>> mock()
>>> mock.call_count
2
```

Mock.**return_value**

Set this to configure the value returned by calling the mock:

```
>>> mock = Mock()
>>> mock.return_value = 'fish'
>>> mock()
'fish'
```

The default return value is a mock object and you can configure it in the normal way:

```
>>> mock = Mock()
>>> mock.return_value.attribute = sentinel.Attribute
>>> mock.return_value()
<mock.Mock object at 0x...>
>>> mock.return_value.assert_called_with()
```

return_value can also be set in the constructor:

```
>>> mock = Mock(return_value=3)
>>> mock.return_value
3
>>> mock()
3
```

Mock.**side_effect**

This can either be a function to be called when the mock is called, or an exception (class or instance) to be raised.

If you pass in a function it will be called with same arguments as the mock and unless the function returns the DEFAULT singleton the call to the mock will then return whatever the function returns. If the function returns DEFAULT then the mock will return its normal value (from the Mock.return_value.

An example of a mock that raises an exception (to test exception handling of an API):

```
>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

Using `side_effect` to return a sequence of values:

```
>>> mock = Mock()
>>> results = [1, 2, 3]
>>> def side_effect(*args, **kwargs):
...     return results.pop()
...
>>> mock.side_effect = side_effect
>>> mock(), mock(), mock()
(3, 2, 1)
```

The *side_effect* function is called with the same arguments as the mock (so it is wise for it to take arbitrary args and keyword arguments) and whatever it returns is used as the return value for the call. The exception is if *side_effect* returns `DEFAULT`, in which case the normal `Mock.return_value` is used.

```
>>> mock = Mock(return_value=3)
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> mock.side_effect = side_effect
>>> mock()
3
```

side_effect can be set in the constructor. Here's an example that adds one to the value the mock is called with and returns it:

```
>>> side_effect = lambda value: value + 1
>>> mock = Mock(side_effect=side_effect)
>>> mock(3)
4
>>> mock(-8)
-7
```

`Mock.`**`call_args`**

This is either `None` (if the mock hasn't been called), or the arguments that the mock was last called with. This will be in the form of a tuple: the first member is any ordered arguments the mock was called with (or an empty tuple) and the second member is any keyword arguments (or an empty dictionary).

```
>>> mock = Mock(return_value=None)
>>> print mock.call_args
None
>>> mock()
>>> mock.call_args
((), {})
>>> mock.call_args == ()
True
>>> mock(3, 4)
```

```
>>> mock.call_args
((3, 4), {})
>>> mock.call_args == ((3, 4),)
True
>>> mock(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args
((3, 4, 5), {'key': 'fish', 'next': 'w00t!'})
```

The tuple is lenient when comparing against tuples with empty elements skipped. This can make tests less verbose:

```
>>> mock = Mock(return_value=None)
>>> mock()
>>> mock.call_args == ()
True
```

Mock.**call_args_list**

This is a list of all the calls made to the mock object in sequence (so the length of the list is the number of times it has been called). Before any calls have been made it is an empty list. Its elements compare "softly" when positional arguments or keyword arguments are skipped:

```
>>> mock = Mock(return_value=None)
>>> mock()
>>> mock(3, 4)
>>> mock(key='fish', next='w00t!')
>>> mock.call_args_list
[((), {}), ((3, 4), {}), ((), {'key': 'fish', 'next': 'w00t!'})]
>>> expected = [(), ((3, 4),), ({'key': 'fish', 'next': 'w00t!'},)]
>>> mock.call_args_list == expected
True
```

Mock.**method_calls**

As well as tracking calls to themselves, mocks also track calls to methods and attributes, and *their* methods and attributes:

```
>>> mock = Mock()
>>> mock.method()
<mock.Mock object at 0x...>
>>> mock.property.method.attribute()
<mock.Mock object at 0x...>
>>> mock.method_calls
[('method', (), {}), ('property.method.attribute', (), {})]
```

The tuples in method_calls compare equal even if empty positional and keyword arguments are skipped.

```
>>> mock = Mock()
>>> mock.method()
<mock.Mock object at 0x...>
>>> mock.method(1, 2)
<mock.Mock object at 0x...>
>>> mock.method(a="b")
<mock.Mock object at 0x...>
>>> mock.method_calls == [('method',), ('method', (1, 2)),
... ('method', {"a": "b"})]
True
```

The `Mock` class has support for mocking magic methods. See *magic methods* for the full details.

1.2 Patch Decorators

The patch decorators are used for patching objects only within the scope of the function they decorate. They automatically handle the unpatching for you, even if exceptions are raised. All of these functions can also be used in with statements or as class decorators.

1.2.1 patch

Note: *patch* is straightforward to use. The key is to do the patching in the right namespace. See the section where to patch.

---

patch(*target*, *new=<SentinelObject "DEFAULT">*, *spec=None*, *create=False*, *mocksignature=False*, *spec_set=None*)

`patch` acts as a function decorator, class decorator or a context manager. Inside the body of the function or with statement, the `target` (specified in the form *'PackageName.ModuleName.ClassName'*) is patched with a `new` object. When the function/with statement exits the patch is undone.

The `target` is imported and the specified attribute patched with the new object, so it must be importable from the environment you are calling the decorator from.

If `new` is omitted, then a new `Mock` is created and passed in as an extra argument to the decorated function.

The `spec` and `spec_set` keyword arguments are passed to the `Mock` if patch is creating one for you.

In addition you can pass `spec=True` or `spec_set=True`, which causes patch to pass in the object being mocked as the spec/spec_set object.

If `mocksignature` is True then the patch will be done with a function created by mocking the one being replaced. If the object being replaced is a class then the signature of *__init__* will be copied. If the object being replaced is a callable object then the signature of *__call__* will be copied.

By default `patch` will fail to replace attributes that don't exist. If you pass in 'create=True' and the attribute doesn't exist, patch will create the attribute for you when the patched function is called, and delete it again afterwards. This is useful for writing tests against attributes that your production code creates at runtime. It is off by by default because it can be dangerous. With it switched on you can write passing tests against APIs that don't actually exist!

Patch can be used as a TestCase class decorator. It works by decorating each test method in the class. This reduces the boilerplate code when your test methods share a common patchings set.

Patch can be used with the with statement, if this is available in your version of Python. Here the patching applies to the indented block after the with statement. If you use "as" then the patched object will be bound to the name after the "as"; very useful if *patch* is creating a mock object for you.

patch.dict(...) and *patch.object(...)* are available for alternate use-cases.

Note: Patching a class replaces the class with a Mock *instance*. If the class is instantiated in the code under test then it will be the *return_value* of the mock that will be used.

If the class is instantiated multiple times you could use `Mock.side_effect` to return a new mock each time. Alternatively you can set the *return_value* to be anything you want.

To configure return values on methods of *instances* on the patched class you must do this on the *return_value*. For example:

```
>>> @patch('package.module.Class')
... def test(MockClass):
...     instance = MockClass.return_value
...     instance.method.return_value = 'foo'
...     from package.module import Class
...     assert Class() is instance
...     assert Class().method() == 'foo'
...
>>> test()
```

1.2.2 patch.object

patch.**object**(*target*, *attribute*, *new=DEFAULT*, *spec=None*, *create=False*, *mocksignature=False*, *spec_set=None*)

patch the named member (*attribute*) on an object (*target*) with a mock object.

Arguments new, spec, create, mocksignature and spec_set have the same meaning as for patch.

You can either call *patch.object* with three arguments or two arguments. The three argument form takes the object to be patched, the attribute name and the object to replace the attribute with.

When calling with the two argument form you omit the replacement object, and a mock is created for you and passed in as an extra argument to the decorated function:

```
>>> @patch.object(SomeClass, 'class_method')
... def test(mock_method):
...     SomeClass.class_method(3)
...     mock_method.assert_called_with(3)
...
>>> test()
```

`spec` and `create` have the same meaning as for the patch decorator.

`patch.object` is also a context manager and can be used with `with` statements in the same way as `patch`. It can also be used as a class decorator with same semantics as `patch`.

1.2.3 patch_object

Deprecated since version 0.7: This is the same as `patch.object`. Use the renamed version.

1.2.4 patch.dict

patch.**dict**(*in_dict*, *values=()*, *clear=False*)

Patch a dictionary and restore the dictionary to its original state after the test.

in_dict can be a dictionary or a mapping like container. If it is a mapping then it must at least support getting, setting and deleting items plus iterating over keys.

in_dict can also be a string specifying the name of the dictionary, which will then be fetched by importing it.

values can be a dictionary of values to set in the dictionary. *values* can also be an iterable of `(key, value)` pairs.

If *clear* is True then the dictionary will be cleared before the new values are set.

Like `patch()` and `patch.object()` `patch.dict` can be used as a decorator or a context manager. It can be used to add members to a dictionary, or simply let a test change a dictionary, and ensure the dictionary is restored when the test ends.

```
>>> from mock import patch
>>> foo = {}
>>> with patch.dict(foo, {'newkey': 'newvalue'}):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == {}

>>> import os
>>> with patch.dict('os.environ', {'newkey': 'newvalue'}):
...     print os.environ['newkey']
...
newvalue
>>> assert 'newkey' not in os.environ
```

1.2.5 patch methods: start and stop

All three patchers have *start* and *stop* methods. These make it simpler to do patching in *setUp* methods or where you want to do multiple patches without nesting decorators or with statements.

To use them call *patch*, *patch.object* or *patch.dict* as normal and keep a reference to the returned *patcher* object. You can then call *start* to put the patch in place and *stop* to undo it.

If you are using *patch* to create a mock for you then it will be returned by the call to *patcher.start*.

```
>>> patcher = patch('package.module.ClassName')
>>> from package import module
>>> original = module.ClassName
>>> new_mock = patcher.start()
>>> assert module.ClassName is not original
>>> assert module.ClassName is new_mock
>>> patcher.stop()
>>> assert module.ClassName is original
>>> assert module.ClassName is not new_mock
```

A typical use case for this might be for doing multiple patches in the *setUp* method of a *TestCase*:

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         self.patcher1 = patch('package.module.Class1')
...         self.patcher2 = patch('package.module.Class2')
...         self.MockClass1 = self.patcher1.start()
...         self.MockClass2 = self.patcher2.start()
...
```

```
...     def tearDown(self):
...         self.patcher1.stop()
...         self.patcher2.stop()
...
...     def test_something(self):
...         assert package.module.Class1 is self.MockClass1
...         assert package.module.Class2 is self.MockClass2
...
>>> MyTest('test_something').run()
```

Caution: If you use this technique you must ensure that the patching is "undone" by calling *stop*. This can be fiddlier than you might think, because if an exception is raised in the setUp then tearDown is not called. unittest2 cleanup functions make this easier.

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         patcher = patch('package.module.Class')
...         self.MockClass = patcher.start()
...         self.addCleanup(patcher.stop)
...
...     def test_something(self):
...         assert package.module.Class is self.MockClass
...
>>> MyTest('test_something').run()
```

As an added bonus you no longer need to keep a reference to the *patcher* object.

In fact *start* and *stop* are just aliases for the context manager *__enter__* and *__exit__* methods.

1.2.6 Nesting Patch Decorators

If you want to perform multiple patches then you can simply stack up the decorators.

You can stack up multiple patch decorators using this pattern:

```
>>> @patch.object(SomeClass, 'class_method')
... @patch.object(SomeClass, 'static_method')
... def test(mock1, mock2):
...     assert SomeClass.static_method is mock1
...     assert SomeClass.class_method is mock2
...     SomeClass.static_method('foo')
...     SomeClass.class_method('bar')
...     return mock1, mock2
...
>>> mock1, mock2 = test()
>>> mock1.assert_called_once_with('foo')
```

```
>>> mock2.assert_called_once_with('bar')
```

Note that the decorators are applied from the bottom upwards. This is the standard way that Python applies decorators. The order of the created mocks passed into your test function matches this order.

Like all context-managers patches can be nested using contextlib's nested function; *every* patching will appear in the tuple after "as":

```
>>> from contextlib import nested
>>> with nested(
...         patch('package.module.ClassName1'),
...         patch('package.module.ClassName2')
...     ) as (MockClass1, MockClass2):
...     assert package.module.ClassName1 is MockClass1
...     assert package.module.ClassName2 is MockClass2
...
```

1.2.7 Where to patch

patch works by (temporarily) changing the object that a *name* points to with another one. There can be many names pointing to any individual object, so for patching to work you must ensure that you patch the name used by the system under test.

The basic principle is that you patch where an object is *used*, which is not necessarily the same place as where it is defined. A couple of examples will help to clarify this.

Imagine we have a project that we want to test with the following structure:

```
a.py
    -> Defines SomeClass

b.py
    -> from a import SomeClass
    -> some_function instantiates SomeClass
```

Now we want to test *some_function* but we want to mock out *SomeClass* using *patch*. The problem is that when we import module b, which we will have to do then it imports *SomeClass* from module a. If we use *patch* to mock out *a.SomeClass* then it will have no effect on our test; module b already has a reference to the *real SomeClass* and it looks like our patching had no effect.

The key is to patch out *SomeClass* where it is used (or where it is looked up). In this case *some_function* will actually look up *SomeClass* in module b, where we have imported it. The patching should look like:

```
    @patch('b.SomeClass')
```

However, consider the alternative scenario where instead of *from a import SomeClass* module b does *import a* and *some_function* uses *a.SomeClass*. Both of these import forms are common. In this case the class we want to patch is being looked up on the a module and so we have to patch *a.SomeClass* instead:

```
@patch('a.SomeClass')
```

1.2.8 Patching Descriptors and Proxy Objects

Since version 0.6.0 both patch and patch.object have been able to correctly patch and restore descriptors: class methods, static methods and properties. You should patch these on the *class* rather than an instance.

Since version 0.7.0 patch and patch.object work correctly with some objects that proxy attribute access, like the django setttings object.

Note: In django *import settings* and *from django.conf import settings* return different objects. If you are using libraries / apps that do both you may have to patch both. Grrr...

1.3 Sentinel

sentinel

The `sentinel` object provides a convenient way of providing unique objects for your tests.

Attributes are created on demand when you access them by name. Accessing the same attribute will always return the same object. The objects returned have a sensible repr so that test failure messages are readable.

DEFAULT

The `DEFAULT` object is a pre-created sentinel (actually `sentinel.DEFAULT`). It can be used by `Mock.side_effect` functions to indicate that the normal return value should be used.

1.3.1 Sentinel Example

Sometimes when testing you need to test that a specific object is passed as an argument to another method, or returned. It can be common to create named sentinel objects to test this. *sentinel* provides a convenient way of creating and testing the identity of objects like this.

In this example we monkey patch *method* to return *sentinel.some_object*:

```
>>> real = ProductionClass()
>>> real.method = Mock(name="method")
>>> real.method.return_value = sentinel.some_object
>>> result = real.method()
>>> assert result is sentinel.some_object
>>> sentinel.some_object
<SentinelObject "some_object">
```

1.4 Mocking Magic Methods

`Mock` supports mocking magic methods. This allows mock objects to replace containers or other objects that implement Python protocols.

Because magic methods are looked up differently from normal methods [2], this support has been specially implemented. This means that only specific magic methods are supported. The supported list includes *almost* all of them. If there are any missing that you need please let us know!

You mock magic methods by setting the method you are interested in to a function or a mock instance. If you are using a function then it *must* take `self` as the first argument [3].

```
>>> def __str__(self):
...     return 'fooble'
...
>>> mock = Mock()
>>> mock.__str__ = __str__
>>> str(mock)
'fooble'

>>> mock = Mock()
>>> mock.__str__ = Mock()
>>> mock.__str__.return_value = 'fooble'
>>> str(mock)
'fooble'

>>> mock = Mock()
>>> mock.__iter__ = Mock(return_value=iter([]))
>>> list(mock)
[]
```

One use case for this is for mocking objects used as context managers in a `with` statement:

[2] Magic methods *should* be looked up on the class rather than the instance. Different versions of Python are inconsistent about applying this rule. The supported protocol methods should work with all supported versions of Python.

[3] The function is basically hooked up to the class, but each `Mock` instance is kept isolated from the others.

```
>>> mock = Mock()
>>> mock.__enter__ = Mock()
>>> mock.__exit__ = Mock()
>>> mock.__exit__.return_value = False
>>> with mock:
...     pass
...
>>> mock.__enter__.assert_called_with()
>>> mock.__exit__.assert_called_with(None, None, None)
```

Calls to magic methods do not (yet) appear in `Mock.method_calls`. This may change in a future release.

Note: If you use the `spec` keyword argument to create a mock then attempting to set a magic method that isn't in the spec will raise an AttributeError.

The full list of supported magic methods is:

- `__hash__`, `__sizeof__`, `__repr__` and `__str__`
- `__dir__`, `__format__` and `__subclasses__`
- `__floor__`, `__trunc__` and `__ceil__`
- Comparisons: `__cmp__`, `__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__` and `__ne__`
- Container methods: `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`, `__len__`, `__iter__`, `__getslice__`, `__setslice__`, `__reversed__` and `__missing__`
- Context manager: `__enter__` and `__exit__`
- Unary numeric methods: `__neg__`, `__pos__` and `__invert__`
- The numeric methods (including right hand and in-place variants): `__add__`, `__sub__`, `__mul__`, `__div__`, `__truediv__`, `__floordiv__`, `__mod__`, `__divmod__`, `__lshift__`, `__rshift__`, `__and__`, `__xor__`, `__or__`, and `__pow__`
- Numeric conversion methods: `__complex__`, `__int__`, `__float__`, `__index__` and `__coerce__`
- Descriptor methods: `__get__`, `__set__` and `__delete__`
- Pickling: `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` and `__setstate__`

The following methods are supported in Python 2 but don't exist in Python 3:

- `__unicode__`, `__long__`, `__oct__`, `__hex__` and `__nonzero__`

The following methods are supported in Python 3 but don't exist in Python 2:

- `__bool__` and `__next__`

The following methods exist but are *not* supported as they are either in use by mock, can't be set dynamically, or can cause problems:

- `__getattr__`, `__setattr__`, `__init__` and `__new__`
- `__prepare__`, `__instancecheck__`, `__subclasscheck__`, `__del__`

1.5 Magic Mock

class `MagicMock` (**args*, ***kw*)

`MagicMock` is a subclass of `Mock` with default implementations of most of the magic methods. You can use `MagicMock` without having to configure the magic methods yourself.

If you use the `spec` argument then *only* magic methods that exist in the spec will be created.

The magic methods are setup with `Mock` objects, so you can configure them and use them in the usual way:

```
>>> mock = MagicMock()
>>> mock[3] = 'fish'
>>> mock.__setitem__.assert_called_with(3, 'fish')
>>> mock.__getitem__.return_value = 'result'
>>> mock[2]
'result'
```

By default many of the protocol methods are required to return objects of a specific type. These methods are preconfigured with a default return value, so that they can be used without you having to do anything if you aren't interested in the return value. You can still *set* the return value manually if you want to change the default.

Methods and their defaults:

- `__int__` : 1
- `__contains__` : False
- `__len__` : 1
- `__iter__` : iter([])
- `__exit__` : False
- `__complex__` : 1j
- `__float__` : 1.0
- `__bool__` : True
- `__nonzero__` : True

- __oct__ : '1'
- __hex__ : '0x1'
- __long__ : long(1)
- __index__ : 1
- __hash__ : default hash for the mock
- __str__ : default str for the mock
- __unicode__ : default unicode for the mock
- __sizeof__: default sizeof for the mock

For example:

```
>>> mock = MagicMock()
>>> int(mock)
1
>>> len(mock)
0
>>> hex(mock)
'0x1'
>>> list(mock)
[]
>>> object() in mock
False
```

The two equality method, *__eq__* and *__ne__*, are special (changed in 0.7.2). They do the default equality comparison on identity, using a side effect, unless you change their return value to return something else:

```
>>> MagicMock() == 3
False
>>> MagicMock() != 3
True
>>> mock = MagicMock()
>>> mock.__eq__.return_value = True
>>> mock == 3
True
```

`MagicMock` has all of the supported magic methods configured except for some of the obscure and obsolete ones. You can still set these up if you want.

Magic methods that are supported but not setup by default in `MagicMock` are:

- __cmp__
- __getslice__ and __setslice__
- __coerce__

- `__subclasses__`
- `__dir__`
- `__format__`
- `__get__`, `__set__` and `__delete__`
- `__reversed__` and `__missing__`
- `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` and `__setstate__`
- `__getformat__` and `__setformat__`

1.6 mocksignature

A problem with using mock objects to replace real objects in your tests is that `Mock` can be *too* flexible. Your code can treat the mock objects in any way and you have to manually check that they were called correctly. If your code calls functions or methods with the wrong number of arguments then mocks don't complain.

The solution to this is `mocksignature`, which creates functions with the same signature as the original, but delegating to a mock. You can interrogate the mock in the usual way to check it has been called with the *right* arguments, but if it is called with the wrong number of arguments it will raise a `TypeError` in the same way your production code would.

Another advantage is that your mocked objects are real functions, which can be useful when your code uses inspect or depends on functions being functions.

mocksignature(*func*, *mock=None*, *skipfirst=False*)

Create a new function with the same signature as *func* that delegates to *mock*. If *skipfirst* is True the first argument is skipped, useful for methods where *self* needs to be omitted from the new function.

If you don't pass in a *mock* then one will be created for you.

The mock is set as the *mock* attribute of the returned function for easy access.

mocksignature can also be used with classes. It copies the signature of the *__init__* method.

When used with callable objects (instances) it copies the signature of the *__call__* method.

`mocksignature` will work out if it is mocking the signature of a method on an instance or a method on a class and do the "right thing" with the `self` argument in both cases.

Because of a limitation in the way that arguments are collected by functions created by `mocksignature` they are *always* passed as positional arguments (including defaults) and not keyword arguments.

1.6.1 Example use

Basic use

```
>>> from mock import mocksignature, Mock
>>> def function(a, b, c=None):
...     pass
...
>>> mock = Mock()
>>> function = mocksignature(function, mock)
>>> function()
Traceback (most recent call last):
  ...
TypeError: <lambda>() takes at least 2 arguments (0 given)
>>> mock.return_value = 'some value'
>>> function(1, 2, 'foo')
'some value'
>>> function.mock.assert_called_with(1, 2, 'foo')
```

Keyword arguments

Note that arguments to functions created by `mocksignature` are always passed in to the underlying mock by position even when called with keywords:

```
>>> from mock import mocksignature
>>> def function(a, b, c=None):
...     pass
...
>>> function = mocksignature(function)
>>> function.mock.return_value = None
>>> function(1, 2)
>>> function.mock.assert_called_with(1, 2, None)
```

Mocking methods and self

When you use `mocksignature` to replace a method on a class then `self` will be included in the method signature - and you will need to include the instance when you do your asserts:

```
>>> from mock import mocksignature
>>> class SomeClass(object):
...     def method(self, a, b, c=None):
...         pass
...
>>> SomeClass.method = mocksignature(SomeClass.method)
>>> SomeClass.method.mock.return_value = None
```

```
>>> instance = SomeClass()
>>> instance.method()
Traceback (most recent call last):
  ...
TypeError: <lambda>() takes at least 4 arguments (1 given)
>>> instance.method(1, 2, 3)
>>> instance.method.mock.assert_called_with(instance, 1, 2, 3)
```

When you use `mocksignature` on instance methods `self` isn't included:

```
>>> from mock import mocksignature
>>> class SomeClass(object):
...     def method(self, a, b, c=None):
...         pass
...
>>> instance = SomeClass()
>>> instance.method = mocksignature(instance.method)
>>> instance.method.mock.return_value = None
>>> instance.method(1, 2, 3)
>>> instance.method.mock.assert_called_with(1, 2, 3)
```

mocksignature with classes

When used with a class *mocksignature* copies the signature of the *__init__* method.

```
>>> from mock import mocksignature
>>> class Something(object):
...     def __init__(self, foo, bar):
...         pass
...
>>> MockSomething = mocksignature(Something)
>>> instance = MockSomething(10, 9)
>>> assert instance is MockSomething.mock.return_value
>>> MockSomething.mock.assert_called_with(10, 9)
>>> MockSomething()
Traceback (most recent call last):
  ...
TypeError: <lambda>() takes at least 2 arguments (0 given)
```

Because the object returned by *mocksignature* is a function rather than a *Mock* you lose the other capabilities of *Mock*, like dynamic attribute creation.

mocksignature with callable objects

When used with a callable object *mocksignature* copies the signature of the *__call__* method.

```
>>> from mock import mocksignature
>>> class Something(object):
...     def __call__(self, spam, eggs):
...         pass
...
>>> something = Something()
>>> mock_something = mocksignature(something)
>>> result = mock_something(10, 9)
>>> mock_something.mock.assert_called_with(10, 9)
>>> mock_something()
Traceback (most recent call last):
  ...
TypeError: <lambda>() takes at least 2 arguments (0 given)
```

Because the object returned by *mocksignature* is a function rather than a *Mock* you lose the other capabilities of *Mock*, like dynamic attribute creation.

1.6.2 mocksignature argument to patch

`mocksignature` is available as a keyword argument to `patch()` or `patch.object()`. It can be used with functions / methods / classes and callable objects.

```
>>> from mock import patch
>>> class SomeClass(object):
...     def method(self, a, b, c=None):
...         pass
...
>>> @patch.object(SomeClass, 'method', mocksignature=True)
... def test(mock_method):
...     instance = SomeClass()
...     mock_method.return_value = None
...     instance.method(1, 2)
...     mock_method.assert_called_with(instance, 1, 2, None)
...
>>> test()
```

CHAPTER

TWO

USER GUIDE

2.1 Getting Started with Mock

For comprehensive examples, see the unit tests included in the full source distribution.

2.1.1 Using Mock

Mock Patching Methods

`Mock` objects can be used for:

- Patching methods
- Recording method calls on objects

```
>>> from mock import Mock
>>> real = SomeClass()
>>> real.method = Mock()
>>> real.method(3, 4, 5, key='value')
<mock.Mock object at 0x...>
```

Once the mock has been used it has methods and attributes that allow you to make assertions about how it has been used.

Mock objects are callable. If they are called then the `called` attribute is set to *True*.

This example tests that calling `method` results in a call to `something`:

```
>>> from mock import Mock
>>> class ProductionClass(object):
...     def method(self):
...         self.something()
...     def something(self):
...         pass
```

```
...
>>> real = ProductionClass()
>>> real.something = Mock()
>>> real.method()
>>> real.something.called
True
```

If you want access to the actual arguments the mock was called with, for example to make assertions about the arguments themselves, then this information is available.

```
>>> real = ProductionClass()
>>> real.something = Mock()
>>> real.method()
>>> real.something.call_count
1
>>> args, kwargs = (), {}
>>> assert real.something.call_args == (args, kwargs)
>>> assert real.something.call_args_list == [(args, kwargs)]
```

Checking `call_args_list` tests how many times the mock was called, and the arguments for each call, in a single assertion.

From 0.7.0 you can omit empty args and keyword args, which makes comparisons less verbose:

```
>>> real = ProductionClass()
>>> real.something = Mock()
>>> real.method()
>>> assert real.something.call_args == ()
>>> assert real.something.call_args_list == [()]
```

If the mock has only been called once then you can use `Mock.assert_called_once_with()`:

```
>>> real = ProductionClass()
>>> real.something = Mock()
>>> real.method()
>>> real.something.assert_called_once_with()
```

If you don't care how many times an object has been called, but are just interested in the most recent call, then you can use `Mock.assert_called_with()`:

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, 3)
>>> mock.assert_called_with(1, 2, 3)
```

Mock for Method Calls on an Object

```
>>> class ProductionClass(object):
...     def closer(self, something):
...         something.close()
...
>>> real = ProductionClass()
>>> mock = Mock()
>>> real.closer(mock)
>>> mock.close.assert_called_with()
```

We don't have to do any work to provide the 'close' method on our mock. Accessing close creates it. So, if 'close' hasn't already been called then accessing it in the test will create it, but `Mock.assert_called_with()` will raise a failure exception.

As `close` is a mock object is has all the attributes from the previous example.

Mocking Classes

A common use case is to mock out classes instantiated by your code under test. When you patch a class, then that class is replaced with a mock. Instances are created by *calling the class*. This means you access the "mock instance" by looking at the return value of the mocked class.

In the example below we have a function *some_function* that instantiates *Foo* and calls a method on it. We can mock out the class *Foo*, and configure the behaviour of the *Foo* instance by configuring the *mock.return_value*.

```
>>> def some_function():
...     instance = module.Foo()
...     return instance.method()
...
>>> with patch('module.Foo') as mock:
...     instance = mock.return_value
...     instance.method.return_value = 'the result'
...     result = some_function()
...     assert result == 'the result'
```

Naming your mocks

It can be useful to give your mocks a name. The name is shown in the repr of the mock and can be helpful when the mock appears in test failure messages. The name is also propagated to attributes or methods of the mock:

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

```
>>> mock.method
<Mock name='foo.method' id='...'>
```

Limiting Available Methods

The disadvantage of the approach above is that *all* method access creates a new mock. This means that you can't tell if any methods were called that shouldn't have been. There are two ways round this. The first is by restricting the methods available on your mock.

```
>>> mock = Mock(spec=['close'])
>>> real.closer(mock)
>>> mock.close.assert_called_with()
>>> mock.foo
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'foo'
```

If `closer` calls any methods on `mock` *other* than close, then an `AttributeError` will be raised.

When you use *spec* it is still possible to set arbitrary attributes on the mock object. For a stronger form that only allows you to *set* attributes that are in the spec you can use *spec_set* instead:

```
>>> mock = Mock(spec=['close'])
>>> mock.foo = 3
>>> mock.foo
3
>>> mock = Mock(spec_set=['close'])
>>> mock.foo = 3
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'foo'
```

Mock objects that use a class or an instance as a *spec* or *spec_set* are able to pass *isintance* tests:

```
>>> mock = Mock(spec=SomeClass)
>>> isinstance(mock, SomeClass)
True
>>> mock = Mock(spec_set=SomeClass())
>>> isinstance(mock, SomeClass)
True
```

Tracking all Method Calls

An alternative way to verify that only the expected methods have been accessed is to use the `method_calls` attribute of the mock. This records all calls to child attributes of the mock - and

also to their children.

This is useful if you have a mock where you expect an attribute method to be called. You could access the attribute directly, but `method_calls` provides a convenient way of looking at all method calls:

```
>>> mock = Mock()
>>> mock.method()
<mock.Mock object at 0x...>
>>> mock.Property.method(10, x=53)
<mock.Mock object at 0x...>
>>> mock.method_calls
[('method', (), {}), ('Property.method', (10,), {'x': 53})]
```

If you make an assertion about `method_calls` and any unexpected methods have been called, then the assertion will fail.

Again, from 0.7.0, empty arguments and keyword arguments can be omitted for less verbose comparisons:

```
>>> mock = Mock()
>>> mock.method1()
<mock.Mock object at 0x...>
>>> mock.method2()
<mock.Mock object at 0x...>
>>> assert mock.method_calls == [('method1',), ('method2',)]
```

Setting Return Values and Attributes

Setting the return values on a mock object is trivially easy:

```
>>> mock = Mock()
>>> mock.return_value = 3
>>> mock()
3
```

Of course you can do the same for methods on the mock:

```
>>> mock = Mock()
>>> mock.method.return_value = 3
>>> mock.method()
3
```

The return value can also be set in the constructor:

```
>>> mock = Mock(return_value=3)
>>> mock()
3
```

If you need an attribute setting on your mock, just do it:

```
>>> mock = Mock()
>>> mock.x = 3
>>> mock.x
3
```

Sometimes you want to mock up a more complex situation, like for example `mock.connection.cursor().execute("SELECT 1")`:

```
>>> mock = Mock()
>>> cursor = mock.connection.cursor.return_value
>>> cursor.execute.return_value = None
>>> mock.connection.cursor().execute("SELECT 1")
>>> mock.method_calls
[('connection.cursor', (), {})]
>>> cursor.method_calls
[('execute', ('SELECT 1',), {})]
```

Creating a Mock from an Existing Object

One problem with over use of mocking is that it couples your tests to the implementation of your mocks rather than your real code. Suppose you have a class that implements `some_method`. In a test for another class, you provide a mock of this object that *also* provides `some_method`. If later you refactor the first class, so that it no longer has `some_method` - then your tests will continue to pass even though your code is now broken!

`Mock` allows you to provide an object as a specification for the mock, using the `spec` keyword argument. Accessing methods / attributes on the mock that don't exist on your specification object will immediately raise an attribute error. If you change the implementation of your specification, then tests that use that class will start failing immediately without you having to instantiate the class in those tests.

```
>>> mock = Mock(spec=SomeClass)
>>> mock.old_method()
Traceback (most recent call last):
   ...
AttributeError: object has no attribute 'old_method'
```

Again, if you want a stronger form of specification that prevents the setting of arbitrary attributes as well as the getting of them then you can use *spec_set* instead of *spec*.

Raising exceptions with mocks

A useful attribute is *side_effect*. If you set this to an exception class or instance then the exception will be raised when the mock is called. If you set it to a callable then it will be called whenever

the mock is called. This allows you to do things like return members of a sequence from repeated calls:

```
>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!

>>> results = [1, 2, 3]
>>> def side_effect(*args, **kwargs):
...     return results.pop()
...
>>> mock.side_effect = side_effect
>>> mock(), mock(), mock()
(3, 2, 1)
```

2.1.2 Sentinel

`sentinel` is a useful object for providing unique objects in your tests:

```
>>> from mock import sentinel
>>> real = SomeClass()
>>> real.method = Mock()
>>> real.method.return_value = sentinel.return_value
>>> real.method()
<SentinelObject "return_value">
```

2.1.3 Patch Decorators

Note: With *patch* it matters that you patch objects in the namespace where they are looked up. This is normally straightforward, but for a quick guide read *where to patch*.

A common need in tests is to patch a class attribute or a module attribute, for example patching a builtin or patching a class in a module to test that it is instantiated. Modules and classes are effectively global, so patching on them has to be undone after the test or the patch will persist into other tests and cause hard to diagnose problems.

mock provides three convenient decorators for this: *patch*, *patch.object* and *patch.dict*. *patch* takes a single string, of the form *package.module.Class.attribute* to specify the attribute you are patching. It also optionally takes a value that you want the attribute (or class or whatever) to be replaced with. 'patch.object' takes an object and the name of the attribute you would like patched, plus optionally the value to patch it with.

patch.object:

```
>>> original = SomeClass.attribute
>>> @patch.object(SomeClass, 'attribute', sentinel.attribute)
... def test():
...     assert SomeClass.attribute == sentinel.attribute
...
>>> test()
>>> assert SomeClass.attribute == original

>>> @patch('package.module.attribute', sentinel.attribute)
... def test():
...     from package.module import attribute
...     assert attribute is sentinel.attribute
...
>>> test()
```

If you are patching a module (including `__builtin__`) then use `patch` instead of `patch.object`:

```
>>> mock = Mock()
>>> mock.return_value = sentinel.file_handle
>>> @patch('__builtin__.open', mock)
... def test():
...     return open('filename', 'r')
...
>>> handle = test()
>>> mock.assert_called_with('filename', 'r')
>>> assert handle == sentinel.file_handle, "incorrect file handle returned"
```

The module name can be 'dotted', in the form `package.module` if needed:

```
>>> @patch('package.module.ClassName.attribute', sentinel.attribute)
... def test():
...     from package.module import ClassName
...     assert ClassName.attribute == sentinel.attribute
...
>>> test()
```

If you don't want to call the decorated test function yourself, you can add *apply* as a decorator on top, this calls it immediately.

A nice pattern is to actually decorate test methods themselves:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'attribute', sentinel.attribute)
...     def test_something(self):
...         self.assertEqual(SomeClass.attribute, sentinel.attribute)
...
```

```
>>> original = SomeClass.attribute
>>> MyTest('test_something').test_something()
>>> assert SomeClass.attribute == original
```

If you want to patch with a Mock, you can use *patch* with only one argument (or `patch.object` with two arguments). The mock will be created for you and passed into the test function / method:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'static_method')
...     def test_something(self, mock_method):
...         SomeClass.static_method()
...         mock_method.assert_called_with()
...
>>> MyTest('test_something').test_something()
```

You can stack up multiple patch decorators using this pattern:

```
>>> class MyTest(unittest.TestCase):
...     @patch('package.module.ClassName1')
...     @patch('package.module.ClassName2')
...     def test_something(self, MockClass2, MockClass1):
...         self.assertTrue(package.module.ClassName1 is MockClass1)
...         self.assertTrue(package.module.ClassName2 is MockClass2)
...
>>> MyTest('test_something').test_something()
```

When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal *python* order that decorators are applied). This means from the bottom up, so in the example above the mock for *test_module.ClassName2* is passed in first.

There is also `patch.dict()` for setting values in a dictionary just during a scope and restoring the dictionary to its original state when the test ends:

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

patch, *patch.object* and *patch.dict* can all be used as context managers.

Where you use *patch* to create a mock for you, you can get a reference to the mock using the "as" form of the with statement:

```
>>> class ProductionClass(object):
...     def method(self):
...         pass
...
```

```
>>> with patch.object(ProductionClass, 'method') as mock_method:
...     mock_method.return_value = None
...     real = ProductionClass()
...     real.method(1, 2, 3)
...
>>> mock_method.assert_called_with(1, 2, 3)
```

As an alternative *patch*, *patch.object* and *patch.dict* can be used as class decorators. When used in this way it is the same as applying the decorator indvidually to every method whose name starts with "test".

2.2 Further Examples

Here are some more examples for some slightly more advanced scenarios than in the *getting started* guide.

2.2.1 Mocking chained calls

Mocking chained calls is actually straightforward with mock once you understand the `Mock.return_value` attribute. When a mock is called for the first time, or you fetch its *return_value* before it has been called, a new *Mock* is created.

This means that you can see how the object returned from a call to a mocked object has been used by interrogating the *return_value* mock:

```
>>> mock = Mock()
>>> mock().foo(a=2, b=3)
<mock.Mock object at 0x...>
>>> mock.return_value.foo.assert_called_with(a=2, b=3)
```

From here it is a simple step to configure and then make assertions about chained calls. Of course another alternative is writing your code in a more testable way in the first place...

So, suppose we have some code that looks a little bit like this:

```
>>> class Something(object):
...     def __init__(self):
...         self.backend = BackendProvider()
...     def method(self):
...         response = self.backend.get_endpoint('foobar').create_call('spam', 'egg
...         # more code
```

Assuming that *BackendProvider* is already well tested, how do we test *method()*? Specifically, we want to test that the code section `# more code` uses the response object in the correct way.

As this chain of calls is made from an instance attribute we can monkey patch the *backend* attribute on a *Something* instance. In this particular case we are only interested in the return value from the final call to *start_call* so we don't have much configuration to do. Let's assume the object it returns is 'file-like', so we'll ensure that our response object uses the builtin `file` as its *spec*.

To do this we create a mock instance as our mock backend and create a mock response object for it. To set the response as the return value for that final *start_call* we could do this:

```
mock_backend.get_endpoint.return_value.create_call.return_value.start_c
= mock_response.
```

Here's how we might do it in a slightly nicer way. We start by creating our initial mocks:

```
>>> something = Something()
>>> mock_response = Mock(spec=file)
>>> mock_backend = Mock()
>>> get_endpoint = mock_backend.get_endpoint
>>> create_call = get_endpoint.return_value.create_call
>>> start_call = create_call.return_value.start_call
>>> start_call.return_value = mock_response
```

With these we monkey patch the "mock backend" in place and can make the real call:

```
>>> something.backend = mock_backend
>>> something.method()
```

Keeping references to the intermediate methods makes our assertions easier, and also makes the code less ugly.

```
>>> get_endpoint.assert_called_with('foobar')
>>> create_call.assert_called_with('spam', 'eggs')
>>> start_call.assert_called_with()
>>> # make assertions on mock_response about how it is used
```

2.2.2 Partial mocking

In some tests I wanted to mock out a call to datetime.date.today() to return a known date, but I didn't want to prevent the code under test from creating new date objects. Unfortunately *datetime.date* is written in C, and so I couldn't just monkey-patch out the static *date.today* method.

I found a simple way of doing this that involved effectively wrapping the date class with a mock, but passing through calls to the constructor to the real class (and returning real instances).

The `patch decorator` is used here to mock out the *date* class in the module under test. The `side_effect` attribute on the mock date class is then set to a lambda function that returns a real date. When the mock date class is called a real date will be constructed and returned by *side_effect*.

```
>>> from datetime import date
>>> with patch('mymodule.date') as mock_date:
...     mock_date.today.return_value = date(2010, 10, 8)
...     mock_date.side_effect = lambda *args, **kw: date(*args, **kw)
...
...     assert mymodule.date.today() == date(2010, 10, 8)
...     assert mymodule.date(2009, 6, 8) == date(2009, 6, 8)
...
```

Note that we don't patch *datetime.date* globally, we patch *date* in the module that *uses* it. See *where to patch*.

When *date.today()* is called a known date is returned, but calls to the *date(...)* constructor still return normal dates. Without this you can find yourself having to calculate an expected result using exactly the same algorithm as the code under test, which is a classic testing anti-pattern.

Calls to the date constructor are recorded in the *mock_date* attributes (*call_count* and friends) which may also be useful for your tests.

2.2.3 Mocking open

Using *open* as a context manager is a great way to ensure your file handles are closed properly and is becoming common:

```
with open('/some/path', 'w') as f:
    f.write('something')
```

The issue is that even if you mock out the call to *open* it is the *returned object* that is used as a context manager (and has *__enter__* and *__exit__* called).

So first the topic of creating a mock object that can be called, with the return value able to act as a context manager. The easiest way of doing this is to use the new `MagicMock`, which is preconfigured to be able to act as a context manger. As an added bonus we'll use the spec argument to ensure that the mocked object can only be used in the same ways a real file could be used (attempting to access a method or attribute not on the *file* will raise an `AttributeError`):

```
>>> mock_open = Mock()
>>> mock_open.return_value = MagicMock(spec=file)
```

In terms of configuring our mock this is all that needs to be done. In fact it could be constructed with a one liner: `mock_open = Mock(return_value=MagicMock(spec=file))`.

So what is the best way of patching the builtin `open` function? One way would be to globally patch `__builtin__.open`. So long as you are sure that none of the other code being called also accesses `open` this is perfectly reasonable. It does make some people nervous however. By default we can't patch the `open` name in the module where it is used, because `open` doesn't exist as an attribute in that namespace. `patch` refuses to patch attributes that don't exist because that

is a great way of having tests that pass but code that is horribly broken (your code can access attributes that only exist during your tests!). `patch` *will* however create (and then remove again) non-existent attributes if you tell it that you are really sure you know what you're doing.

By passing `create=True` into `patch` we can just patch the `open` function in the module under test instead of patching it globally:

```
>>> open_name = '%s.open' % __name__
>>> with patch(open_name, create=True) as mock_open:
...     mock_open.return_value = MagicMock(spec=file)
...
...     with open('/some/path', 'w') as f:
...         f.write('something')
...
<mock.Mock object at 0x...>
>>> file_handle = mock_open.return_value.__enter__.return_value
>>> file_handle.write.assert_called_with('something')
```

2.2.4 Applying the same patch to every test method

If you want several patches in place for multiple test methods the obvious way is to apply the patch decorators to every method. This can feel like unnecessary repetition. For Python 2.6 or more recent you can use *patch* (in all its various forms) as a class decorator. This applies the patches to all test methods on the class. A test method is identified by methods whose names start with *test*:

```
>>> @patch('mymodule.SomeClass')
... class MyTest(TestCase):
...
...     def test_one(self, MockSomeClass):
...         self.assertTrue(mymodule.SomeClass is MockSomeClass)
...
...     def test_two(self, MockSomeClass):
...         self.assertTrue(mymodule.SomeClass is MockSomeClass)
...
...     def not_a_test(self):
...         return 'something'
...
>>> MyTest('test_one').test_one()
>>> MyTest('test_two').test_two()
>>> MyTest('test_two').not_a_test()
'something'
```

An alternative way of managing patches is to use the *patch methods: start and stop*. These allow you to move the patching into your *setUp* and *tearDown* methods.

```
>>> class MyTest(TestCase):
...     def setUp(self):
```

```
...         self.patcher = patch('mymodule.foo')
...         self.mock_foo = self.patcher.start()
...
...     def test_foo(self):
...         self.assertTrue(mymodule.foo is self.mock_foo)
...
...     def tearDown(self):
...         self.patcher.stop()
...
>>> MyTest('test_foo').run()
```

If you use this technique you must ensure that the patching is "undone" by calling *stop*. This can be fiddlier than you might think, because if an exception is raised in the setUp then tearDown is not called. unittest2 cleanup functions make this simpler:

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         patcher = patch('mymodule.foo')
...         self.addCleanup(patcher.stop)
...         self.mock_foo = patcher.start()
...
...     def test_foo(self):
...         self.assertTrue(mymodule.foo is self.mock_foo)
...
>>> MyTest('test_foo').run()
```

2.2.5 Mocking Unbound Methods

Whilst writing tests today I needed to patch an *unbound method* (patching the method on the class rather than on the instance). I needed self to be passed in as the first argument because I want to make asserts about which objects were calling this particular method. The issue is that you can't patch with a mock for this, because if you replace an unbound method with a mock it doesn't become a bound method when fetched from the instance, and so it doesn't get self passed in. The workaround is to patch the unbound method with a real function instead. The `patch()` decorator makes it so simple to patch out methods with a mock that having to create a real function becomes a nuisance.

If you pass `mocksignature=True` to patch then it does the patching with a *real* function object. This function object has the same signature as the one it is replacing, but delegates to a mock under the hood. You still get your mock auto-created in exactly the same way as before. What it means though, is that if you use it to patch out an unbound method on a class the mocked function will be turned into a bound method if it is fetched from an instance. It will have *self* passed in as the first argument, which is exactly what I wanted:

```
>>> class Foo(object):
...   def foo(self):
...     pass
...
>>> with patch.object(Foo, 'foo', mocksignature=True) as mock_foo:
...   mock_foo.return_value = 'foo'
...   foo = Foo()
...   foo.foo()
...
'foo'
>>> mock_foo.assert_called_once_with(foo)
```

If we don't use *mocksignature=True* then the unbound method is patched out with a Mock instance instead, and isn't called with *self*.

2.2.6 Mocking Properties

A few people have asked about mocking properties, specifically tracking when properties are fetched from objects or even having side effects when properties are fetched.

You can already do this by subclassing `Mock` and providing your own property. Delegating to another mock is one way to record the property being accessed whilst still able to control things like return values:

```
>>> mock_foo = Mock(return_value='fish')
>>> class MyMock(Mock):
...     @property
...     def foo(self):
...         return mock_foo()
...
>>> mock = MyMock()
>>> mock.foo
'fish'
>>> mock_foo.assert_called_once_with()
```

2.2.7 Checking multiple calls with mock

mock has a nice API for making assertions about how your mock objects are used.

```
>>> mock = Mock()
>>> mock.foo_bar.return_value = None
>>> mock.foo_bar('baz', spam='eggs')
>>> mock.foo_bar.assert_called_with('baz', spam='eggs')
```

If your mock is only being called once you can use the `assert_called_once_with()` method that also asserts that the `call_count` is one.

```
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
>>> mock.foo_bar()
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
Traceback (most recent call last):
    ...
AssertionError: Expected to be called once. Called 2 times.
```

Both *assert_called_with* and *assert_called_once_with* make assertions about the *most recent* call. If your mock is going to be called several times, and you want to make assertions about *all* those calls, the API is not quite so nice.

All of the calls, in order, are stored in *call_args_list* as tuples of (positional args, keyword args).

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, 3)
>>> mock(4, 5, 6)
>>> mock()
>>> mock.call_args_list
[((1, 2, 3), {}), ((4, 5, 6), {}), ((), {})]
```

Because it stores positional args *and* keyword args, even if they are empty, the list is overly verbose which makes for ugly tests. It turns out that I do this rarely enough that I've never got around to improving it. One of the new features in 0.7.0 helps with this. The tuples of (positional, keyword) arguments are now custom objects that allow for 'soft comparisons' (implemented by Konrad Delong). This allows you to omit empty positional or keyword arguments from tuples you compare against.

```
>>> mock.call_args_list
[((1, 2, 3), {}), ((4, 5, 6), {}), ((), {})]
>>> expected = [((1, 2, 3),), ((4, 5, 6),), ()]
>>> mock.call_args_list == expected
True
```

This is an improvement, but still not as nice as *assert_called_with*. Here's a helper function that pops the last argument of the call args list and decrements the call count. This allows you to make asserts as a series of calls to *assert_called_with* followed by a *pop_last_call*.

```
>>> def pop_last_call(mock):
...     if not mock.call_count:
...         raise AssertionError("Cannot pop last call: call_count is 0")
...     mock.call_args_list.pop()
...     try:
...         mock.call_args = mock.call_args_list[-1]
...     except IndexError:
...         mock.call_args = None
...         mock.called = False
```

```
...     mock.call_count -=1
...
>>> mock = Mock(return_value=None)
>>> mock(1, foo='bar')
>>> mock(2, foo='baz')
>>> mock(3, foo='spam')
>>> mock.assert_called_with(3, foo='spam')
>>> pop_last_call(mock)
>>> mock.assert_called_with(2, foo='baz')
>>> pop_last_call(mock)
>>> mock.assert_called_once_with(1, foo='bar')
```

The calls to *assert_called_with* are made in reverse order to the actual calls. Your final call can be a call to *assert_called_once_with*, that ensures there were no extra calls you weren't expecting. You could, if you wanted, extend the function to take args and kwargs and do the assert for you.

2.2.8 Coping with mutable arguments

Another situation is rare, but can bite you, is when your mock is called with mutable arguments. *call_args* and *call_args_list* store *references* to the arguments. If the arguments are mutated by the code under test then you can no longer make assertions about what the values were when the mock was called.

Here's some example code that shows the problem. Imagine the following functions defined in 'mymodule':

```
def frob(val):
    pass

def grob(val):
    "First frob and then clear val"
    frob(val)
    val.clear()
```

When we try to test that *grob* calls *frob* with the correct argument look what happens:

```
>>> with patch('mymodule.frob') as mock_frob:
...     val = set([6])
...     mymodule.grob(val)
...
>>> val
set([])
>>> mock_frob.assert_called_with(set([6]))
Traceback (most recent call last):
    ...
AssertionError: Expected: ((set([6]),), {})
Called with: ((set([]),), {})
```

One possibility would be for mock to copy the arguments you pass in. This could then cause problems if you do assertions that rely on object identity for equality.

Here's one solution that uses the `side_effect` functionality. If you provide a *side_effect* function for a mock then *side_effect* will be called with the same args as the mock. This gives us an opportunity to copy the arguments and store them for later assertions. In this example I'm using *another* mock to store the arguments so that I can use the mock methods for doing the assertion. Again a helper function sets this up for me.

```
>>> from copy import deepcopy
>>> from mock import Mock, patch, DEFAULT
>>> def copy_call_args(mock):
...     new_mock = Mock()
...     def side_effect(*args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         new_mock(*args, **kwargs)
...         return DEFAULT
...     mock.side_effect = side_effect
...     return new_mock
...
>>> with patch('mymodule.frob') as mock_frob:
...     new_mock = copy_call_args(mock_frob)
...     val = set([6])
...     mymodule.grob(val)
...
>>> new_mock.assert_called_with(set([6]))
>>> new_mock.call_args
((set([6]),), {})
```

copy_call_args is called with the mock that will be called. It returns a new mock that we do the assertion on. The *side_effect* function makes a copy of the args and calls our *new_mock* with the copy.

Note: If your mock is only going to be used once there is an easier way of checking arguments at the point they are called. You can simply do the checking inside a *side_effect* function.

```
>>> def side_effect(arg):
...     assert arg == set([6])
...
>>> mock = Mock(side_effect=side_effect)
>>> mock(set([6]))
>>> mock(set())
Traceback (most recent call last):
  ...
AssertionError
```

2.2.9 Multiple calls with different effects

Handling code that needs to behave differently on subsequent calls during the test can be tricky. For example you may have a function that needs to raise an exception the first time it is called but returns a response on the second call (testing retry behaviour).

One approach is to use a `side_effect` function that replaces itself. The first time it is called the *side_effect* sets a new *side_effect* that will be used for the second call. It then raises an exception:

```
>>> def side_effect(*args):
...     def second_call(*args):
...       return 'response'
...     mock.side_effect = second_call
...     raise Exception('boom')
...
>>> mock = Mock(side_effect=side_effect)
>>> mock('first')
Traceback (most recent call last):
  ...
Exception: boom
>>> mock('second')
'response'
>>> mock.assert_called_with('second')
```

Another perfectly valid way would be to pop return values from a list. If the return value is an exception, raise it instead of returning it:

```
>>> returns = [Exception('boom'), 'response']
>>> def side_effect(*args):
...     result = returns.pop(0)
...     if isinstance(result, Exception):
...       raise result
...     return result
...
>>> mock = Mock(side_effect=side_effect)
>>> mock('first')
Traceback (most recent call last):
  ...
Exception: boom
>>> mock('second')
'response'
>>> mock.assert_called_with('second')
```

Which approach you prefer is a matter of taste. The first approach is actually a line shorter but maybe the second approach is more readable.

2.2.10 Nesting Patches

Using patch as a context manager is nice, but if you do multiple patches you can end up with nested with statements indenting further and further to the right:

```
>>> class MyTest(TestCase):
...
...     def test_foo(self):
...         with patch('mymodule.Foo') as mock_foo:
...             with patch('mymodule.Bar') as mock_bar:
...                 with patch('mymodule.Spam') as mock_spam:
...                     assert mymodule.Foo is mock_foo
...                     assert mymodule.Bar is mock_bar
...                     assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').test_foo()
>>> assert mymodule.Foo is original
```

With unittest2 *cleanup* functions and the *patch methods: start and stop* we can achieve the same effect without the nested indentation. A simple helper method, *create_patch*, puts the patch in place and returns the created mock for us:

```
>>> class MyTest(TestCase):
...
...     def create_patch(self, name):
...         patcher = patch(name)
...         thing = patcher.start()
...         self.addCleanup(patcher.stop)
...         return thing
...
...     def test_foo(self):
...         mock_foo = self.create_patch('mymodule.Foo')
...         mock_bar = self.create_patch('mymodule.Bar')
...         mock_spam = self.create_patch('mymodule.Spam')
...
...         assert mymodule.Foo is mock_foo
...         assert mymodule.Bar is mock_bar
...         assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').run()
>>> assert mymodule.Foo is original
```

2.2.11 Mocking a dictionary with MagicMock

You may want to mock a dictionary, or other container object, recording all access to it whilst having it still behave like a dictionary.

We can do this with `MagicMock`, which will behave like a dictionary, and using `Mock.side_effect` to delegate dictionary access to a real underlying dictionary that is under our control.

When the *__getitem__* and *__setitem__* methods of our *MagicMock* are called (normal dictionary access) then *side_effect* is called with the key (and in the case of *__setitem__* the value too). We can also control what is returned.

After the *MagicMock* has been used we can use attributes like `Mock.call_args_list` to assert about how the dictionary was used:

```
>>> my_dict = {'a': 1, 'b': 2, 'c': 3}
>>> def getitem(name):
...      return my_dict[name]
...
>>> def setitem(name, val):
...     my_dict[name] = val
...
>>> mock = MagicMock()
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

Note: An alternative to using *MagicMock* is to use *Mock* and *only* provide the magic methods you specifically want:

```
>>> mock = Mock()
>>> mock.__setitem__ = Mock(side_effect=getitem)
>>> mock.__getitem__ = Mock(side_effect=setitem)
```

A *third* option is to use *MagicMock* but passing in *dict* as the *spec* (or *spec_set*) argument so that the *MagicMock* created only has dictionary magic methods available:

```
>>> mock = MagicMock(spec_set=dict)
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

With these side effect functions in place, the *mock* will behave like a normal dictionary but recording the access. It even raises a *KeyError* if you try to access a key that doesn't exist.

```
>>> mock['a']
1
>>> mock['c']
3
```

```
>>> mock['d']
Traceback (most recent call last):
  ...
KeyError: 'd'
>>> mock['b'] = 'fish'
>>> mock['d'] = 'eggs'
>>> mock['b']
'fish'
>>> mock['d']
'eggs'
```

After it has been used you can make assertions about the access using the normal mock methods and attributes:

```
>>> mock.__getitem__.call_args_list
[(('a',), {}), (('c',), {}), (('d',), {}), (('b',), {}), (('d',), {})]
>>> mock.__setitem__.call_args_list
[(('b', 'fish'), {}), (('d', 'eggs'), {})]
>>> my_dict
{'a': 1, 'c': 3, 'b': 'fish', 'd': 'eggs'}
```

2.2.12 Less verbose configuration of mock objects

If you have a mock object, particularly one created for you by patch, setting up attributes and return values for methods takes one line for every aspect of configuration.

A feature I'm considering for mock 0.8.0 is an api for making configuring mocks less verbose. As is the way of these things, it is easy to prototype this first with a function that you can use right now.

configure_mock is a function that takes a Mock() instance along with keyword arguments for attributes of the mock you want to set. For example, to set *mock.foo* to 3 and *mock.bar* to None, you call:

```
>>> mock = Mock()
>>> configure_mock(mock, foo=3, bar=None)
<mock.Mock object at 0x...>
>>> mock.foo
3
>>> print mock.bar
None
```

return_value and *side_effect* can be used to set them directly on the main mock *anyway* as they are just attributes.

```
>>> mock = Mock()
>>> configure_mock(mock, side_effect=KeyError)
```

```
<mock.Mock object at 0x...>
>>> mock()
Traceback (most recent call last):
  ...
KeyError
```

This is fine for directly setting attributes, but what if you want to configure the return values or side effects of child mocks? How about using standard dotted notation to specify these. Instead of normal keyword arguments you'll need to build a dictionary of arguments and pass them in with **. The function could also create a mock for us if we pass in *None*:

```
>>> args = {'foo.baz.return_value': 'fish', 'foo.side_effect':
... RuntimeError, 'side_effect': KeyError, 'foo.bar': 3}
...
>>> mock = configure_mock(None, **args)
>>> mock.foo.bar
3
>>> mock()
Traceback (most recent call last):
  ...
KeyError
>>> mock.foo.baz()
'fish'
>>> mock.foo()
Traceback (most recent call last):
  ...
RuntimeError
```

If you have any opinions on this then please comment on the issue.

A minimal implementation of *configure_mock* that you can start using now is:

```
def configure_mock(mock, **kwargs):
    if mock is None:
        mock = Mock()
    for arg, val in sorted(kwargs.items(),
                           key=lambda entry: len(entry[0].split('.'))):
        args = arg.split('.')
        final = args.pop()
        obj = mock
        for entry in args:
            obj = getattr(obj, entry)
        setattr(obj, final, val)
    return mock
```

2.2.13 Mocking imports with patch.dict

One situation where mocking can be hard is where you have a local import inside a function. These are harder to mock because they aren't using an object from the module namespace that we can patch out.

Generally local imports are to be avoided. They are sometimes done to prevent circular dependencies, for which there is *usually* a much better way to solve the problem (refactor the code) or to prevent "up front costs" by delaying the import. This can also be solved in better ways than an unconditional local import (store the module as a class or module attribute and only do the import on first use).

That aside there is a way to use *mock* to affect the results of an import. Importing fetches an *object* from the *sys.modules* dictionary. Note that it fetches an *object*, which need not be a module. Importing a module for the first time results in a module object being put in *sys.modules*, so usually when you import something you get a module back. This need not be the case however.

This means you can use `patch.dict()` to *temporarily* put a mock in place in *sys.modules*. Any imports whilst this patch is active will fetch the mock. When the patch is complete (the decorated function exits, the with statement body is complete or *patcher.stop()* is called) then whatever was there previously will be restored safely.

Here's an example that mocks out the 'fooble' module.

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     import fooble
...     fooble.blob()
...
<mock.Mock object at 0x...>
>>> assert 'fooble' not in sys.modules
>>> mock.blob.assert_called_once_with()
```

As you can see the *import fooble* succeeds, but on exit there is no 'fooble' left in *sys.modules*.

This also works for the *from module import name* form:

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     from fooble import blob
...     blob.blip()
...
<mock.Mock object at 0x...>
>>> mock.blob.blip.assert_called_once_with()
```

With slightly more work you can also mock package imports:

```
>>> mock = Mock()
>>> modules = {'package': mock, 'package.module': mock.module}
>>> with patch.dict('sys.modules', modules):
```

```
...     from package.module import fooble
...     fooble()
...
<mock.Mock object at 0x...>
>>> mock.module.fooble.assert_called_once_with()
```

Unfortunately it seems that using *patch.dict* as a test *decorator* on *sys.modules* interferes with the way nosetests collects tests. *nosetests* does some manipulation of *sys.modules* (along with *sys.path* manipulation) and using *patch.dict* with *sys.modules* can cause it to not find tests. Using *patch.dict* as a context manager, or using the *patch methods: start and stop*, work around this by taking a reference to *sys.modules* inside the test rather than at import time. (Using *patch.dict* as a decorator takes a *reference* to *sys.modules* at import time, it doesn't do the patching until the test is executed though.)

2.2.14 Tracking order of calls and less verbose call assertions

The `Mock` class allows you to track the *order* of method calls on your mock objects through the `Mock.method_calls` attribute. This doesn't allow you to track the order of calls between separate mock objects, however we can use *method_calls* to achieve the same effect.

Because mocks track calls to child mocks in *method_calls*, and accessing an arbitrary attribute of a mock creates a child mock, we can create our separate mocks from a parent one. Calls to those child mock will then all be recorded, in order, in the *method_calls* of the parent:

```
>>> manager = Mock()

>>> mock_foo = manager.foo
>>> mock_bar = manager.bar

>>> mock_foo.something()
<mock.Mock object at 0x...>
>>> mock_bar.other.thing()
<mock.Mock object at 0x...>

>>> manager.method_calls
[('foo.something', (), {}), ('bar.other.thing', (), {})]
```

Using the "soft comparisons" feature of mock 0.7.0 we can make the final assertion about the expected calls less verbose:

```
>>> expected_calls = [('foo.something',), ('bar.other.thing',)]
>>> manager.method_calls == expected_calls
True
```

To make them even less verbose I would like to add a new *call* object to mock 0.8.0. You can see the issues I expect to work on for 0.8.0 in the issues list.

call would look something like this:

```
class Call(object):
    def __init__(self, name=None):
        self.name = name

    def __call__(self, *args, **kwargs):
        if self.name is None:
            return (args, kwargs)
        return (self.name, args, kwargs)

    def __getattr__(self, attr):
        if self.name is None:
            return Call(attr)
        name = '%s.%s' % (self.name, attr)
        return Call(name)

call = Call()
```

You can then use it like this:

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, 3)
>>> mock(a=3, b=6)
>>> mock.call_args_list == [call(1, 2, 3), call(a=3, b=6)]
True

>>> mock = Mock()
>>> mock.foo(1, 2 ,3)
<mock.Mock object at 0x...>
>>> mock.bar.baz(a=3, b=6)
<mock.Mock object at 0x...>
>>> mock.method_calls == [call.foo(1, 2, 3), call.bar.baz(a=3, b=6)]
True
```

And for good measure, the first example (tracking order of calls between mocks) using the new *call* object for assertions:

```
>>> manager = Mock()

>>> mock_foo = manager.foo
>>> mock_bar = manager.bar

>>> mock_foo.something()
<mock.Mock object at 0x...>
>>> mock_bar.other.thing()
<mock.Mock object at 0x...>

>>> manager.method_calls == [call.foo.something(), call.bar.other.thing()]
```

```
True
```

2.2.15 Matching any argument in assertions

Sometimes you may need to make assertions about *some* of the arguments in a call to mock, but either not care about some of the arguments or want to pull them individually out of `Mock.call_args` and make more complex assertions on them.

To ignore certain arguments you can pass in objects that compare equal to *everything*. Calls to `Mock.assert_called_with()` and `Mock.assert_called_once_with()` will then succeed no matter what was passed in.

Here's an example implementation:

```
>>> class _ANY(object):
...     def __eq__(self, other):
...         return True
...
>>> ANY = _ANY()
```

And an example of using it:

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar=object())
>>> mock.assert_called_once_with('foo', bar=ANY)
```

2.2.16 More complex argument matching

Using the same basic concept as the *ANY* pattern above we can implement matchers to do more complex assertions on objects used as arguments to mocks.

Suppose we expect some object to be passed to a mock that by default compares equal based on object identity (which is the Python default for user defined classes). To use `Mock.assert_called_with()` we would need to pass in the exact same object. If we are only interested in some of the attributes of this object then we can create a matcher that will check these attributes for us.

You can see in this example how a 'standard' call to *assert_called_with* isn't sufficient:

```
>>> class Foo(object):
...     def __init__(self, a, b):
...         self.a, self.b = a, b
...
>>> mock = Mock(return_value=None)
>>> mock(Foo(1, 2))
>>> mock.assert_called_with(Foo(1, 2))
```

```
Traceback (most recent call last):
  ...
AssertionError: Expected: ((<__main__.Foo object at 0x...>,), {})
Called with: ((<__main__.Foo object at 0x...>,), {})
```

A comparison function for our *Foo* class might look something like this:

```
>>> def compare(self, other):
...     if not type(self) == type(other):
...         return False
...     if self.a != other.a:
...         return False
...     if self.b != other.b:
...         return False
...     return True
...
```

And a matcher object that can use comparison functions like this for its equality operation would look something like this:

```
>>> class Matcher(object):
...     def __init__(self, compare, some_obj):
...         self.compare = compare
...         self.some_obj = some_obj
...     def __eq__(self, other):
...         return self.compare(self.some_obj, other)
...
```

Putting all this together:

```
>>> match_foo = Matcher(compare, Foo(1, 2))
>>> mock.assert_called_with(match_foo)
```

The *Matcher* is instantiated with our compare function and the *Foo* object we want to compare against. In *assert_called_with* the *Matcher* equality method will be called, which compares the object the mock was called with against the one we created our matcher with. If they match then *assert_called_with* passes, and if they don't an *AssertionError* is raised:

```
>>> match_wrong = Matcher(compare, Foo(3, 4))
>>> mock.assert_called_with(match_wrong)
Traceback (most recent call last):
  ...
AssertionError: Expected: ((<Matcher object at 0x...>,), {})
Called with: ((<Foo object at 0x...>,), {})
```

With a bit of tweaking you could have the comparison function raise the *AssertionError* directly and provide a more useful failure message.

As of version 1.5, the Python testing library PyHamcrest provides similar functionality, that may

be useful here, in the form of its equality matcher (hamcrest.library.integration.match_equality).

2.3 Mock Library Comparison

A side-by-side comparison of how to accomplish some basic tasks with mock and some other popular Python mocking libraries and frameworks.

These are:

- flexmock
- mox
- Mocker
- dingus
- fudge

Popular python mocking frameworks not yet represented here include MiniMock.

pMock (last release 2004 and doesn't import in recent versions of Python) and python-mock (last release 2005) are intentionally omitted.

Note: A more up to date, and tested for all mock libraries (only the mock examples on this page can be executed as doctests) version of this comparison is maintained by Gary Bernhardt:

- Python Mock Library Comparison

This comparison is by no means complete, and also may not be fully idiomatic for all the libraries represented. *Please* contribute corrections, missing comparisons, or comparisons for additional libraries to the mock issue tracker.

This comparison page was originally created by the Mox project and then extended for flexmock and mock by Herman Sheremetyev. Dingus examples written by Gary Bernhadt. fudge examples provided by Kumar McMillan.

Note: The examples tasks here were originally created by Mox which is a mocking *framework* rather than a library like mock. The tasks shown naturally exemplify tasks that frameworks are good at and not the ones they make harder. In particular you can take a *Mock* or *MagicMock* object and use it in any way you want with no up-front configuration. The same is also true for Dingus.

The examples for mock here assume version 0.7.0.

2.3.1 Simple fake object

```
>>> # mock
>>> my_mock = mock.Mock()
>>> my_mock.some_method.return_value = "calculated value"
>>> my_mock.some_attribute = "value"
>>> assertEqual("calculated value", my_mock.some_method())
>>> assertEqual("value", my_mock.some_attribute)

# Flexmock
mock = flexmock(some_method=lambda: "calculated value", some_attribute="value")
assertEqual("calculated value", mock.some_method())
assertEqual("value", mock.some_attribute)

# Mox
mock = mox.MockAnything()
mock.some_method().AndReturn("calculated value")
mock.some_attribute = "value"
mox.Replay(mock)
assertEqual("calculated value", mock.some_method())
assertEqual("value", mock.some_attribute)

# Mocker
mock = mocker.mock()
mock.some_method()
mocker.result("calculated value")
mocker.replay()
mock.some_attribute = "value"
assertEqual("calculated value", mock.some_method())
assertEqual("value", mock.some_attribute)

>>> # Dingus
>>> my_dingus = dingus.Dingus(some_attribute="value",
...                           some_method__returns="calculated value")
>>> assertEqual("calculated value", my_dingus.some_method())
>>> assertEqual("value", my_dingus.some_attribute)

>>> # fudge
>>> my_fake = (fudge.Fake()
...            .provides('some_method')
...            .returns("calculated value")
...            .has_attr(some_attribute="value"))
...
>>> assertEqual("calculated value", my_fake.some_method())
>>> assertEqual("value", my_fake.some_attribute)
```

2.3.2 Simple mock

```
>>> # mock
>>> my_mock = mock.Mock()
>>> my_mock.some_method.return_value = "value"
>>> assertEqual("value", my_mock.some_method())
>>> my_mock.some_method.assert_called_once_with()

# Flexmock
mock = flexmock()
mock.should_receive("some_method").and_return("value").once
assertEqual("value", mock.some_method())

# Mox
mock = mox.MockAnything()
mock.some_method().AndReturn("value")
mox.Replay(mock)
assertEqual("value", mock.some_method())
mox.Verify(mock)

# Mocker
mock = mocker.mock()
mock.some_method()
mocker.result("value")
mocker.replay()
assertEqual("value", mock.some_method())
mocker.verify()

>>> # Dingus
>>> my_dingus = dingus.Dingus(some_method__returns="value")
>>> assertEqual("value", my_dingus.some_method())
>>> assert my_dingus.some_method.calls().once()

>>> # fudge
>>> @fudge.test
... def test():
...     my_fake = (fudge.Fake()
...                .expects('some_method')
...                .returns("value")
...                .times_called(1))
...
>>> test()
Traceback (most recent call last):
...
AssertionError: fake:my_fake.some_method() was not called
```

2.3.3 Creating partial mocks

```
>>> # mock
>>> SomeObject.some_method = mock.Mock(return_value='value')
>>> assertEqual("value", SomeObject.some_method())

# Flexmock
flexmock(SomeObject).should_receive("some_method").and_return('value')
assertEqual("value", mock.some_method())

# Mox
mock = mox.MockObject(SomeObject)
mock.some_method().AndReturn("value")
mox.Replay(mock)
assertEqual("value", mock.some_method())
mox.Verify(mock)

# Mocker
mock = mocker.mock(SomeObject)
mock.Get()
mocker.result("value")
mocker.replay()
assertEqual("value", mock.some_method())
mocker.verify()

>>> # Dingus
>>> object = SomeObject
>>> object.some_method = dingus.Dingus(return_value="value")
>>> assertEqual("value", object.some_method())

>>> # fudge
>>> fake = fudge.Fake().is_callable().returns("<fudge-value>")
>>> with fudge.patched_context(SomeObject, 'some_method', fake):
...     s = SomeObject()
...     assertEqual("<fudge-value>", s.some_method())
...
```

2.3.4 Ensure calls are made in specific order

```
>>> # mock
>>> my_mock = mock.Mock(spec=SomeObject)
>>> my_mock.method1()
<mock.Mock object at 0x...>
>>> my_mock.method2()
<mock.Mock object at 0x...>
>>> assertEqual(my_mock.method_calls, [('method1',), ('method2',)])
```

```
# Flexmock
mock = flexmock(SomeObject)
mock.should_receive('method1').once.ordered.and_return('first thing')
mock.should_receive('method2').once.ordered.and_return('second thing')

# Mox
mock = mox.MockObject(SomeObject)
mock.method1().AndReturn('first thing')
mock.method2().AndReturn('second thing')
mox.Replay(mock)
mox.Verify(mock)

# Mocker
mock = mocker.mock()
with mocker.order():
    mock.method1()
    mocker.result('first thing')
    mock.method2()
    mocker.result('second thing')
    mocker.replay()
    mocker.verify()

>>> # Dingus
>>> my_dingus = dingus.Dingus()
>>> my_dingus.method1()
<Dingus ...>
>>> my_dingus.method2()
<Dingus ...>
>>> assertEqual(['method1', 'method2'], [call.name for call in my_dingus.calls])

>>> # fudge
>>> @fudge.test
... def test():
...     my_fake = (fudge.Fake()
...                .remember_order()
...                .expects('method1')
...                .expects('method2'))
...     my_fake.method2()
...     my_fake.method1()
...
>>> test()
Traceback (most recent call last):
...
AssertionError: Call #1 was fake:my_fake.method2(); Expected: #1 fake:my_fake.metho
```

2.3.5 Raising exceptions

```
>>> # mock
>>> my_mock = mock.Mock()
>>> my_mock.some_method.side_effect = SomeException("message")
>>> assertRaises(SomeException, my_mock.some_method)

# Flexmock
mock = flexmock()
mock.should_receive("some_method").and_raise(SomeException("message"))
assertRaises(SomeException, mock.some_method)

# Mox
mock = mox.MockAnything()
mock.some_method().AndRaise(SomeException("message"))
mox.Replay(mock)
assertRaises(SomeException, mock.some_method)
mox.Verify(mock)

# Mocker
mock = mocker.mock()
mock.some_method()
mocker.throw(SomeException("message"))
mocker.replay()
assertRaises(SomeException, mock.some_method)
mocker.verify()

>>> # Dingus
>>> my_dingus = dingus.Dingus()
>>> my_dingus.some_method = dingus.exception_raiser(SomeException)
>>> assertRaises(SomeException, my_dingus.some_method)

>>> # fudge
>>> my_fake = (fudge.Fake()
...             .is_callable()
...             .raises(SomeException("message")))
...
>>> my_fake()
Traceback (most recent call last):
...
SomeException: message
```

2.3.6 Override new instances of a class

```
>>> # mock
>>> with mock.patch('somemodule.Someclass') as MockClass:
...     MockClass.return_value = some_other_object
...     assertEqual(some_other_object, somemodule.Someclass())
...

# Flexmock
flexmock(some_module.SomeClass, new_instances=some_other_object)
assertEqual(some_other_object, some_module.SomeClass())

# Mox
# (you will probably have mox.Mox() available as self.mox in a real test)
mox.Mox().StubOutWithMock(some_module, 'SomeClass', use_mock_anything=True)
some_module.SomeClass().AndReturn(some_other_object)
mox.ReplayAll()
assertEqual(some_other_object, some_module.SomeClass())

# Mocker
# (TODO)

>>> # Dingus
>>> MockClass = dingus.Dingus(return_value=some_other_object)
>>> with dingus.patch('somemodule.SomeClass', MockClass):
...     assertEqual(some_other_object, somemodule.SomeClass())
...

>>> # fudge
>>> @fudge.patch('somemodule.SomeClass')
... def test(FakeClass):
...     FakeClass.is_callable().returns(some_other_object)
...     assertEqual(some_other_object, somemodule.SomeClass())
...
>>> test()
```

2.3.7 Call the same method multiple times

Note: You don't need to do *any* configuration to call *mock.Mock()* methods multiple times. Attributes like *call_count*, *call_args_list* and *method_calls* provide various different ways of making assertions about how the mock was used.

```
>>> # mock
>>> my_mock = mock.Mock()
```

```
>>> my_mock.some_method()
<mock.Mock object at 0x...>
>>> my_mock.some_method()
<mock.Mock object at 0x...>
>>> assert my_mock.some_method.call_count >= 2

# Flexmock # (verifies that the method gets called at least twice)
flexmock(some_object).should_receive('some_method').at_least.twice

# Mox
# (does not support variable number of calls, so you need to create a new entry for
mock = mox.MockObject(some_object)
mock.some_method(mox.IgnoreArg(), mox.IgnoreArg())
mock.some_method(mox.IgnoreArg(), mox.IgnoreArg())
mox.Replay(mock)
mox.Verify(mock)

# Mocker
# (TODO)

>>> # Dingus
>>> my_dingus = dingus.Dingus()
>>> my_dingus.some_method()
<Dingus ...>
>>> my_dingus.some_method()
<Dingus ...>
>>> assert len(my_dingus.calls('some_method')) == 2

>>> # fudge
>>> @fudge.test
... def test():
...     my_fake = fudge.Fake().expects('some_method').times_called(2)
...     my_fake.some_method()
...
>>> test()
Traceback (most recent call last):
...
AssertionError: fake:my_fake.some_method() was called 1 time(s). Expected 2.
```

2.3.8 Mock chained methods

```
>>> # mock
>>> my_mock = mock.Mock()
>>> method3 = my_mock.method1.return_value.method2.return_value.method3
>>> method3.return_value = 'some value'
```

```
>>> assertEqual('some value', my_mock.method1().method2().method3(1, 2))
>>> method3.assert_called_once_with(1, 2)

# Flexmock
# (intermediate method calls are automatically assigned to temporary fake objects
# and can be called with any arguments)
flexmock(some_object).should_receive(
    'method1.method2.method3'
).with_args(arg1, arg2).and_return('some value')
assertEqual('some_value', some_object.method1().method2().method3(arg1, arg2))

# Mox
mock = mox.MockObject(some_object)
mock2 = mox.MockAnything()
mock3 = mox.MockAnything()
mock.method1().AndReturn(mock1)
mock2.method2().AndReturn(mock2)
mock3.method3(arg1, arg2).AndReturn('some_value')
self.mox.ReplayAll()
assertEqual("some_value", some_object.method1().method2().method3(arg1, arg2))
self.mox.VerifyAll()

# Mocker
# (TODO)

>>> # Dingus
>>> my_dingus = dingus.Dingus()
>>> method3 = my_dingus.method1.return_value.method2.return_value.method3
>>> method3.return_value = 'some value'
>>> assertEqual('some value', my_dingus.method1().method2().method3(1, 2))
>>> assert method3.calls('()', 1, 2).once()

>>> # fudge
>>> @fudge.test
... def test():
...     my_fake = fudge.Fake()
...     (my_fake
...      .expects('method1')
...      .returns_fake()
...      .expects('method2')
...      .returns_fake()
...      .expects('method3')
...      .with_args(1, 2)
...      .returns('some value'))
...     assertEqual('some value', my_fake.method1().method2().method3(1, 2))
...
>>> test()
```

2.3.9 Mocking a context manager

Examples for mock, Dingus and fudge only (so far):

```
>>> # mock
>>> my_mock = mock.MagicMock()
>>> with my_mock:
...     pass
...
>>> my_mock.__enter__.assert_called_with()
>>> my_mock.__exit__.assert_called_with(None, None, None)

>>> # Dingus (nothing special here; all dinguses are "magic mocks")
>>> my_dingus = dingus.Dingus()
>>> with my_dingus:
...     pass
...
>>> assert my_dingus.__enter__.calls()
>>> assert my_dingus.__exit__.calls('()', None, None, None)

>>> # fudge
>>> my_fake = fudge.Fake().provides('__enter__').provides('__exit__')
>>> with my_fake:
...     pass
...
```

2.3.10 Mocking the builtin open used as a context manager

Example for mock only (so far):

```
>>> # mock
>>> my_mock = mock.MagicMock()
>>> with mock.patch('__builtin__.open', my_mock):
...     manager = my_mock.return_value.__enter__.return_value
...     manager.read.return_value = 'some data'
...     with open('foo') as h:
...         data = h.read()
...
>>> data
'some data'
>>> my_mock.assert_called_once_with('foo')
```

or:

```
>>> # mock
>>> with mock.patch('__builtin__.open') as my_mock:
...     my_mock.return_value.__enter__ = lambda s: s
```

```
...     my_mock.return_value.__exit__ = mock.Mock()
...     my_mock.return_value.read.return_value = 'some data'
...     with open('foo') as h:
...         data = h.read()
...
>>> data
'some data'
>>> my_mock.assert_called_once_with('foo')

>>> # Dingus
>>> my_dingus = dingus.Dingus()
>>> with dingus.patch('__builtin__.open', my_dingus):
...     file_ = open.return_value.__enter__.return_value
...     file_.read.return_value = 'some data'
...     with open('foo') as h:
...         data = f.read()
...
>>> data
'some data'
>>> assert my_dingus.calls('()', 'foo').once()

>>> # fudge
>>> from contextlib import contextmanager
>>> from StringIO import StringIO
>>> @contextmanager
... def fake_file(filename):
...     yield StringIO('sekrets')
...
>>> with fudge.patch('__builtin__.open') as fake_open:
...     fake_open.is_callable().calls(fake_file)
...     with open('/etc/password') as f:
...         data = f.read()
...
fake:__builtin__.open
>>> data
'sekrets'
```

2.4 TODO and Limitations

Contributions, bug reports and comments welcomed!

Feature requests and bug reports are handled on the issue tracker:

- mock module issue tracker

When mocking a class with `patch`, passing in `spec=True`, the mock class has an instance

created from the same spec. Should this be the default behaviour for mocks anyway (mock return values inheriting the spec from their parent), or should it be controlled by an additional keyword argument (`inherit`) to the Mock constructor?

Interaction of magic methods with spec, wraps. For example, should spec cause all methods to be wrapped with mocksignature perhaps? (or another keyword argument perhaps?)

Should magic method calls (including *__call__*?) be tracked in *method_calls*?

Could take a patch keyword argument and auto-do the patching in the constructor and unpatch in the destructor. This would be useful in itself, but violates TOOWTDI and would be unsafe for IronPython (non-deterministic calling of destructors). Destructors aren't called *anyway* where there are cycles, but a weak reference with a callback can be used to get round this.

`Mock` has several attributes. This makes it unsuitable for mocking objects that use these attribute names. A way round this would be to provide `start` and `stop` (or similar) methods that *hide* these attributes when needed. (*_name* is a mock attribute that will be not-uncommon, but it is accessed externally on mocks when building *method_calls* names and so can't be changed to a double underscore name.)

If a patch is started using *patch.start* and then not stopped correctly, due to a bug in the test code, then the unpatching is not done. Using weak references it would be possible to detect and fix this when the patch object itself is garbage collected. This is tricky to get right though.

2.5 CHANGELOG

2.5.1 2011/05/30 Version 0.7.2

- BUGFIX: instances of list subclasses can now be used as mock specs
- BUGFIX: MagicMock equality / inequality protocol methods changed to use the default equality / inequality. This is done through a *side_effect* on the mocks used for *__eq__* / *__ne__*

2.5.2 2011/05/06 Version 0.7.1

Package fixes contributed by Michael Fladischer. No code changes.

- Include template in package
- Use isolated binaries for the tox tests
- Unset executable bit on docs
- Fix DOS line endings in getting-started.txt

2.5.3 2011/03/05 Version 0.7.0

No API changes since 0.7.0 rc1. Many documentation changes including a stylish new Sphinx theme.

The full set of changes since 0.6.0 are:

- Python 3 compatibility
- Ability to mock magic methods with *Mock* and addition of *MagicMock* with pre-created magic methods
- Addition of *mocksignature* and *mocksignature* argument to *patch* and *patch.object*
- Addition of *patch.dict* for changing dictionaries during a test
- Ability to use *patch*, *patch.object* and *patch.dict* as class decorators
- Renamed `patch_object` to *patch.object* (`patch_object` is deprecated)
- Addition of soft comparisons: *call_args*, *call_args_list* and *method_calls* now return tuple-like objects which compare equal even when empty args or kwargs are skipped
- patchers (*patch*, *patch.object* and *patch.dict*) have start and stop methods
- Addition of *assert_called_once_with* method
- Mocks can now be named (*name* argument to constructor) and the name is used in the repr
- repr of a mock with a spec includes the class name of the spec
- *assert_called_with* works with *python -OO*
- New *spec_set* keyword argument to *Mock* and *patch*. If used, attempting to *set* an attribute on a mock not on the spec will raise an *AttributeError*
- Mocks created with a spec can now pass *isinstance* tests (*__class__* returns the type of the spec)
- Added docstrings to all objects
- Improved failure message for *Mock.assert_called_with* when the mock has not been called at all
- Decorated functions / methods have their docstring and *__module__* preserved on Python 2.4.
- BUGFIX: *mock.patch* now works correctly with certain types of objects that proxy attribute access, like the django settings object
- BUGFIX: mocks are now copyable (thanks to Ned Batchelder for reporting and diagnosing this)
- BUGFIX: *spec=True* works with old style classes

- BUGFIX: `help(mock)` works now (on the module). Can no longer use `__bases__` as a valid sentinel name (thanks to Stephen Emslie for reporting and diagnosing this)
- BUGFIX: `side_effect` now works with `BaseException` exceptions like `KeyboardInterrupt`
- BUGFIX: *reset_mock* caused infinite recursion when a mock is set as its own return value
- BUGFIX: patching the same object twice now restores the patches correctly
- with statement tests now skipped on Python 2.4
- Tests require unittest2 (or unittest2-py3k) to run
- Tested with tox on Python 2.4 - 3.2, jython and pypy (excluding 3.0)
- Added 'build_sphinx' command to setup.py (requires setuptools or distribute) Thanks to Florian Bauer
- Switched from subversion to mercurial for source code control
- Konrad Delong added as co-maintainer

2.5.4 2011/02/16 Version 0.7.0 RC 1

Changes since beta 4:

- Tested with jython, pypy and Python 3.2 and 3.1
- Decorated functions / methods have their docstring and *__module__* preserved on Python 2.4
- BUGFIX: *mock.patch* now works correctly with certain types of objects that proxy attribute access, like the django settings object
- BUGFIX: *reset_mock* caused infinite recursion when a mock is set as its own return value

2.5.5 2010/11/12 Version 0.7.0 beta 4

- patchers (*patch*, *patch.object* and *patch.dict*) have start and stop methods
- Addition of *assert_called_once_with* method
- repr of a mock with a spec includes the class name of the spec
- *assert_called_with* works with *python -OO*
- New *spec_set* keyword argument to *Mock* and *patch*. If used, attempting to *set* an attribute on a mock not on the spec will raise an *AttributeError*
- Attributes and return value of a *MagicMock* are *MagicMock* objects

- Attempting to set an unsupported magic method now raises an *AttributeError*
- *patch.dict* works as a class decorator
- Switched from subversion to mercurial for source code control
- BUGFIX: mocks are now copyable (thanks to Ned Batchelder for reporting and diagnosing this)
- BUGFIX: *spec=True* works with old style classes
- BUGFIX: *mocksignature=True* can now patch instance methods via *patch.object*

2.5.6 2010/09/18 Version 0.7.0 beta 3

- Using spec with `MagicMock` only pre-creates magic methods in the spec
- Setting a magic method on a mock with a `spec` can only be done if the spec has that method
- Mocks can now be named (*name* argument to constructor) and the name is used in the repr
- *mocksignature* can now be used with classes (signature based on *__init__*) and callable objects (signature based on *__call__*)
- Mocks created with a spec can now pass *isinstance* tests (*__class__* returns the type of the spec)
- Default numeric value for MagicMock is 1 rather than zero (because the MagicMock bool defaults to True and 0 is False)
- Improved failure message for `Mock.assert_called_with()` when the mock has not been called at all
- Adding the following to the set of supported magic methods:
  - `__getformat__` and `__setformat__`
  - pickle methods
  - `__trunc__`, `__ceil__` and `__floor__`
  - `__sizeof__`
- Added 'build_sphinx' command to setup.py (requires setuptools or distribute) Thanks to Florian Bauer
- with statement tests now skipped on Python 2.4
- Tests require unittest2 to run on Python 2.7
- Improved several docstrings and documentation

2.5.7 2010/06/23 Version 0.7.0 beta 2

- `patch.dict()` works as a context manager as well as a decorator
- `patch.dict` takes a string to specify dictionary as well as a dictionary object. If a string is supplied the name specified is imported
- BUGFIX: `patch.dict` restores dictionary even when an exception is raised

2.5.8 2010/06/22 Version 0.7.0 beta 1

- Addition of `mocksignature()`
- Ability to mock magic methods
- Ability to use `patch` and `patch.object` as class decorators
- Renamed `patch_object` to `patch.object()` (`patch_object` is deprecated)
- Addition of `MagicMock` class with all magic methods pre-created for you
- Python 3 compatibility (tested with 3.2 but should work with 3.0 & 3.1 as well)
- Addition of `patch.dict()` for changing dictionaries during a test
- Addition of `mocksignature` argument to `patch` and `patch.object`
- `help(mock)` works now (on the module). Can no longer use `__bases__` as a valid sentinel name (thanks to Stephen Emslie for reporting and diagnosing this)
- Addition of soft comparisons: *call_args*, *call_args_list* and *method_calls* now return tuple-like objects which compare equal even when empty args or kwargs are skipped
- Added docstrings.
- BUGFIX: `side_effect` now works with `BaseException` exceptions like `KeyboardInterrupt`
- BUGFIX: patching the same object twice now restores the patches correctly
- The tests now require unittest2 to run
- Konrad Delong added as co-maintainer

2.5.9 2009/08/22 Version 0.6.0

- New test layout compatible with test discovery
- Descriptors (static methods / class methods etc) can now be patched and restored correctly
- Mocks can raise exceptions when called by setting `side_effect` to an exception class or instance

- Mocks that wrap objects will not pass on calls to the underlying object if an explicit return_value is set

2.5.10 2009/04/17 Version 0.5.0

- Made DEFAULT part of the public api.
- Documentation built with Sphinx.
- `side_effect` is now called with the same arguments as the mock is called with and if returns a non-DEFAULT value that is automatically set as the `mock.return_value`.
- `wraps` keyword argument used for wrapping objects (and passing calls through to the wrapped object).
- `Mock.reset` renamed to `Mock.reset_mock`, as reset is a common API name.
- `patch` / `patch_object` are now context managers and can be used with `with`.
- A new 'create' keyword argument to patch and patch_object that allows them to patch (and unpatch) attributes that don't exist. (Potentially unsafe to use - it can allow you to have tests that pass when they are testing an API that doesn't exist - use at your own risk!)
- The methods keyword argument to Mock has been removed and merged with spec. The spec argument can now be a list of methods or an object to take the spec from.
- Nested patches may now be applied in a different order (created mocks passed in the opposite order). This is actually a bugfix.
- patch and patch_object now take a spec keyword argument. If spec is passed in as 'True' then the Mock created will take the object it is replacing as its spec object. If the object being replaced is a class, then the return value for the mock will also use the class as a spec.
- A Mock created without a spec will not attempt to mock any magic methods / attributes (they will raise an `AttributeError` instead).

2.5.11 2008/10/12 Version 0.4.0

- Default return value is now a new mock rather than None
- return_value added as a keyword argument to the constructor
- New method 'assert_called_with'
- Added 'side_effect' attribute / keyword argument called when mock is called
- patch decorator split into two decorators:
  - `patch_object` which takes an object and an attribute name to patch (plus optionally a value to patch with which defaults to a mock object)

  - `patch` which takes a string specifying a target to patch; in the form 'package.module.Class.attribute'. (plus optionally a value to patch with which defaults to a mock object)

- Can now patch objects with `None`
- Change to patch for nose compatibility with error reporting in wrapped functions
- Reset no longer clears children / return value etc - it just resets call count and call args. It also calls reset on all children (and the return value if it is a mock).

Thanks to Konrad Delong, Kevin Dangoor and others for patches and suggestions.

2.5.12 2007/12/03 Version 0.3.1

`patch` maintains the name of decorated functions for compatibility with nose test autodiscovery.

Tests decorated with `patch` that use the two argument form (implicit mock creation) will receive the mock(s) passed in as extra arguments.

Thanks to Kevin Dangoor for these changes.

2.5.13 2007/11/30 Version 0.3.0

Removed `patch_module`. `patch` can now take a string as the first argument for patching modules.

The third argument to `patch` is optional - a mock will be created by default if it is not passed in.

2.5.14 2007/11/21 Version 0.2.1

Bug fix, allows reuse of functions decorated with `patch` and `patch_module`.

2.5.15 2007/11/20 Version 0.2.0

Added `spec` keyword argument for creating `Mock` objects from a specification object.

Added `patch` and `patch_module` monkey patching decorators.

Added `sentinel` for convenient access to unique objects.

Distribution includes unit tests.

2.5.16 2007/11/19 Version 0.1.0

Initial release.

CHAPTER

THREE

INSTALLING

The current version is 0.7.2. Mock is still experimental; the API may change (although we are moving towards a 1.0 release when the API will stabilise). If you find bugs or have suggestions for improvements / extensions then please contact us.

- mock on PyPI
- mock documentation as PDF
- Google Code Home & Mercurial Repository

You can checkout the latest development version from the Google Code Mercurial repository with the following command:

```
hg clone https://mock.googlecode.com/hg/ mock
```

If you have pip, setuptools or distribute you can install mock with:

```
easy_install -U mock
pip install -U mock
```

Alternatively you can download the mock distribution from PyPI and after unpacking run:

```
python setup.py install
```

CHAPTER

FOUR

QUICK GUIDE

`Mock` objects create all attributes and methods as you access them and store details of how they have been used. You can configure them, to specify return values or limit what attributes are available, and then make assertions about how they have been used:

```
>>> from mock import Mock
>>> real = ProductionClass()
>>> real.method = Mock(return_value=3)
>>> real.method(3, 4, 5, key='value')
3
>>> real.method.assert_called_with(3, 4, 5, key='value')
```

`side_effect` allows you to perform side effects, return different values or raise an exception when a mock is called:

```
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
  ...
KeyError: 'foo'
>>> values = [1, 2, 3]
>>> def side_effect():
...     return values.pop()
...
>>> mock.side_effect = side_effect
>>> mock(), mock(), mock()
(3, 2, 1)
```

Mock has many other ways you can configure it and control its behaviour. For example the `spec` argument configures the mock to take its specification from another object. Attempting to access attributes or methods on the mock that don't exist on the spec will fail with an `AttributeError`.

The `patch()` decorator / context manager makes it easy to mock classes or objects in a module under test. The object you specify will be replaced with a mock (or other object) during the test and restored when the test ends:

```
>>> from mock import patch
>>> @patch('test_module.ClassName1')
... @patch('test_module.ClassName2')
... def test(MockClass2, MockClass1):
...     test_module.ClassName1()
...     test_module.ClassName2()

...     assert MockClass1 is test_module.ClassName1
...     assert MockClass2 is test_module.ClassName2
...     assert MockClass1.called
...     assert MockClass2.called
...
>>> test()
```

Note: When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal *python* order that decorators are applied). This means from the bottom up, so in the example above the mock for *test_module.ClassName2* is passed in first.

With *patch* it matters that you patch objects in the namespace where they are looked up. This is normally straightforward, but for a quick guide read *where to patch*.

As well as a decorator *patch* can be used as a context manager in a with statement:

```
>>> with patch.object(ProductionClass, 'method') as mock_method:
...     mock_method.return_value = None
...     real = ProductionClass()
...     real.method(1, 2, 3)
...
>>> mock_method.assert_called_with(1, 2, 3)
```

There is also `patch.dict()` for setting values in a dictionary just during a scope and restoring the dictionary to its original state when the test ends:

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

Mock supports the mocking of Python *magic methods*. The easiest way of using magic methods is with the `MagicMock` class. It allows you to do things like:

```
>>> from mock import MagicMock
>>> mock = MagicMock()
>>> mock.__str__.return_value = 'foobarbaz'
>>> str(mock)
```

```
'foobarbaz'
>>> mock.__str__.assert_called_with()
```

Mock allows you to assign functions (or other Mock instances) to magic methods and they will be called appropriately. The MagicMock class is just a Mock variant that has all of the magic methods pre-created for you (well - all the useful ones anyway).

The following is an example of using magic methods with the ordinary Mock class:

```
>>> mock = Mock()
>>> mock.__str__ = Mock()
>>> mock.__str__.return_value = 'wheeeeee'
>>> str(mock)
'wheeeeee'
```

`mocksignature()` is a useful companion to Mock and patch. It creates copies of functions that delegate to a mock, but have the same signature as the original function. This ensures that your mocks will fail in the same way as your production code if they are called incorrectly:

```
>>> from mock import mocksignature
>>> def function(a, b, c):
...     pass
...
>>> function2 = mocksignature(function)
>>> function2.mock.return_value = 'fishy'
>>> function2(1, 2, 3)
'fishy'
>>> function2.mock.assert_called_with(1, 2, 3)
>>> function2('wrong arguments')
Traceback (most recent call last):
  ...
TypeError: <lambda>() takes exactly 3 arguments (1 given)
```

mocksignature can also be used on classes, where it copies the signature of the *__init__* method, and on callable objects where it copies the signature of the *__call__* method.

CHAPTER

FIVE

REFERENCES

Articles and blog entries on testing with Mock:

- PyCon 2011 Video: Testing with mock
- Mocking Django
- Mock recipes
- Mockity mock mock - some love for the mock module
- Python Unit Testing with Mock
- Getting started with Python Mock
- Python mock testing techniques and tools
- How To Test Django Template Tags
- A presentation on Unit Testing with Mock
- Mocking with Django and Google AppEngine

CHAPTER

SIX

TESTS

Mock uses unittest2 for its own test suite. In order to run it, use the `unit2` script that comes with `unittest2` module on a checkout of the source repository:

```
unit2 discover
```

If you have setuptools as well as unittest2 you can run:

```
python setup.py test
```

On Python 3.2 you can use `unittest` module from the standard library.

```
python3.2 -m unittest discover
```

On Python 3 the tests for unicode are skipped as they are not relevant. On Python 2.4 tests that use the with statements are skipped as the with statement is invalid syntax on Python 2.4.

CHAPTER

SEVEN

OLDER VERSIONS

Documentation for older versions of mock:

- mock 0.6.0

CHAPTER

EIGHT

TERMINOLOGY

Terminology for objects used to replace other ones can be confusing. Terms like double, fake, mock, stub, and spy are all used with varying meanings.

In classic mock terminology `mock.Mock` is a spy that allows for *post-mortem* examination. This is what I call the "action -> assertion" [1] pattern of testing.

I'm not however a fan of this "statically typed mocking terminology" promulgated by Martin Fowler. It confuses usage patterns with implementation and prevents you from using natural terminology when discussing mocking.

I much prefer duck typing, if an object used in your test suite looks like a mock object and quacks like a mock object then its fine to call it a mock no matter what the implementation looks like.

This terminology is perhaps more useful in less capable languages where different usage patterns will *require* different implementations. *mock.Mock()* is capable of being used in most of the different roles described by Fowler, except (annoyingly / frustratingly / ironically) a Mock itself!

How about a simpler definition: a "mock object" is an object used to replace a real one in a system under test.

[1] This pattern is called "AAA" by some members of the testing community; "Arrange - Act - Assert".

INDEX

Symbols

__call__, 5

A

articles, 77
assert_called_once_with() (Mock method), 4
assert_called_with() (Mock method), 4

C

call_args (Mock attribute), 7
call_args_list (Mock attribute), 8
call_count (Mock attribute), 5
called (Mock attribute), 5
calling, 5

D

DEFAULT (in module mock), 15

E

easy_install, 73

G

Getting Started, 25

H

hg, 73

I

installing, 71
introduction, 1

M

MagicMock (class in mock), 18
method_calls (Mock attribute), 8
Mock (class in mock), 3
mock (module), 1
mocksignature() (in module mock), 20

N

name, 3

O

older versions, 81

P

patch() (in module mock), 9
patch.dict() (in module mock), 11
patch.object() (in module mock), 11
pip, 73
Python 3, 81

R

references, 77
repository, 73
reset_mock() (Mock method), 5
return_value, 3
return_value (Mock attribute), 6

S

sentinel (in module mock), 15
setuptools, 73
side_effect, 3
side_effect (Mock attribute), 6
spec, 3

T

tests, 79

U

unittest2, 79

W

wraps, 3