Mock Documentation

Release 0.8.0

Michael Foord

February 13, 2012

CONTENTS

Author Michael Foord

Version 0.8.0

Date 2012/02/13

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 library for testing in Python. It allows you to replace parts of your system under test with mock objects and make assertions about how they have been used.

mock provides a core `Mock` class removing 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.

Additionally, mock 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, Python 3 plus the latest versions of Jython and PyPy.

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.

`MagicMock` is a subclass of *Mock* with all the magic methods pre-created and ready to use. There are also non-callable variants, useful when you are mocking out objects that aren't callable: `NonCallableMock` and `NonCallableMagicMock`

The `patch()` decorators makes it easy to temporarily replace classes in a particular module with a *Mock* object. By default *patch* will create a *MagicMock* for you. You can specify an alternative class of *Mock* using the *new_callable* argument to *patch*.

class **Mock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, ***kwargs*)

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 `__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*.

[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*.

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

  If *side_effect* is an iterable then each call to the mock will return the next value from the iterable.

  A *side_effect* can be cleared by setting it to *None*.

- *return_value*: The value returned when the mock is called. By default this is a new Mock (created on first access). See the `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.

Mocks can also be called with arbitrary keyword arguments. These will be used to set attributes on the mock after it is created. See the `configure_mock()` method for details.

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 name='mock.method()' id='...'>
>>> mock.method.assert_called_with(1, 2, 3, test='wow')
```

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.
```

assert_any_call(**args*, ***kwargs*)
assert the mock has been called with the specified arguments.

The assert passes if the mock has *ever* been called, unlike `assert_called_with()` and `assert_called_once_with()` that only pass if the call is the most recent one.

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, arg='thing')
>>> mock('some', 'thing', 'else')
>>> mock.assert_any_call(1, 2, arg='thing')
```

assert_has_calls(*calls*, *any_order=False*)
assert the mock has been called with the specified calls. The *mock_calls* list is checked for the calls.

If *any_order* is False (the default) then the calls must be sequential. There can be extra calls before or after the specified calls.

If *any_order* is True then the calls can be in any order, but they must all appear in `mock_calls`.

```
>>> mock = Mock(return_value=None)
>>> mock(1)
>>> mock(2)
>>> mock(3)
>>> mock(4)
>>> calls = [call(2), call(3)]
>>> mock.assert_has_calls(calls)
>>> calls = [call(4), call(2), call(3)]
>>> mock.assert_has_calls(calls, any_order=True)
```

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_mock doesn't* clear the return value, `side_effect` or any child attributes you have set using normal assignment. Child mocks and the return value mock (if any) are reset as well.

mock_add_spec(*spec*, *spec_set=False*)
Add a spec to a mock. *spec* can either be an object or a list of strings. Only attributes

on the *spec* can be fetched as attributes from the mock.

If *spec_set* is *True* then only attributes on the spec can be set.

attach_mock(*mock*, *attribute*)
Attach a mock as an attribute of this one, replacing its name and parent. Calls to the attached mock will be recorded in the `method_calls` and `mock_calls` attributes of this one.

configure_mock(***kwargs*)
Set attributes on the mock through keyword arguments.

Attributes plus return values and side effects can be set on child mocks using standard dot notation and unpacking a dictionary in the method call:

```
>>> mock = Mock()
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock.configure_mock(**attrs)
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

The same thing can be achieved in the constructor call to mocks:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

configure_mock exists to make it easier to do configuration after the mock has been created.

__dir__()
Mock objects limit the results of *dir(some_mock)* to useful results. For mocks with a *spec* this includes all the permitted attributes for the mock.

See `FILTER_DIR` for what this filtering does, and how to switch it off.

_get_child_mock(***kw*)
Create the child mocks for attributes and return value. By default child mocks will be the same type as the parent. Subclasses of Mock may want to override this to customize the way child mocks are made.

For non-callable mocks the callable variant will be used (rather than any custom subclass).

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
```

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
```

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 name='mock()()' id='...'>
>>> 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
```

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 `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()
>>> mock.side_effect = [3, 2, 1]
>>> 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 `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
```

Setting *side_effect* to *None* clears it:

```
>>> from mock import Mock
>>> m = Mock(side_effect=KeyError, return_value=3)
```

```
>>> m()
Traceback (most recent call last):
  ...
KeyError
>>> m.side_effect = None
>>> m()
3
```

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

call_args, along with members of the lists `call_args_list`, `method_calls` and `mock_calls` are `call` objects. These are tuples, so they can be unpacked to get at the individual arguments and make more complex assertions. See *calls as tuples*.

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. The `call` object can be used for conveniently constructing lists of calls to compare with *call_args_list*.

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

```
>>> mock.call_args_list == expected
True
```

Members of *call_args_list* are `call` objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

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 name='mock.method()' id='...'>
>>> mock.property.method.attribute()
<Mock name='mock.property.method.attribute()' id='...'>
>>> mock.method_calls
[call.method(), call.property.method.attribute()]
```

Members of *method_calls* are `call` objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

mock_calls

mock_calls records *all* calls to the mock object, its methods, magic methods *and* return value mocks.

```
>>> mock = MagicMock()
>>> result = mock(1, 2, 3)
>>> mock.first(a=3)
<MagicMock name='mock.first()' id='...'>
>>> mock.second()
<MagicMock name='mock.second()' id='...'>
>>> int(mock)
1
>>> result(1)
<MagicMock name='mock()()' id='...'>
>>> expected = [call(1, 2, 3), call.first(a=3), call.second(),
... call.__int__(), call()(1)]
>>> mock.mock_calls == expected
True
```

Members of *mock_calls* are `call` objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

__class__

Normally the *__class__* attribute of an object will return its type. For a mock object with a *spec* *__class__* returns the spec class instead. This allows mock objects to pass *isinstance* tests for the object they are replacing / masquerading as:

```
>>> mock = Mock(spec=3)
>>> isinstance(mock, int)
True
```

class NonCallableMock(*spec=None*, *wraps=None*, *name=None*, *spec_set=None*, ***kwargs*)

A non-callable version of *Mock*. The constructor parameters have the same meaning of *Mock*, with the exception of *return_value* and *side_effect* which have no meaning on a non-callable mock.

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
```

The *Mock* classes have support for mocking magic methods. See *magic methods* for the full details.

The mock classes and the `patch()` decorators all take arbitrary keyword arguments for configuration. For the *patch* decorators the keywords are passed to the constructor of the mock being created. The keyword arguments are for configuring attributes of the mock:

```
>>> m = MagicMock(attribute=3, other='fish')
>>> m.attribute
3
>>> m.other
'fish'
```

The return value and side effect of child mocks can be set in the same way, using dotted notation. As you can't use dotted names directly in a call you have to create a dictionary and unpack it using **:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

1.2 Calling

Mock objects are callable. The call will return the value set as the `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 like `call_args` and `call_args_list`.

If `side_effect` is set then it will be called after the call has been recorded, so if *side_effect* raises an exception the call is still recorded.

The simplest way to make a mock raise an exception when called is to make `side_effect` an exception class or instance:

```
>>> m = MagicMock(side_effect=IndexError)
>>> m(1, 2, 3)
Traceback (most recent call last):
  ...
IndexError
>>> m.mock_calls
[call(1, 2, 3)]
>>> m.side_effect = KeyError('Bang!')
>>> m('two', 'three', 'four')
Traceback (most recent call last):
  ...
KeyError: 'Bang!'
>>> m.mock_calls
[call(1, 2, 3), call('two', 'three', 'four')]
```

If *side_effect* is a function then whatever that function returns is what calls to the mock return. The *side_effect* function is called with the same arguments as the mock. This allows you to vary the return value of the call dynamically, based on the input:

```
>>> def side_effect(value):
...     return value + 1
...
>>> m = MagicMock(side_effect=side_effect)
>>> m(1)
2
>>> m(2)
3
>>> m.mock_calls
[call(1), call(2)]
```

If you want the mock to still return the default return value (a new mock), or any set return value, then there are two ways of doing this. Either return *mock.return_value* from inside *side_effect*, or return `DEFAULT`:

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

To remove a *side_effect*, and return to the default behaviour, set the *side_effect* to *None*:

```
>>> m = MagicMock(return_value=6)
>>> def side_effect(*args, **kwargs):
...     return 3
...
>>> m.side_effect = side_effect
>>> m()
3
>>> m.side_effect = None
>>> m()
6
```

The *side_effect* can also be any iterable object. Repeated calls to the mock will return values from the iterable (until the iterable is exhausted and a *StopIteration* is raised):

```
>>> m = MagicMock(side_effect=[1, 2, 3])
>>> m()
1
>>> m()
2
>>> m()
3
>>> m()
Traceback (most recent call last):
  ...
StopIteration
```

1.3 Attaching Mocks as Attributes

When you attach a mock as an attribute of another mock (or as the return value) it becomes a "child" of that mock. Calls to the child are recorded in the `method_calls` and `mock_calls` attributes of the parent. This is useful for configuring child mocks and then attaching them to the parent, or for attaching mocks to a parent that records all calls to the children and allows you to make assertions about the order of calls between mocks:

```
>>> parent = MagicMock()
>>> child1 = MagicMock(return_value=None)
>>> child2 = MagicMock(return_value=None)
>>> parent.child1 = child1
>>> parent.child2 = child2
>>> child1(1)
>>> child2(2)
>>> parent.mock_calls
[call.child1(1), call.child2(2)]
```

The exception to this is if the mock has a name. This allows you to prevent the "parenting" if for some reason you don't want it to happen.

```
>>> mock = MagicMock()
>>> not_a_child = MagicMock(name='not-a-child')
>>> mock.attribute = not_a_child
>>> mock.attribute()
<MagicMock name='not-a-child()' id='...'>
>>> mock.mock_calls
[]
```

Mocks created for you by `patch()` are automatically given names. To attach mocks that have names to a parent you use the `attach_mock()` method:

```
>>> thing1 = object()
>>> thing2 = object()
>>> parent = MagicMock()
>>> with patch('__main__.thing1', return_value=None) as child1:
...     with patch('__main__.thing2', return_value=None) as child2:
...         parent.attach_mock(child1, 'child1')
...         parent.attach_mock(child2, 'child2')
...         child1('one')
...         child2('two')
...
>>> parent.mock_calls
[call.child1('one'), call.child2('two')]
```

1.4 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.

1.4.1 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=DEFAULT*, *spec=None*, *create=False*, *mocksignature=False*, *spec_set=None*, *autospec=False*, *new_callable=None*, ***kwargs*)

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 *'package.module.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. The target is imported when the decorated function is executed, not at decoration time.

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

The *spec* and *spec_set* keyword arguments are passed to the *MagicMock* 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.

new_callable allows you to specify a different class, or callable object, that will be called to create the *new* object. By default *MagicMock* is used.

A more powerful form of *spec* is *autospec*. If you set *autospec=True* then the mock with be created with a spec from the object being replaced. All attributes of the mock will also have the spec of the corresponding attribute of the object being replaced. Methods and functions being mocked will have their arguments checked and will raise a *TypeError* if they are called with the wrong signature (similar to *mocksignature*). For mocks replacing a class, their return value (the 'instance') will have the same spec as the class. See the `create_autospec()` function and *Autospeccing*.

Instead of *autospec=True* you can pass *autospec=some_object* to use an arbitrary object as the spec instead of the one being replaced.

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* finds tests by looking for method names that start with *patch.TEST_PREFIX*. By default this is *test*, which matches the way *unittest* finds tests. You can specify an alternative prefix by setting *patch.TEST_PREFIX*.

Patch can be used as a context manager, with the with statement. 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 takes arbitrary keyword arguments. These will be passed to the *Mock* (or *new_callable*) on construction.

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

Patching a class replaces the class with a *MagicMock 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 `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:

```
>>> class Class(object):
...     def method(self):
...         pass
...
>>> with patch('__main__.Class') as MockClass:
...     instance = MockClass.return_value
...     instance.method.return_value = 'foo'
...     assert Class() is instance
...     assert Class().method() == 'foo'
...
```

If you use *spec* or *spec_set* and *patch* is replacing a *class*, then the return value of the created mock will have the same spec.

```
>>> Original = Class
>>> patcher = patch('__main__.Class', spec=True)
>>> MockClass = patcher.start()
>>> instance = MockClass()
>>> assert isinstance(instance, Original)
>>> patcher.stop()
```

The *new_callable* argument is useful where you want to use an alternative class to the default `MagicMock` for the created mock. For example, if you wanted a `NonCallableMock` to be used:

```
>>> thing = object()
>>> with patch('__main__.thing', new_callable=NonCallableMock) as mock_thing:
...     assert thing is mock_thing
...     thing()
...
Traceback (most recent call last):
  ...
TypeError: 'NonCallableMock' object is not callable
```

Another use case might be to replace an object with a *StringIO* instance:

```
>>> from StringIO import StringIO
>>> def foo():
...     print 'Something'
...
>>> @patch('sys.stdout', new_callable=StringIO)
... def test(mock_stdout):
...     foo()
...     assert mock_stdout.getvalue() == 'Something\n'
...
>>> test()
```

When *patch* is creating a mock for you, it is common that the first thing you need to do is to configure the mock. Some of that configuration can be done in the call to patch. Any arbitrary keywords you pass into the call will be used to set attributes on the created mock:

```
>>> patcher = patch('__main__.thing', first='one', second='two')
>>> mock_thing = patcher.start()
>>> mock_thing.first
'one'
>>> mock_thing.second
'two'
```

As well as attributes on the created mock attributes, like the `return_value` and `side_effect`, of child mocks can also be configured. These aren't syntactically valid to pass in directly as keyword arguments, but a dictionary with these as keys can still be expanded into a *patch* call using **:

```
>>> config = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> patcher = patch('__main__.thing', **config)
>>> mock_thing = patcher.start()
>>> mock_thing.method()
3
>>> mock_thing.other()
Traceback (most recent call last):
  ...
KeyError
```

1.4.2 patch.object

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

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

patch.object can be used as a decorator, class decorator or a context manager. Arguments *new*, *spec*, *create*, *mocksignature*, *spec_set*, *autospec* and *new_callable* have the same meaning as for *patch*. Like *patch*, *patch.object* takes arbitrary keyword arguments for configuring the mock object it creates.

When used as a class decorator *patch.object* honours *patch.TEST_PREFIX* for choosing which methods to wrap.

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, *create* and the other arguments to *patch.object* have the same meaning as they do for *patch*.

1.4.3 patch.dict

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

Patch a dictionary, or dictionary like object, 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.

patch.dict can also be called with arbitrary keyword arguments to set values in the dictionary.

patch.dict can be used as a context manager, decorator or class decorator. When used as a class decorator *patch.dict* honours *patch.TEST_PREFIX* for choosing which methods to wrap.

patch.dict 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
```

Keywords can be used in the *patch.dict* call to set values in the dictionary:

```
>>> mymodule = MagicMock()
>>> mymodule.function.return_value = 'fish'
>>> with patch.dict('sys.modules', mymodule=mymodule):
...     import mymodule
...     mymodule.function('some', 'args')
...
'fish'
```

patch.dict can be used with dictionary like objects that aren't actually dictionaries. At the very minimum they must support item getting, setting, deleting and either iteration or membership test. This corresponds to the magic methods *__getitem__*, *__setitem__*, *__delitem__* and either *__iter__* or *__contains__*.

```
>>> class Container(object):
...     def __init__(self):
...         self.values = {}
...     def __getitem__(self, name):
...         return self.values[name]
...     def __setitem__(self, name, value):
...         self.values[name] = value
...     def __delitem__(self, name):
...         del self.values[name]
...     def __iter__(self):
...         return iter(self.values)
...
>>> thing = Container()
>>> thing['one'] = 1
>>> with patch.dict(thing, one=2, two=3):
...     assert thing['one'] == 2
...     assert thing['two'] == 3
...
>>> assert thing['one'] == 1
>>> assert list(thing) == ['one']
```

1.4.4 patch.multiple

patch.**multiple**(*target*, *spec=None*, *create=False*, *mocksignature=False*, *spec_set=None*, *autospec=False*, *new_callable=None*, ***kwargs*)

Perform multiple patches in a single call. It takes the object to be patched (either as an object or a string to fetch the object by importing) and keyword arguments for the patches:

```
with patch.multiple(settings, FIRST_PATCH='one', SECOND_PATCH='two'):
    ...
```

Use DEFAULT as the value if you want *patch.multiple* to create mocks for you. In this case the created mocks are passed into a decorated function by keyword, and a dictionary is returned when *patch.multiple* is used as a context manager.

patch.multiple can be used as a decorator, class decorator or a context manager. The arguments *spec*, *spec_set*, *create*, *mocksignature*, *autospec* and *new_callable* have the same meaning as for *patch*. These arguments will be applied to *all* patches done by *patch.multiple*.

When used as a class decorator *patch.multiple* honours *patch.TEST_PREFIX* for choosing which methods to wrap.

If you want *patch.multiple* to create mocks for you, then you can use DEFAULT as the value. If you use *patch.multiple* as a decorator then the created mocks are passed into the decorated function by keyword.

```
>>> thing = object()
>>> other = object()

>>> @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(thing, other):
...     assert isinstance(thing, MagicMock)
...     assert isinstance(other, MagicMock)
...
>>> test_function()
```

patch.multiple can be nested with other *patch* decorators, but put arguments passed by keyword *after* any of the standard arguments created by *patch*:

```
>>> @patch('sys.exit')
... @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(mock_exit, other, thing):
...     assert 'other' in repr(other)
...     assert 'thing' in repr(thing)
...     assert 'exit' in repr(mock_exit)
...
>>> test_function()
```

If *patch.multiple* is used as a context manager, the value returned by the context manger is a dictionary where created mocks are keyed by name:

```
>>> with patch.multiple('__main__', thing=DEFAULT, other=DEFAULT) as values:
...     assert 'other' in repr(values['other'])
...     assert 'thing' in repr(values['thing'])
...     assert values['thing'] is thing
...     assert values['other'] is other
...
```

1.4.5 patch methods: start and stop

All the 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.

1.4.6 TEST_PREFIX

All of the patchers can be used as class decorators. When used in this way they wrap every test method on the class. The patchers recognise methods that start with *test* as being test methods. This is the same way that the *unittest.TestLoader* finds test methods by default.

It is possible that you want to use a different prefix for your tests. You can inform the patchers of the different prefix by setting *patch.TEST_PREFIX*:

```
>>> patch.TEST_PREFIX = 'foo'
>>> value = 3
>>>
>>> @patch('__main__.value', 'not three')
... class Thing(object):
...     def foo_one(self):
...         print value
...     def foo_two(self):
...         print value
...
>>>
>>> Thing().foo_one()
not three
>>> Thing().foo_two()
not three
>>> value
3
```

1.4.7 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
...
```

1.4.8 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 *looked up*, 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')

1.4.9 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...

1.5 Helpers

1.5.1 call

call(**args*, ***kwargs*)

call is a helper object for making simpler assertions, for comparing with `call_args`, `call_args_list`, `mock_calls` and:attr: *~Mock.method_calls*. *call* can also be used with `assert_has_calls()`.

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, a='foo', b='bar')
>>> m()
>>> m.call_args_list == [call(1, 2, a='foo', b='bar'), call()]
True
```

call.**call_list**()

For a call object that represents multiple calls, *call_list* returns a list of all the intermediate calls as well as the final call.

call_list is particularly useful for making assertions on "chained calls". A chained call is multiple calls on a single line of code. This results in multiple entries in `mock_calls` on a mock. Manually constructing the sequence of calls can be tedious.

`call_list()` can construct the sequence of calls from the same chained call:

```
>>> m = MagicMock()
>>> m(1).method(arg='foo').other('bar')(2.0)
<MagicMock name='mock().method().other()()' id='...'>
>>> kall = call(1).method(arg='foo').other('bar')(2.0)
>>> kall.call_list()
```

```
[call(1),
 call().method(arg='foo'),
 call().method().other('bar'),
 call().method().other()(2.0)]
>>> m.mock_calls == kall.call_list()
True
```

A *call* object is either a tuple of (positional args, keyword args) or (name, positional args, keyword args) depending on how it was constructed. When you construct them yourself this isn't particularly interesting, but the *call* objects that are in the `Mock.call_args`, `Mock.call_args_list` and `Mock.mock_calls` attributes can be introspected to get at the individual arguments they contain.

The *call* objects in `Mock.call_args` and `Mock.call_args_list` are two-tuples of (positional args, keyword args) whereas the *call* objects in `Mock.mock_calls`, along with ones you construct yourself, are three-tuples of (name, positional args, keyword args).

You can use their "tupleness" to pull out the individual arguments for more complex introspection and assertions. The positional arguments are a tuple (an empty tuple if there are no positional arguments) and the keyword arguments are a dictionary:

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, 3, arg='one', arg2='two')
>>> kall = m.call_args
>>> args, kwargs = kall
>>> args
(1, 2, 3)
>>> kwargs
{'arg2': 'two', 'arg': 'one'}
>>> args is kall[0]
True
>>> kwargs is kall[1]
True

>>> m = MagicMock()
>>> m.foo(4, 5, 6, arg='two', arg2='three')
<MagicMock name='mock.foo()' id='...'>
>>> kall = m.mock_calls[0]
>>> name, args, kwargs = kall
>>> name
'foo'
>>> args
(4, 5, 6)
>>> kwargs
{'arg2': 'three', 'arg': 'two'}
>>> name is m.mock_calls[0][0]
True
```

1.5.2 create_autospec

create_autospec(*spec*, *spec_set=False*, *instance=False*, ***kwargs*)

Create a mock object using another object as a spec. Attributes on the mock will use the corresponding attribute on the *spec* object as their spec.

Functions or methods being mocked will have their arguments checked in a similar way to `mocksignature()` to check that they are called with the correct signature.

If *spec_set* is *True* then attempting to set attributes that don't exist on the spec object will raise an *AttributeError*.

If a class is used as a spec then the return value of the mock (the instance of the class) will have the same spec. You can use a class as the spec for an instance object by passing *instance=True*. The returned mock will only be callable if instances of the mock are callable.

create_autospec also takes arbitrary keyword arguments that are passed to the constructor of the created mock.

See *Autospeccing* for examples of how to use auto-speccing with *create_autospec* and the *autospec* argument to `patch()`.

1.5.3 ANY

ANY

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 `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 `assert_called_with()` and `assert_called_once_with()` will then succeed no matter what was passed in.

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

ANY can also be used in comparisons with call lists like `mock_calls`:

```
>>> m = MagicMock(return_value=None)
>>> m(1)
>>> m(1, 2)
>>> m(object())
>>> m.mock_calls == [call(1), call(1, 2), ANY]
True
```

1.5.4 FILTER_DIR

FILTER_DIR

FILTER_DIR is a module level variable that controls the way mock objects respond to *dir* (only for Python 2.6 or more recent). The default is *True*, which uses the filtering described below, to only show useful members. If you dislike this filtering, or need to switch it off for diagnostic purposes, then set *mock.FILTER_DIR = False*.

With filtering on, *dir(some_mock)* shows only useful attributes and will include any dynamically created attributes that wouldn't normally be shown. If the mock was created with a *spec* (or *autospec* of course) then all the attributes from the original are shown, even if they haven't been accessed yet:

```
>>> dir(Mock())
['assert_any_call',
 'assert_called_once_with',
 'assert_called_with',
 'assert_has_calls',
 'attach_mock',
 ...
>>> import urllib2
>>> dir(Mock(spec=urllib2))
['AbstractBasicAuthHandler',
 'AbstractDigestAuthHandler',
 'AbstractHTTPHandler',
 'BaseHandler',
 ...
```

Many of the not-very-useful (private to *Mock* rather than the thing being mocked) underscore and double underscore prefixed attributes have been filtered from the result of calling *dir* on a *Mock*. If you dislike this behaviour you can switch it off by setting the module level switch *FILTER_DIR*:

```
>>> import mock
>>> mock.FILTER_DIR = False
>>> dir(mock.Mock())
['_NonCallableMock__get_return_value',
 '_NonCallableMock__get_side_effect',
 '_NonCallableMock__return_value_doc',
 '_NonCallableMock__set_return_value',
 '_NonCallableMock__set_side_effect',
 '__call__',
 '__class__',
 ...
```

Alternatively you can just use *vars(my_mock)* (instance members) and *dir(type(my_mock))* (type members) to bypass the filtering irrespective of *mock.FILTER_DIR*.

1.5.5 Autospeccing

Autospeccing is based on the existing *spec* feature of mock. It limits the api of mocks to the api of an original object (the spec), but it is recursive (implemented lazily) so that attributes of mocks only have the same api as the attributes of the spec. In addition mocked functions / methods have the same call signature as the original so they raise a *TypeError* if they are called incorrectly. This feature is a better version of both the *spec* and *mocksignature* features in the current mock.

Before I explain how auto-speccing works, here's why it is needed.

Mock is a very powerful and flexible object, but it suffers from two flaws when used to mock out objects from a system under test. One of these flaws is specific to the *Mock* api and the other is a more general problem with using mock objects.

First the problem specific to *Mock*. *Mock* has two assert methods that are extremely handy: `assert_called_with()` and `assert_called_once_with()`.

```
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
Traceback (most recent call last):
  ...
AssertionError: Expected to be called once. Called 2 times.
```

Because mocks auto-create attributes on demand, and allow you to call them with arbitrary arguments, if you misspell one of these assert methods then your assertion is gone:

```
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assret_called_once_with(4, 5, 6)
```

Your tests can pass silently and incorrectly because of the typo.

The second issue is more general to mocking. If you refactor some of your code, rename members and so on, any tests for code that is still using the *old api* but uses mocks instead of the real objects will still pass. This means your tests can all pass even though your code is broken.

Note that this is another reason why you need integration tests as well as unit tests. Testing everything in isolation is all fine and dandy, but if you don't test how your units are "wired together" there is still lots of room for bugs that tests might have caught.

mock already provides a feature to help with this, called speccing. If you use a class or instance as the *spec* for a mock then you can only access attributes on the mock that exist on the real class:

```
>>> import urllib2
>>> mock = Mock(spec=urllib2.Request)
>>> mock.assret_called_with
Traceback (most recent call last):
```

```
  ...
AttributeError: Mock object has no attribute 'assret_called_with'
```

The spec only applies to the mock itself, so we still have the same issue with any methods on the mock:

```
>>> mock.has_data()
<mock.Mock object at 0x...>
>>> mock.has_data.assret_called_with()
```

Auto-speccing solves this problem, and combines the features of *mocksignature*. You can either pass *autospec=True* to *patch* / *patch.object* or use the *create_autospec* function to create a mock with a spec. If you use the *autospec=True* argument to *patch* then the object that is being replaced will be used as the spec object. Because the speccing is done "lazily" (the spec is created as attributes on the mock are accessed) you can use it with very complex or deeply nested objects (like modules that import modules that import modules) without a big performance hit.

Here's an example of it in use:

```
>>> import urllib2
>>> patcher = patch('__main__.urllib2', autospec=True)
>>> mock_urllib2 = patcher.start()
>>> urllib2 is mock_urllib2
True
>>> urllib2.Request
<MagicMock name='urllib2.Request' spec='Request' id='...'>
```

You can see that *urllib2.Request* has a spec. *urllib2.Request* takes two arguments in the constructor (one of which is *self*). Here's what happens if we try to call it incorrectly:

```
>>> req = urllib2.Request()
Traceback (most recent call last):
  ...
TypeError: <lambda>() takes at least 2 arguments (1 given)
```

The spec also applies to instantiated classes (i.e. the return value of specced mocks):

```
>>> req = urllib2.Request('foo')
>>> req
<NonCallableMagicMock name='urllib2.Request()' spec='Request' id='...'>
```

Request objects are not callable, so the return value of instantiating our mocked out *urllib2.Request* is a non-callable mock. With the spec in place any typos in our asserts will raise the correct error:

```
>>> req.add_header('spam', 'eggs')
<MagicMock name='urllib2.Request().add_header()' id='...'>
>>> req.add_header.assret_called_with
Traceback (most recent call last):
  ...
```

```
AttributeError: Mock object has no attribute 'assret_called_with'
>>> req.add_header.assert_called_with('spam', 'eggs')
```

In many cases you will just be able to add *autospec=True* to your existing *patch* calls and then be protected against bugs due to typos and api changes.

As well as using *autospec* through *patch* there is a `create_autospec()` for creating autospecced mocks directly:

```
>>> import urllib2
>>> mock_urllib2 = create_autospec(urllib2)
>>> mock_urllib2.Request('foo', 'bar')
<NonCallableMagicMock name='mock.Request()' spec='Request' id='...'>
```

This isn't without caveats and limitations however, which is why it is not the default behaviour. In order to know what attributes are available on the spec object, autospec has to introspect (access attributes) the spec. As you traverse attributes on the mock a corresponding traversal of the original object is happening under the hood. If any of your specced objects have properties or descriptors that can trigger code execution then you may not be able to use autospec. On the other hand it is much better to design your objects so that introspection is safe [2].

A more serious problem is that it is common for instance attributes to be created in the *__init__* method and not to exist on the class at all. *autospec* can't know about any dynamically created attributes and restricts the api to visible attributes.

```
>>> class Something(object):
...   def __init__(self):
...     self.a = 33
...
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

There are a few different ways of resolving this problem. The easiest, but not necessarily the least annoying, way is to simply set the required attributes on the mock after creation. Just because *autospec* doesn't allow you to fetch attributes that don't exist on the spec it doesn't prevent you setting them:

```
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
```

[2] This only applies to classes or already instantiated objects. Calling a mocked class to create a mock instance *does not* create a real instance. It is only attribute lookups - along with calls to *dir* - that are done. A way round this problem would have been to use getattr_static, which can fetch attributes without triggering code execution. Descriptors like *classmethod* and *staticmethod need* to be fetched correctly though, so that their signatures can be mocked correctly.

```
...     thing.a = 33
...
```

There is a more aggressive version of both *spec* and *autospec* that *does* prevent you setting non-existent attributes. This is useful if you want to ensure your code only *sets* valid attributes too, but obviously it prevents this particular scenario:

```
>>> with patch('__main__.Something', autospec=True, spec_set=True):
...     thing = Something()
...     thing.a = 33
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

Probably the best way of solving the problem is to add class attributes as default values for instance members initialised in *__init__*. Note that if you are only setting default attributes in *__init__* then providing them via class attributes (shared between instances of course) is faster too. e.g.

```
class Something(object):
    a = 33
```

This brings up another issue. It is relatively common to provide a default value of *None* for members that will later be an object of a different type. *None* would be useless as a spec because it wouldn't let you access *any* attributes or methods on it. As *None* is *never* going to be useful as a spec, and probably indicates a member that will normally of some other type, *autospec* doesn't use a spec for members that are set to *None*. These will just be ordinary mocks (well - *MagicMocks*):

```
>>> class Something(object):
...     member = None
...
>>> mock = create_autospec(Something)
>>> mock.member.foo.bar.baz()
<MagicMock name='mock.member.foo.bar.baz()' id='...'>
```

If modifying your production classes to add defaults isn't to your liking then there are more options. One of these is simply to use an instance as the spec rather than the class. The other is to create a subclass of the production class and add the defaults to the subclass without affecting the production class. Both of these require you to use an alternative object as the spec. Thankfully *patch* supports this - you can simply pass the alternative object as the *autospec* argument:

```
>>> class Something(object):
...   def __init__(self):
...     self.a = 33
...
>>> class SomethingForTest(Something):
...   a = 33
...
```

```
>>> p = patch('__main__.Something', autospec=SomethingForTest)
>>> mock = p.start()
>>> mock.a
<NonCallableMagicMock name='Something.a' spec='int' id='...'>
```

Note: An additional limitation (currently) with *autospec* is that unbound methods on mocked classes *don't* take an "explicit self" as the first argument - so this usage will fail with *autospec*.

```
>>> class Foo(object):
...   def foo(self):
...     pass
...
>>> Foo.foo(Foo())
>>> MockFoo = create_autospec(Foo)
>>> MockFoo.foo(MockFoo())
Traceback (most recent call last):
  ...
TypeError: <lambda>() takes exactly 1 argument (2 given)
```

The reason is that its very hard to tell the difference between functions, unbound methods and staticmethods across Python 2 & 3 and the alternative implementations. This restriction may be fixed in future versions.

1.6 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 `side_effect` functions to indicate that the normal return value should be used.

1.6.1 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
sentinel.some_object
```

1.7 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 [3], 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 [4].

```
>>> 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:

[3] 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.

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

```
>>> mock = Mock()
>>> mock.__enter__ = Mock(return_value='foo')
>>> mock.__exit__ = Mock(return_value=False)
>>> with mock as m:
...     assert m == 'foo'
...
>>> mock.__enter__.assert_called_with()
>>> mock.__exit__.assert_called_with(None, None, None)
```

Calls to magic methods do not appear in `method_calls`, but they are recorded in `mock_calls`.

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__`, `__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__`
- `__truediv__` and `__rtruediv__`

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__

1.8 Magic Mock

There are two *MagicMock* variants: *MagicMock* and *NonCallableMagicMock*.

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.

The constructor parameters have the same meaning as for Mock.

If you use the *spec* or *spec_set* arguments then *only* magic methods that exist in the spec will be created.

class NonCallableMagicMock (**args*, ***kw*)

A non-callable version of *MagicMock*.

The constructor parameters have the same meaning as for MagicMock, with the exception of *return_value* and *side_effect* which have no meaning on a non-callable mock.

The magic methods are setup with *MagicMock* 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
```

In *0.8* the *__iter__* also gained special handling implemented with a side effect. The return value of *MagicMock.__iter__* can be any iterable object and isn't required to be an iterator:

```
>>> mock = MagicMock()
>>> mock.__iter__.return_value = ['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
```

If the return value *is* an iterator, then iterating over it once will consume it and subsequent iterations will result in an empty list:

```
>>> mock.__iter__.return_value = iter(['a', 'b', 'c'])
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
[]
```

`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__`

1.9 mocksignature

Note: *Autospeccing*, added in mock 0.8, is a more advanced version of *mocksignature* and can be used for many of the same use cases.

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 function objects.

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.

Functions returned by *mocksignature* have many of the same attributes and assert methods as a mock object.

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.

1.9.1 mocksignature api

Although the objects returned by *mocksignature* api are real function objects, they have much of the same api as the `Mock` class. This includes the assert methods:

```
>>> def func(a, b, c):
...     pass
...
>>> func2 = mocksignature(func)
>>> func2.called
False
>>> func2.return_value = 3
```

```
>>> func2(1, 2, 3)
3
>>> func2.called
True
>>> func2.assert_called_once_with(1, 2, 3)
>>> func2.assert_called_with(1, 2, 4)
Traceback (most recent call last):
  ...
AssertionError: Expected call: mock(1, 2, 4)
Actual call: mock(1, 2, 3)
>>> func2.call_count
1
>>> func2.side_effect = IndexError
>>> func2(4, 5, 6)
Traceback (most recent call last):
  ...
IndexError
```

The mock object that is being delegated to is available as the *mock* attribute of the function created by *mocksignature*.

```
>>> func2.mock.mock_calls
[call(1, 2, 3), call(4, 5, 6)]
```

The methods and attributes available on functions returned by *mocksignature* are:

`assert_any_call()`, `assert_called_once_with()`, `assert_called_with()`, `assert_has_calls()`, `call_args`, `call_args_list`, `call_count`, `called`, `method_calls`, *mock*, `mock_calls`, `reset_mock()`, `return_value`, and `side_effect`.

1.9.2 Example use

Basic use

```
>>> 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)
>>> function.return_value = 'some value'
>>> function(1, 2, 'foo')
```

```
'some value'
>>> function.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:

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

As a curious factor of the way Python (2) wraps methods fetched from a class, we can *get* the *return_value* from a function set on a class, but we can't set it. We have to do this through the exposed *mock* attribute instead:

```
>>> 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.assert_called_with(instance, 1, 2, 3)
```

When you use *mocksignature* on instance methods *self* isn't included (and we can set the *return_value* etc directly):

```
>>> class SomeClass(object):
...     def method(self, a, b, c=None):
...         pass
...
```

```
>>> instance = SomeClass()
>>> instance.method = mocksignature(instance.method)
>>> instance.method.return_value = None
>>> instance.method(1, 2, 3)
>>> instance.method.assert_called_with(1, 2, 3)
```

mocksignature with classes

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

```
>>> class Something(object):
...     def __init__(self, foo, bar):
...         pass
...
>>> MockSomething = mocksignature(Something)
>>> instance = MockSomething(10, 9)
>>> assert instance is MockSomething.return_value
>>> MockSomething.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.

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

1.9.3 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.

```
>>> 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

2.1.1 Using Mock

Mock Patching Methods

Common uses for `Mock` objects include:

- Patching methods
- Recording method calls on objects

You might want to replace a method on an object to check that it is called with the correct arguments by another part of the system:

```
>>> real = SomeClass()
>>> real.method = MagicMock(name='method')
>>> real.method(3, 4, 5, key='value')
<MagicMock name='method()' id='...'>
```

Once our mock has been used (*real.method* in this example) it has methods and attributes that allow you to make assertions about how it has been used.

Note: In most of these examples the `Mock` and `MagicMock` classes are interchangeable. As the *MagicMock* is the more capable class it makes a sensible one to use by default.

Once the mock has been called its `called` attribute is set to *True*. More importantly we can use the `assert_called_with()` or :meth‘~Mock.assert_called_once_with‘ method to check that it was called with the correct arguments.

This example tests that calling *ProductionClass().method* results in a call to the *something* method:

```
>>> from mock import MagicMock
>>> class ProductionClass(object):
...     def method(self):
...         self.something(1, 2, 3)
...     def something(self, a, b, c):
...         pass
...
>>> real = ProductionClass()
>>> real.something = MagicMock()
>>> real.method()
>>> real.something.assert_called_once_with(1, 2, 3)
```

Mock for Method Calls on an Object

In the last example we patched a method directly on an object to check that it was called correctly. Another common use case is to pass an object into a method (or some part of the system under test) and then check that it is used in the correct way.

The simple *ProductionClass* below has a *closer* method. If it is called with an object then it calls *close* on it.

```
>>> class ProductionClass(object):
...     def closer(self, something):
...         something.close()
...
```

So to test it we need to pass in an object with a *close* method and check that it was called correctly.

```
>>> 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 `assert_called_with()` will raise a failure exception.

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. The call to *patch* replaces the class *Foo* with a mock. The *Foo* instance is the result of calling the mock, so it is configured by modify 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 = MagicMock(name='foo')
>>> mock
<MagicMock name='foo' id='...'>
>>> mock.method
<MagicMock name='foo.method' id='...'>
```

Tracking all Calls

Often you want to track more than a single call to a method. The `mock_calls` attribute records all calls to child attributes of the mock - and also to their children.

```
>>> mock = MagicMock()
>>> mock.method()
<MagicMock name='mock.method()' id='...'>
>>> mock.attribute.method(10, x=53)
<MagicMock name='mock.attribute.method()' id='...'>
>>> mock.mock_calls
[call.method(), call.attribute.method(10, x=53)]
```

If you make an assertion about *mock_calls* and any unexpected methods have been called, then the assertion will fail. This is useful because as well as asserting that the calls you expected have been made, you are also checking that they were made in the right order and with no additional calls:

You use the `call` object to construct lists for comparing with *mock_calls*:

```
>>> expected = [call.method(), call.attribute.method(10, x=53)]
>>> mock.mock_calls == expected
True
```

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")*. If we wanted this call to return a list, then we have to configure the result of the nested call.

We can use `call` to construct the set of calls in a "chained call" like this for easy assertion afterwards:

```
>>> mock = Mock()
>>> cursor = mock.connection.cursor.return_value
>>> cursor.execute.return_value = ['foo']
>>> mock.connection.cursor().execute("SELECT 1")
['foo']
>>> expected = call.connection.cursor().execute("SELECT 1").call_list()
>>> mock.mock_calls
[call.connection.cursor(), call.connection.cursor().execute('SELECT 1')]
>>> mock.mock_calls == expected
True
```

It is the call to *.call_list()* that turns our call object into a list of calls representing the chained calls.

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.

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

Side effect functions and iterables

side_effect can also be set to a function or an iterable. The use case for *side_effect* as an iterable is where your mock is going to be called several times, and you want each call to return a different value. When you set *side_effect* to an iterable every call to the mock returns the next value from the iterable:

```
>>> mock = MagicMock(side_effect=[4, 5, 6])
>>> mock()
4
>>> mock()
5
>>> mock()
6
```

For more advanced use cases, like dynamically varying the return values depending on what the mock is called with, *side_effect* can be a function. The function will be called with the same arguments as the mock. Whatever the function returns is what the call returns:

```
>>> vals = {(1, 2): 1, (2, 3): 2}
>>> def side_effect(*args):
...     return vals[args]
...
>>> mock = MagicMock(side_effect=side_effect)
>>> mock(1, 2)
1
>>> mock(2, 3)
2
```

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'
```

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*.

2.1.2 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 = MagicMock(return_value = sentinel.file_handle)
>>> with patch('__builtin__.open', mock):
...     handle = open('filename', 'r')
...
>>> 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()
```

A nice pattern is to actually decorate test methods themselves:

```
>>> class MyTest(unittest2.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(unittest2.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(unittest2.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".

For some more advanced examples, see the *Further Examples* page.

2.2 Further Examples

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

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 `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 name='mock().foo()' id='...'>
>>> 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', 'eg
...         # 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_call.return_value = 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()
```

Using `mock_calls` we can check the chained call with a single assert. A chained call is several calls in one line of code, so there will be several entries in *mock_calls*. We can use `call.call_list()` to create this list of calls for us:

```
>>> chained = call.get_endpoint('foobar').create_call('spam', 'eggs').start_call()
>>> call_list = chained.call_list()
>>> assert mock_backend.mock_calls == call_list
```

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.

An alternative way of dealing with mocking dates, or other builtin classes, is discussed in this blog entry.

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).

Mocking context managers with a `MagicMock` is common enough and fiddly enough that a helper function is useful. Here *mock_open* creates and configures a *MagicMock* that behaves as a file context manager. You can optionally supply a *StringIO* object as the *data* argument for the actual file handle:

```
from mock import inPy3k, MagicMock
if inPy3k:
    file_spec = ['_CHUNK_SIZE', '__enter__', '__eq__', '__exit__',
        '__format__', '__ge__', '__gt__', '__hash__', '__iter__', '__le__',
        '__lt__', '__ne__', '__next__', '__repr__', '__str__',
        '_checkClosed', '_checkReadable', '_checkSeekable',
        '_checkWritable', 'buffer', 'close', 'closed', 'detach',
        'encoding', 'errors', 'fileno', 'flush', 'isatty',
        'line_buffering', 'mode', 'name',
        'newlines', 'peek', 'raw', 'read', 'read1', 'readable',
        'readinto', 'readline', 'readlines', 'seek', 'seekable', 'tell',
        'truncate', 'writable', 'write', 'writelines']
else:
    file_spec = file

def mock_open(mock=None, data=None):
    if mock is None:
        mock = MagicMock(spec=file_spec)

    handle = MagicMock(spec=file_spec)
    handle.write.return_value = None
    if data is None:
        handle.__enter__.return_value = handle
    else:
        handle.__enter__.return_value = data
    mock.return_value = handle
    return mock
```

```
>>> m = mock_open()
>>> with patch('__main__.open', m, create=True):
...     with open('foo', 'w') as h:
...         h.write('some stuff')
...
>>> m.mock_calls
[call('foo', 'w'),
 call().__enter__(),
 call().write('some stuff'),
 call().__exit__(None, None, None)]
>>> m.assert_called_once_with('foo', 'w')
>>> handle = m()
>>> handle.write.assert_called_once_with('some stuff')
```

And for reading files, using a *StringIO* to represent the actual file handle:

```
>>> from StringIO import StringIO
>>> m = mock_open(data=StringIO('foo bar baz'))
>>> with patch('__main__.open', m, create=True):
...     with open('foo') as h:
...         result = h.read()
...
>>> m.assert_called_once_with('foo')
>>> assert result == 'foo bar baz'
```

Let's step through what's happening here in more detail.

To mock *open* we need 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 `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')
...
<MagicMock name='open().__enter__().write()' id='...'>
>>> file_handle = mock_open.return_value.__enter__.return_value
>>> file_handle.write.assert_called_with('something')
```

2.2.4 Mocks without some attributes

Mock objects create attributes on demand. This allows them to pretend to be objects of any type.

What mocks aren't so good at is pretending *not* to have attributes. You may want a mock object to return *False* to a *hasattr* call, or raise an *AttributeError* when an attribute is fetched. You can do this by providing an object as a *spec* for a mock, but that isn't always convenient.

Below is a subclass of *Mock* that allows you to "block" attributes by deleting them. Once deleted, accessing an attribute will raise an *AttributeError*.

```
deleted = object()
missing = object()

class DeletingMock(Mock):
    def __delattr__(self, attr):
        if attr in self.__dict__:
            return super(DeletingMock, self).__delattr__(attr)
        obj = self._mock_children.get(attr, missing)
        if obj is deleted:
            raise AttributeError(attr)
        if obj is not missing:
            del self._mock_children[attr]
        self._mock_children[attr] = deleted

    def __getattr__(self, attr):
        result = super(DeletingMock, self).__getattr__(attr)
        if result is deleted:
            raise AttributeError(attr)
        return result

>>> mock = DeletingMock()
>>> hasattr(mock, 'm')
True
```

```
>>> del mock.m
>>> hasattr(mock, 'm')
False
>>> del mock.f
>>> mock.f
Traceback (most recent call last):
    ...
AttributeError: f
```

2.2.5 Mocking a Generator Method

A Python generator is a function or method that uses the yield statement to return a series of values when iterated over [1].

A generator method / function is called to return the generator object. It is the generator object that is then iterated over. The protocol method for iteration is __iter__, so we can mock this using a *MagicMock*.

Here's an example class with an "iter" method implemented as a generator:

```
>>> class Foo(object):
...     def iter(self):
...         for i in [1, 2, 3]:
...             yield i
...
>>> foo = Foo()
>>> list(foo.iter())
[1, 2, 3]
```

How would we mock this class, and in particular its "iter" method?

To configure the values returned from the iteration (implicit in the call to *list*), we need to configure the object returned by the call to *foo.iter()*.

```
>>> mock_foo = MagicMock()
>>> mock_foo.iter.return_value = iter([1, 2, 3])
>>> list(mock_foo.iter())
[1, 2, 3]
```

2.2.6 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

[1] There are also generator expressions and more advanced uses of generators, but we aren't concerned about them here. A very good introduction to generators and how powerful they are is: Generator Tricks for Systems Programmers.

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()
```

2.2.7 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*.

2.2.8 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()
```

This approach works fine if you can replace the whole object you're mocking. If you *just* want to mock the property on another object here's an alternative approach using the support for magic methods introduced in 0.7:

```
>>> class Foo(object):
...    @property
...    def fish(self):
...      return 'fish'
...
>>> with patch.object(Foo, 'fish') as mock_fish:
...   mock_fish.__get__ = Mock(return_value='mocked fish')
...   foo = Foo()
...   print foo.fish
...
mocked fish
>>> mock_fish.__get__.assert_called_with(mock_fish, foo, Foo)
```

If you're using an earlier version of mock, a third approach is to subclass `Mock` and provide a `__get__` method that delegates back to the mock:

```
>>> class PropertyMock(Mock):
...   def __get__(self, instance, owner):
...     return self()
...
>>> prop_mock = PropertyMock()
>>> with patch.object(Foo, 'fish', prop_mock):
...   foo = Foo()
...   prop_mock.return_value = 'mocked fish'
...   print foo.fish
...
mocked fish
>>> prop_mock.assert_called_with()
```

As you're patching on the class these techniques affect *all* instances of *Foo*.

2.2.9 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 you can use `call_args_list`:

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

The `call` helper makes it easy to make assertions about these calls. You can build up a list of expected calls and compare it to *call_args_list*. This looks remarkably similar to the repr of the *call_args_list*:

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

2.2.10 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
call(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
```

An alternative approach is to create a subclass of *Mock* or *MagicMock* that copies (using copy.deepcopy) the arguments. Here's an example implementation:

```
>>> from copy import deepcopy
>>> class CopyingMock(MagicMock):
...     def __call__(self, *args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         return super(CopyingMock, self).__call__(*args, **kwargs)
...
>>> c = CopyingMock(return_value=None)
>>> arg = set()
>>> c(arg)
>>> arg.add(1)
>>> c.assert_called_with(set())
>>> c.assert_called_with(arg)
Traceback (most recent call last):
    ...
AssertionError: Expected call: mock(set([1]))
Actual call: mock(set([]))
>>> c.foo
<CopyingMock name='mock.foo' id='...'>
```

When you subclass *Mock* or *MagicMock* all dynamically created attributes, and the *return_value* will use your subclass automatically. That means all children of a *CopyingMock* will also have the type *CopyingMock*.

2.2.11 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.

2.2.12 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
```

2.2.13 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 `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 `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
[call('a'), call('c'), call('d'), call('b'), call('d')]
>>> mock.__setitem__.call_args_list
[call('b', 'fish'), call('d', 'eggs')]
>>> my_dict
{'a': 1, 'c': 3, 'b': 'fish', 'd': 'eggs'}
```

2.2.14 Mock subclasses and their attributes

There are various reasons why you might want to subclass *Mock*. One reason might be to add helper methods. Here's a silly example:

```
>>> class MyMock(MagicMock):
...     def has_been_called(self):
...         return self.called
...
>>> mymock = MyMock(return_value=None)
>>> mymock
<MyMock id='...'>
>>> mymock.has_been_called()
False
>>> mymock()
>>> mymock.has_been_called()
True
```

The standard behaviour for *Mock* instances is that attributes and the return value mocks are of the same type as the mock they are accessed on. This ensures that *Mock* attributes are *Mocks* and *MagicMock* attributes are *MagicMocks* [2]. So if you're subclassing to add helper methods then they'll also be available on the attributes and return value mock of instances of your subclass.

[2] An exception to this rule are the non-callable mocks. Attributes use the callable variant because otherwise non-callable mocks couldn't have callable methods.

```
>>> mymock.foo
<MyMock name='mock.foo' id='...'>
>>> mymock.foo.has_been_called()
False
>>> mymock.foo()
<MyMock name='mock.foo()' id='...'>
>>> mymock.foo.has_been_called()
True
```

Sometimes this is inconvenient. For example, one user is subclassing mock to created a Twisted adaptor. Having this applied to attributes too actually causes errors.

Mock (in all its flavours) uses a method called *_get_child_mock* to create these "sub-mocks" for attributes and return values. You can prevent your subclass being used for attributes by overriding this method. The signature is that it takes arbitrary keyword arguments (***kwargs*) which are then passed onto the mock constructor:

```
>>> class Subclass(MagicMock):
...     def _get_child_mock(self, **kwargs):
...         return MagicMock(**kwargs)
...
>>> mymock = Subclass()
>>> mymock.foo
<MagicMock name='mock.foo' id='...'>
>>> assert isinstance(mymock, Subclass)
>>> assert not isinstance(mymock.foo, Subclass)
>>> assert not isinstance(mymock(), Subclass)
```

2.2.15 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 name='mock.blob()' id='...'>
>>> 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 name='mock.blob.blip()' id='...'>
>>> 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 name='mock.module.fooble()' id='...'>
>>> 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.)

2.2.16 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 `method_calls` attribute. This doesn't allow you to track the order of calls between separate mock objects, however we can use `mock_calls` to achieve the same effect.

Because mocks track calls to child mocks in *mock_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 *mock_calls* of the parent:

```
>>> manager = Mock()
>>> mock_foo = manager.foo
>>> mock_bar = manager.bar

>>> mock_foo.something()
<Mock name='mock.foo.something()' id='...'>
>>> mock_bar.other.thing()
<Mock name='mock.bar.other.thing()' id='...'>

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

We can then assert about the calls, including the order, by comparing with the *mock_calls* attribute on the manager mock:

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

If *patch* is creating, and putting in place, your mocks then you can attach them to a manager mock using the `attach_mock()` method. After attaching calls will be recorded in *mock_calls* of the manager.

```
>>> manager = MagicMock()
>>> with patch('mymodule.Class1') as MockClass1:
...     with patch('mymodule.Class2') as MockClass2:
...         manager.attach_mock(MockClass1, 'MockClass1')
...         manager.attach_mock(MockClass2, 'MockClass2')
...         MockClass1().foo()
...         MockClass2().bar()
...
<MagicMock name='mock.MockClass1().foo()' id='...'>
<MagicMock name='mock.MockClass2().bar()' id='...'>
>>> manager.mock_calls
[call.MockClass1(),
 call.MockClass1().foo(),
 call.MockClass2(),
 call.MockClass2().bar()]
```

If many calls have been made, but you're only interested in a particular sequence of them then an alternative is to use the `assert_has_calls()` method. This takes a list of calls (constructed with the `call` object). If that sequence of calls are in `mock_calls` then the assert succeeds.

```
>>> m = MagicMock()
>>> m().foo().bar().baz()
<MagicMock name='mock().foo().bar().baz()' id='...'>
>>> m.one().two().three()
<MagicMock name='mock.one().two().three()' id='...'>
>>> calls = call.one().two().three().call_list()
>>> m.assert_has_calls(calls)
```

Even though the chained call *m.one().two().three()* aren't the only calls that have been made to the mock, the assert still succeeds.

Sometimes a mock may have several calls made to it, and you are only interested in asserting about *some* of those calls. You may not even care about the order. In this case you can pass *any_order=True* to *assert_has_calls*:

```
>>> m = MagicMock()
>>> m(1), m.two(2, 3), m.seven(7), m.fifty('50')
(...)
>>> calls = [call.fifty('50'), call(1), call.seven(7)]
>>> m.assert_has_calls(calls, any_order=True)
```

2.2.17 More complex argument matching

Using the same basic concept as *ANY* 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 `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: call(<__main__.Foo object at 0x...>)
Actual call: call(<__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).

2.2.18 Less verbose configuration of mock objects

This recipe, for easier configuration of mock objects, is now part of *Mock*. See the `configure_mock()` method.

2.2.19 Matching any argument in assertions

This example is now built in to mock. See `ANY`.

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 name='mock.method1()' id='...'>
>>> my_mock.method2()
<Mock name='mock.method2()' id='...'>
>>> assertEqual(my_mock.mock_calls, [call.method1(), call.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
instance = mocker.mock()
klass = mocker.replace(SomeClass, spec=None)
klass('expected', 'args')
mocker.result(instance)

>>> # 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 name='mock.some_method()' id='...'>
>>> my_mock.some_method()
<Mock name='mock.some_method()' id='...'>
>>> 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 CHANGELOG

2.4.1 2012/02/13 Version 0.8.0

The only changes since 0.8rc2 are:

- Improved repr of `sentinel` objects
- `ANY` can be used for comparisons against `call` objects
- The return value of the `MagicMock` *__iter__* method can be set to any iterable and isn't required to be an iterator

Full List of changes since 0.7:

mock 0.8.0 is the last version that will support Python 2.4.

- Addition of `mock_calls` list for *all* calls (including magic methods and chained calls)
- `patch()` and `patch.object()` now create a `MagicMock` instead of a `Mock` by default
- The patchers (*patch*, *patch.object* and *patch.dict*), plus *Mock* and *MagicMock*, take arbitrary keyword arguments for configuration
- New mock method `configure_mock()` for setting attributes and return values / side effects on the mock and its attributes
- New mock assert methods `assert_any_call()` and `assert_has_calls()`
- Implemented *Autospeccing* (recursive, lazy speccing of mocks with mocked signatures for functions/methods), as the *autospec* argument to *patch*
- Added the `create_autospec()` function for manually creating 'auto-specced' mocks
- `patch.multiple()` for doing multiple patches in a single call, using keyword arguments
- Setting `side_effect` to an iterable will cause calls to the mock to return the next value from the iterable
- New *new_callable* argument to *patch* and *patch.object* allowing you to pass in a class or callable object (instead of *MagicMock*) that will be called to replace the object being patched
- Addition of `NonCallableMock` and `NonCallableMagicMock`, mocks without a *__call__* method
- Addition of `mock_add_spec()` method for adding (or changing) a spec on an existing mock
- Protocol methods on `MagicMock` are magic mocks, and are created lazily on first lookup. This means the result of calling a protocol method is a *MagicMock* instead of a *Mock* as it was previously
- Addition of `attach_mock()` method

- Added `ANY` for ignoring arguments in `assert_called_with()` calls
- Addition of `call` helper object
- Improved repr for mocks
- Improved repr for `Mock.call_args` and entries in `Mock.call_args_list`, `Mock.method_calls` and `Mock.mock_calls`
- Improved repr for `sentinel` objects
- *patch* lookup is done at use time not at decoration time
- In Python 2.6 or more recent, *dir* on a mock will report all the dynamically created attributes (or the full list of attributes if there is a spec) as well as all the mock methods and attributes.
- Module level `FILTER_DIR` added to control whether *dir(mock)* filters private attributes. *True* by default.
- *patch.TEST_PREFIX* for controlling how patchers recognise test methods when used to decorate a class
- Support for using Java exceptions as a `side_effect` on Jython
- *Mock* call lists (*call_args_list*, *method_calls* & *mock_calls*) are now custom list objects that allow membership tests for "sub lists" and have a nicer representation if you *str* or *print* them
- Mocks attached as attributes or return values to other mocks have calls recorded in *method_calls* and *mock_calls* of the parent (unless a name is already set on the child)
- Improved failure messages for *assert_called_with* and *assert_called_once_with*
- The return value of the `MagicMock` *__iter__* method can be set to any iterable and isn't required to be an iterator
- Added the Mock API (*assert_called_with* etc) to functions created by `mocksignature()`
- Tuples as well as lists can be used to specify allowed methods for *spec* & *spec_set* arguments
- Calling *stop* on an unstarted patcher fails with a more meaningful error message
- Renamed the internal classes *Sentinel* and *SentinelObject* to prevent abuse
- BUGFIX: an error creating a patch, with nested patch decorators, won't leave patches in place
- BUGFIX: *__truediv__* and *__rtruediv__* not available as magic methods on mocks in Python 3
- BUGFIX: *assert_called_with* / *assert_called_once_with* can be used with *self* as a keyword argument
- BUGFIX: when patching a class with an explicit spec / spec_set (not a boolean) it applies "spec inheritance" to the return value of the created mock (the "instance")
- BUGFIX: remove the *__unittest* marker causing traceback truncation

- Removal of deprecated *patch_object*
- Private attributes *_name*, *_methods*, '_children', *_wraps* and *_parent* (etc) renamed to reduce likelihood of clash with user attributes.
- Added license file to the distribution

2.4.2 2012/01/10 Version 0.8.0 release candidate 2

- Removed the *configure* keyword argument to *create_autospec* and allow arbitrary keyword arguments (for the *Mock* constructor) instead
- Fixed *ANY* equality with some types in *assert_called_with* calls
- Switched to a standard Sphinx theme (compatible with readthedocs.org)

2.4.3 2011/12/29 Version 0.8.0 release candidate 1

- *create_autospec* on the return value of a mocked class will use *__call__* for the signature rather than *__init__*
- Performance improvement instantiating *Mock* and *MagicMock*
- Mocks used as magic methods have the same type as their parent instead of being hardcoded to *MagicMock*

Special thanks to Julian Berman for his help with diagnosing and improving performance in this release.

2.4.4 2011/10/09 Version 0.8.0 beta 4

- *patch* lookup is done at use time not at decoration time
- When attaching a Mock to another Mock as a magic method, calls are recorded in mock_calls
- Addition of *attach_mock* method
- Renamed the internal classes *Sentinel* and *SentinelObject* to prevent abuse
- BUGFIX: various issues around circular references with mocks (setting a mock return value to be itself etc)

2.4.5 2011/08/15 Version 0.8.0 beta 3

- Mocks attached as attributes or return values to other mocks have calls recorded in *method_calls* and *mock_calls* of the parent (unless a name is already set on the child)

- Addition of *mock_add_spec* method for adding (or changing) a spec on an existing mock
- Improved repr for *Mock.call_args* and entries in *Mock.call_args_list*, *Mock.method_calls* and *Mock.mock_calls*
- Improved repr for mocks
- BUGFIX: minor fixes in the way *mock_calls* is worked out, especially for "intermediate" mocks in a call chain

2.4.6 2011/08/05 Version 0.8.0 beta 2

- Setting *side_effect* to an iterable will cause calls to the mock to return the next value from the iterable
- Added *assert_any_call* method
- Moved *assert_has_calls* from call lists onto mocks
- BUGFIX: *call_args* and all members of *call_args_list* are two tuples of *(args, kwargs)* again instead of three tuples of *(name, args, kwargs)*

2.4.7 2011/07/25 Version 0.8.0 beta 1

- *patch.TEST_PREFIX* for controlling how patchers recognise test methods when used to decorate a class
- *Mock* call lists (*call_args_list*, *method_calls* & *mock_calls*) are now custom list objects that allow membership tests for "sub lists" and have an *assert_has_calls* method for unordered call checks
- *callargs* changed to *always* be a three-tuple of *(name, args, kwargs)*
- Addition of *mock_calls* list for *all* calls (including magic methods and chained calls)
- Extension of *call* object to support chained calls and *callargs* for better comparisons with or without names. *call* object has a *call_list* method for chained calls
- Added the public *instance* argument to *create_autospec*
- Support for using Java exceptions as a *side_effect* on Jython
- Improved failure messages for *assert_called_with* and *assert_called_once_with*
- Tuples as well as lists can be used to specify allowed methods for *spec* & *spec_set* arguments
- BUGFIX: Fixed bug in *patch.multiple* for argument passing when creating mocks
- Added license file to the distribution

2.4.8 2011/07/16 Version 0.8.0 alpha 2

- *patch.multiple* for doing multiple patches in a single call, using keyword arguments
- New *new_callable* argument to *patch* and *patch.object* allowing you to pass in a class or callable object (instead of *MagicMock*) that will be called to replace the object being patched
- Addition of *NonCallableMock* and *NonCallableMagicMock*, mocks without a *__call__* method
- Mocks created by *patch* have a *MagicMock* as the *return_value* where a class is being patched
- *create_autospec* can create non-callable mocks for non-callable objects. *return_value* mocks of classes will be non-callable unless the class has a *__call__* method
- *autospec* creates a *MagicMock* without a spec for properties and slot descriptors, because we don't know the type of object they return
- Removed the "inherit" argument from *create_autospec*
- Calling *stop* on an unstarted patcher fails with a more meaningful error message
- BUGFIX: an error creating a patch, with nested patch decorators, won't leave patches in place
- BUGFIX: *__truediv__* and *__rtruediv__* not available as magic methods on mocks in Python 3
- BUGFIX: *assert_called_with* / *assert_called_once_with* can be used with *self* as a keyword argument
- BUGFIX: autospec for functions / methods with an argument named self that isn't the first argument no longer broken
- BUGFIX: when patching a class with an explicit spec / spec_set (not a boolean) it applies "spec inheritance" to the return value of the created mock (the "instance")
- BUGFIX: remove the *__unittest* marker causing traceback truncation

2.4.9 2011/06/14 Version 0.8.0 alpha 1

mock 0.8.0 is the last version that will support Python 2.4.

- The patchers (*patch*, *patch.object* and *patch.dict*), plus *Mock* and *MagicMock*, take arbitrary keyword arguments for configuration
- New mock method *configure_mock* for setting attributes and return values / side effects on the mock and its attributes
- In Python 2.6 or more recent, *dir* on a mock will report all the dynamically created attributes (or the full list of attributes if there is a spec) as well as all the mock methods and attributes.

- Module level *FILTER_DIR* added to control whether *dir(mock)* filters private attributes. *True* by default. Note that *vars(Mock())* can still be used to get all instance attributes and *dir(type(Mock())* will still return all the other attributes (irrespective of *FILTER_DIR*)
- *patch* and *patch.object* now create a *MagicMock* instead of a *Mock* by default
- Added *ANY* for ignoring arguments in *assert_called_with* calls
- Addition of *call* helper object
- Protocol methods on *MagicMock* are magic mocks, and are created lazily on first lookup. This means the result of calling a protocol method is a MagicMock instead of a Mock as it was previously
- Added the Mock API (*assert_called_with* etc) to functions created by *mocksignature*
- Private attributes *_name*, *_methods*, '_children', *_wraps* and *_parent* (etc) renamed to reduce likelihood of clash with user attributes.
- Implemented auto-speccing (recursive, lazy speccing of mocks with mocked signatures for functions/methods)

  Limitations:

  - Doesn't mock magic methods or attributes (it creates MagicMocks, so the magic methods are *there*, they just don't have the signature mocked nor are attributes followed)
  - Doesn't mock function / method attributes
  - Uses object traversal on the objects being mocked to determine types - so properties etc may be triggered
  - The return value of mocked classes (the 'instance') has the same call signature as the class __init__ (as they share the same spec)

  You create auto-specced mocks by passing *autospec=True* to *patch*.

  Note that attributes that are None are special cased and mocked without a spec (so any attribute / method can be used). This is because None is typically used as a default value for attributes that may be of some other type, and as we don't know what type that may be we allow all access.

  Note that the *autospec* option to *patch* obsoletes the *mocksignature* option.
- Added the *create_autospec* function for manually creating 'auto-specced' mocks
- Removal of deprecated *patch_object*

2.4.10 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__*

2.4.11 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

2.4.12 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

2.4.13 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

2.4.14 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*

2.4.15 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 `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

2.4.16 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

2.4.17 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

2.4.18 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

2.4.19 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).

2.4.20 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.

2.4.21 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.

2.4.22 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.

2.4.23 2007/11/21 Version 0.2.1

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

2.4.24 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.

2.4.25 2007/11/19 Version 0.1.0

Initial release.

2.5 TODO and Limitations

Contributions, bug reports and comments welcomed!

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

- mock issue tracker

wraps is not integrated with magic methods.

patch could 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 & PyPy (non-deterministic calling of destructors). Destructors aren't called in CPython 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 methods that *hide* these attributes when needed. In 0.8 many, but not all, of these attributes are renamed to gain a *_mock* prefix, making it less likely that they will clash. Any outstanding attributes that haven't been modified with the prefix should be changed.

If a patch is started using *patch.start* and then not stopped correctly 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 would be tricky to get right though.

When a *Mock* is created by *patch*, arbitrary keywords can be used to set attributes. If *patch* is created with a *spec*, and is replacing a class, then a *return_value* mock is created. The keyword arguments are not applied to the child mock, but could be.

When mocking a class with *patch*, passing in *spec=True* or *autospec=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? *create_autospec* does this, so an additional keyword argument to Mock is probably unnecessary.

The *mocksignature* argument to *patch* with a non *Mock* passed into *new_callable* will *probably* cause an error. Should it just be invalid?

Note that *NonCallableMock* and *NonCallableMagicMock* still have the unused (and unusable) attributes: *return_value*, *side_effect*, *call_count*, *call_args* and *call_args_list*. These could be removed or raise errors on getting / setting. They also have the *assert_called_with* and *assert_called_once_with* methods. Removing these would be pointless as fetching them would create a mock (attribute) that could be called without error.

Some outstanding technical debt. The way autospeccing mocks function signatures was copied and modified from *mocksignature*. This could all be refactored into one set of functions instead of two. The way we tell if patchers are started and if a patcher is being used for a *patch.multiple* call are both horrible. There are now a host of helper functions that should be rationalised. (Probably time to split mock into a package instead of a module.)

Passing arbitrary keyword arguments to *create_autospec*, or *patch* with *autospec*, when mocking a *function* works fine. However, the arbitrary attributes are set on the created mock - but *create_autospec* returns a real function (which doesn't have those attributes). However, what is the use case for using autospec to create functions with attributes that don't exist on the original?

mocksignature, plus the *call_args_list* and *method_calls* attributes of *Mock* could all be deprecated.

CHAPTER

THREE

INSTALLING

The current version is 0.8.0. Mock is stable and widely used. If you do find any 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` and `MagicMock` 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 MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

`side_effect` allows you to perform side effects, including raising an exception when a mock is called:

```
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
 ...
KeyError: 'foo'

>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]
...
>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
>>> mock(), mock(), mock()
(5, 4, 3)
```

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('module.ClassName2')
... @patch('module.ClassName1')
... def test(MockClass1, MockClass2):
...     module.ClassName1()
...     module.ClassName2()
...
...     assert MockClass1 is module.ClassName1
...     assert MockClass2 is 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 *module.ClassName1* 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', return_value=None) as mock_method:
...     thing = ProductionClass()
...     thing.method(1, 2, 3)
...
>>> mock_method.assert_called_once_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:

```
>>> 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(return_value='wheeeeee')
>>> str(mock)
'wheeeeee'
```

For ensuring that the mock objects in your tests have the same api as the objects they are replacing, you can use *auto-speccing*. Auto-speccing can be done through the *autospec* argument to patch, or the `create_autospec()` function. Auto-speccing creates mock objects that have the same attributes and methods as the objects they are replacing, and any functions and methods (including constructors) have the same call signature as the real object.

This ensures that your mocks will fail in the same way as your production code if they are used incorrectly:

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

create_autospec 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:

- mock-django: tools for mocking the Django ORM and models
- PyCon 2011 Video: Testing with mock
- Python: Injecting Mock Objects for Powerful Testing
- Mocking Django
- Mocking dates and other classes that can't be modified
- Mock recipes
- Mockity mock mock - some love for the mock module
- Coverage and Mock (with django)
- 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 0.7
- mock 0.6

Docs from the in-development version of *mock* can be found at mock.readthedocs.org.

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 it's 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

R

S

T

U

W