
Mock Documentation

Release 0.7.0 beta 2

Michael Foord

June 23, 2010

CONTENTS

1	API Documentation	3
1.1	The Mock Class	3
1.2	Patch Decorators	8
1.3	Sentinel	11
1.4	Mocking Magic Methods	12
1.5	Magic Mock	14
1.6	mocksignature	16
2	User Guide	19
2.1	Getting Started with Mock	19
2.2	Examples	22
2.3	TODO and Limitations	27
2.4	CHANGELOG	28
3	Installing	33
4	Quick Guide	35
5	References	39
6	Tests	41
7	Older Versions	43
	Index	45

Author [Michael Foord](#)

Co-maintainer [Konrad Delong](#)

Version Mock 0.7.0 beta 2

Date 2010/06/23

Homepage [Mock Homepage](#)

Download [Mock on PyPI](#)

Documentation [PDF Documentation](#)

License [BSD License](#)

Support [Testing in Python Email List](#)

Issue tracker [Google code project](#)

mock provides a core `Mock` class that is intended to reduce the need to create a host of trivial 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 specific attributes in the normal way.

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

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

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

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 them¹. Accessing the same attribute will always return the same mock. Mocks record how you use them, allowing you to make assertions about what your code has done to them.

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

class `Mock` (*spec=None, side_effect=None, return_value=DEFAULT, wraps=None*)

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

- `spec`: This can be either a list of strings or an existing object (a class or instance) that acts as the specification for the mock object. If you pass in an object then a list of strings is formed by calling `dir` on the object (excluding unsupported magic attributes and methods). Accessing any attribute not in this list will raise an `AttributeError`.
- `side_effect`: A function to be called whenever the `Mock` is called. See the `Mock.side_effect` attribute. Useful for raising exceptions or dynamically changing return values. The function is called with the same arguments as the mock, and unless it returns `DEFAULT`, the return value of this function is used as the return value. Alternatively `side_effect` can be an exception class or instance. In this case the exception will be raised when the mock is called.
- `return_value`: The value returned when the mock is called. By default this is a new `Mock` (created on first access). See the `Mock.return_value` attribute.

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.

- `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.

1.1.1 Methods

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

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

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

`Mock.reset_mock()`

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

```
>>> mock = Mock()
>>> mock('hello')
<mock.Mock object at 0x...>
>>> mock.called
True
>>> mock.reset_mock()
>>> mock.called
False
```

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

1.1.2 Calling

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

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

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

1.1.3 Attributes

`Mock.called`

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

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

`Mock.call_count`

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

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

`Mock.return_value`

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

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

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

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

`Mock.side_effect`

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

If you pass in a function it will be called with same arguments as the mock and unless the mock returns the DEFAULT singleton the mock will return whatever the function re-

turns. If the function returns default then the mock will return its normal value (from the `Mock.return_value`).

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

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

Using `side_effect` to return a sequence of values:

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

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

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

`Mock.call_args`

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

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

```

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

```

Mock.call_args_list

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

```

>>> mock = Mock()
>>> mock()
<mock.Mock object at 0x...>
>>> mock(3, 4, 5, key='fish', next='w00t!')
<mock.Mock object at 0x...>
>>> mock.call_args_list
[(), {}], ((3, 4, 5), {'key': 'fish', 'next': 'w00t!'})]
>>> mock.call_args_list == [(), ((3, 4, 5), {'key': 'fish', 'next': 'w00t!'})]
True

```

Mock.method_calls

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

```

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

```

The tuples in `method_calls` compare in favour even if positional and keyword arguments are skipped.

```

>>> mock = Mock()
>>> mock.method()
<mock.Mock object at 0x...>
>>> mock.method(1, 2)
<mock.Mock object at 0x...>
>>> mock.method(a="b")

```

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

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

1.2 Patch Decorators

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

1.2.1 patch

patch (*target*, *new=None*, *spec=None*, *create=False*)

`patch` decorates a function. Inside the body of the function, the `target` (specified in the form 'PackageName.ModuleName.ClassName') is patched with a new object. When the function exits the patch is undone.

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

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

```
@patch('Package.ModuleName.ClassName')
def test_something(self, MockClass):
    "test something"
```

The `spec` keyword argument is passed to the `Mock` if `patch` is creating one for you.

In addition you can pass `spec=True`, which causes `patch` to pass in the object being mocked as the `spec` object. If you are using `patch` to mock out a class, then the object you are interested in will probably be the return value of the `Mock` (the instance it returns when called). Because of this, if you use 'spec=True' and are patching a class (and having `patch` create a `Mock` for you) then the object being patched will be used as a `spec` for both the `Mock` and its return value.

Using the class as a `spec` object for the created `Mock` (and return value) means that the `Mock` will raise an exception if the code attempts to access any attributes that don't exist.

```
@patch('Package.ModuleName.ClassName', spec=True)
def test_something(self, MockClass):
    instance = ClassName()
    self.assertRaises(AttributeError, lambda: instance.fake_attribute)
```

Patch can be used with the with statement - if this is available in your version of Python. Here the patching applies to the indented block after the with statement. Note that the patched object can always appear after the “as” - even if an object to be patched was specified, though it can be omitted.

```
with patch('Package.ModuleName.ClassName', spec=True) as MockClass:
    instance = ClassName()
    self.assertRaises(AttributeError, lambda: instance.fake_attribute)
```

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 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 also 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('Patch.ModuleName.ClassName')
class SomeTest(unittest.TestCase):
    def test_something(self, MockClass):
        "test something"
```

1.2.2 patch.object

`patch.object` (*target, attribute, new=None, spec=None, create=False*)
patch.object patches named members on objects - usually class or module objects.

You can either call it 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, 'classmethod')
def test_something(self, mockMethod):
    SomeClass.classmethod(3)

    mockMethod.assert_called_with(3)
```

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

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

1.2.3 patch_object

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

1.2.4 patch.dict

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

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

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

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

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

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

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

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

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

1.2.5 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('module.ClassName1')
@patch('module.ClassName2')
def testMethod(self, MockClass2, MockClass1):
    ClassName1()
    ClassName2()
    self.assertEqual(MockClass1.called, "ClassName1 not patched")
    self.assertEqual(MockClass2.called, "ClassName2 not patched")
```

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.ModuleName.ClassName'),
            patch('Package.ModuleName.ClassName2', TestUtils.MockClass2)) as (MockC
instance = ClassName(ClassName2())
self.assertEqual(instance.f(), "expected")
```

1.2.6 Patching Descriptors

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:

```
@patch('module.ClassName.static')
def testMethod(self, mockStatic):
    ClassName.static('foo')
    mockStatic.assert_called_with('foo')
```

1.3 Sentinel

sentinel

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

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

DEFAULT

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

1.3.1 Sentinel Example

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

In this example we monkey patch method to return `sentinel.return_value`. We want to test that this is the value returned when we call `something`:

```
>>> real = ProductionClass()
>>> real.method = Mock()
>>> real.method.return_value = sentinel.return_value
>>> returned = real.something()
>>> self.assertEqual(returned, sentinel.return_value, "something returned the wrong value")

>>> sentinel.return_value
<SentinelObject "return_value">
```

1.4 Mocking Magic Methods

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

Because magic methods are looked up differently from normal methods², 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 argument³.

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

>>> from mock import Mock
>>> mock = Mock()
>>> mock.__str__ = Mock()
```

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

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

```

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

>>> from mock import Mock
>>> 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:

```

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

```

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

The full list of supported magic methods is:

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

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

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

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

- `__bool__` and `__next__`

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

- `__prepare__`, `__instancecheck__`, `__subclasscheck__`, `__del__`

1.5 Magic Mock

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

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

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

```
>>> from mock import MagicMock
>>> 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__`: 0
- `__contains__`: False
- `__len__`: 0
- `__iter__`: `iter([])`
- `__exit__`: False
- `__complex__`: 0j
- `__float__`: 0.0
- `__bool__`: True
- `__nonzero__`: True
- `__oct__`: '0'

- `__hex__`: '0x0'
- `__long__`: `long(0)`
- `__index__`: 0
- `__hash__`: default hash for the mock
- `__repr__`: default repr for the mock
- `__str__`: default str for the mock
- `__unicode__`: default unicode for the mock

For example:

```
>>> from mock import MagicMock
>>> mock = MagicMock()
>>> int(mock)
0
>>> len(mock)
0
>>> hex(mock)
'0x0'
>>> list(mock)
[]
>>> object() in mock
False
```

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__`
-

1.6 mocksignature

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

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

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

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

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

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

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

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

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

1.6.1 Example use

Basic use

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

```
'some value'  
>>> function.mock.assert_called_with(1, 2, 'foo')
```

Keyword arguments

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

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

Mocking methods and self

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

```
>>> from mock import mocksignature  
>>> class SomeClass(object):  
...     def method(self, a, b, c=None):  
...         pass  
...  
>>> SomeClass.method = mocksignature(SomeClass.method)  
>>> SomeClass.method.mock.return_value = None  
>>> instance = SomeClass()  
>>> instance.method()  
Traceback (most recent call last):  
...  
TypeError: <lambda>() takes at least 4 arguments (1 given)  
>>> instance.method(1, 2, 3)  
>>> instance.method.mock.assert_called_with(instance, 1, 2, 3)
```

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

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

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

1.6.2 mocksignature argument to patch

If you are patching a function then you can use the `mocksignature` keyword argument to `patch()` or `patch.object()`.

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

USER GUIDE

2.1 Getting Started with Mock

2.1.1 Using Mock

Mock objects can be used for:

- Patching methods
- Recording method calls on objects

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

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

```
>>> real.method.assert_called_with(3, 4, 5, key='value')
>>> real.method.called
True
>>> real.method.call_args
((3, 4, 5), {'key': 'value'})
>>>
```

Mocks also record calls made to attributes, their ‘child’ attributes:

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

You can also create Mock objects that behave like the class they are intended to mock. This is done with the `spec` keyword argument, which either takes a list of strings describing the attributes that the mock should have - or you can pass in the object (class or instance) that you are mocking out. Attempting to access an attribute on the mock that isn't in the spec will raise an `AttributeError`.

```
>>> mock = Mock(spec=['something'])
>>> mock.something()
<mock.Mock object at 0x...>
>>> mock.something_else()
Traceback (most recent call last):
...
AttributeError: object has no attribute 'something_else'
```

There are various ways of configuring the mock, including setting return values on the mock and its methods. A useful attribute is `side_effect`. If you set this to an exception class or instance then the exception will be raised when the mock is called. If you set it to a callable then it will be called whenever the mock is called. This allows you to do things like return members of a sequence from repeated calls:

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

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

2.1.2 Sentinel

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

```
>>> from mock import sentinel
>>> real = ProductionClass()
>>> real.method = Mock()
>>>
>>> real.method.return_value = sentinel.return_value
```

```
>>> real.method()
<SentinelObject "return_value">
```

2.1.3 Patch Decorators

There are also decorators for doing module and class level patching. As modules and classes are effectively globals any patching has to be undone (or it persists into other tests). These decorators do the unpatching for you, making it easier to test with module and class level patching.

The two decorators are ‘patch’ and ‘patch.object’. ‘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.

```
original = SomeClass.attribute
@patch.object(SomeClass, 'attribute', sentinel.Attribute)
def test():
    self.assertEqual(SomeClass.attribute, sentinel.Attribute, "class attribute not
test()
```

```
self.assertEqual(SomeClass.attribute, original, "attribute not restored")
```

```
@patch('Package.Module.attribute', sentinel.Attribute)
def test():
    "do something"
test()
```

If you don’t want to call the decorated test function yourself, you can add `apply` as a decorator on top:

```
@apply
@patch('Package.Module.attribute', sentinel.Attribute)
def test():
    "do something"
```

(Note that this leaves `test == None`)

A nice pattern is to actually decorate test methods themselves:

```
@patch('Package.Module.attribute', sentinel.Attribute)
def testMethod(self):
    "do something"
```

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:

```
@patch('Package.Module.Class')
def testMethod(self, mMockClass):
    "do something"
```

2.2 Examples

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

2.2.1 Mock Examples

Mock Patching Methods

Mock is callable. If it is called then it sets a called attribute to True.

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

```
def test_method_calls_something(self):
    real = ProductionClass()
    real.something = Mock()

    real.method()

    self.assertTrue(real.something.called, "method didn't call something")
```

If you want to catch the arguments then there is other information exposed:

```
def test_method_calls_something(self):
    real = ProductionClass()
    real.something = Mock()

    real.method()

    self.assertEqual(real.something.call_count, 1, "something called incorrect number of times")

    args = ()
    kwargs = {}
    self.assertEqual(real.something.call_args, (args, kwargs), "something called with incorrect arguments")
    self.assertEqual(real.something.call_args_list, [(args, kwargs)], "something called with incorrect arguments")
```

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

Mock for Method Calls on an Object

```
def test_closer_closes_something(self):
    real = ProductionClass()
    mock = Mock()

    real.closer(mock)

    self.assertTrue(mock.close.called, "closer didn't close something")
```

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 called will be False.

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

Limiting Available Methods

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

```
def test_closer_closes_something(self):
    real = ProductionClass()
    mock = Mock(spec=['close'])

    real.closer(mock)

    self.assertTrue(mock.close.called, "closer didn't close something")
```

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

Tracking all Method Calls

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

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

```
>>> mock = Mock()
>>> mock.method()
<mock.Mock object at 0x...>
```

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

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

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
```

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

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

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

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

Creating a Mock from an Existing Object

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

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

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

2.2.2 Patch Decorator Examples

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.

The `patch` and `patch.object` decorators provide a convenient way of doing this.

`patch.object` patches attributes on objects within the scope of a function they decorate:

```
>>> mock = Mock()

>>> @patch.object(SomeClass, 'class_method', mock)
... def test():
...     SomeClass.class_method()
...
>>> test()

>>> self.assertTrue(mock.called, "class_method not called")
```

The decorator is applied to a function (called `test` above). The patching only applies inside the body of the function. You have to call the function explicitly, this can be useful as the test function can take arguments and be used to implement several tests, it can also return values.

They can be stacked to perform multiple simultaneous patches:

```
>>> mock1 = Mock()
>>> mock2 = Mock()

>>> @patch.object(SomeClass, 'class_method', mock1)
... @patch.object(SomeClass, 'static_method', mock2)
... def test():
...     SomeClass.class_method()
...     SomeClass.static_method()
...
>>> test()

>>> self.assertTrue(mock1.called, "class_method not called")
>>> self.assertTrue(mock2.called, "static_method not called")
```

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

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

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

If you don't want to call the decorated test function yourself, you can add `apply` as a decorator on top:

```
@apply
@patch('Package.Module.attribute', sentinel.Attribute)
def test():
    "do something"
```

A nice pattern is to actually decorate test methods themselves:

```
@patch.object(SomeClass, 'attribute', sentinel.Attribute)
def testMethod(self):
    self.assertEqual(SomeClass.attribute, sentinel.Attribute, "SomeClass not patched")
```

If you omit the second argument to `patch` (or the third argument to `patch.object`) then the attribute will be patched with a mock for you. The mock will be passed in as extra argument(s) to the function / method under test:

```
@patch.object(SomeClass, 'staticmethod')
def testMethod(self, mockMethod):
```

```
SomeClass.staticmethod()  
self.assertTrue(mockMethod.called, "SomeClass not patched with a mock")
```

You can stack up multiple patch decorators using this pattern:

```
@patch('module.ClassName1')  
@patch('module.ClassName2')  
def test_method(self, MockClass1, MockClass2):  
    module.ClassName1()  
    module.ClassName2()  
    self.assertTrue(MockClass1.called, "ClassName1 not patched")  
    self.assertTrue(MockClass2.called, "ClassName2 not patched")
```

2.3 TODO and Limitations

Contributions, bug reports and comments welcomed!

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

- [mock module issue tracker](#)

When mocking a class with `patch`, passing in `spec=True`, the mock class has an instance created from the same spec. Should this be the default behaviour for mocks anyway (mock return values inheriting the spec from their parent), or should it be controlled by an additional keyword argument (`inherit`) to the `Mock` constructor?

More complete docstrings.

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

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

Could take a `patch` keyword argument and auto-do the patching in the constructor and unpatch in the destructor. This would be useful in itself, but violates TOOWTDI and would be unsafe for IronPython (non-deterministic calling of destructors).

`Mock` has several attributes. This makes it unsuitable for mocking objects that use these attribute names. A way round this would be to provide `start` and `stop` (or similar) methods that *hide* these attributes when needed.

2.4 CHANGELOG

2.4.1 2010/06/XX 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

2.4.2 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` return now 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

2.4.3 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

2.4.4 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).

2.4.5 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.

2.4.6 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.

2.4.7 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.

2.4.8 2007/11/21 Version 0.2.1

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

2.4.9 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.

2.4.10 2007/11/19 Version 0.1.0

Initial release.

INSTALLING

The current version is **0.7.0 beta 2**, dated 2010/06/23. Mock is still experimental; the API may change (although we are moving towards a 1.0 release when the API will stabilise). If you find bugs or have suggestions for improvements / extensions then please contact us.

- [mock on PyPI](#)
- [mock documentation as PDF](#)
- [Google Code Home & Subversion Repository](#)

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

```
svn checkout http://mock.googlecode.com/svn/trunk/  
mock-read-only
```

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

```
easy_install mock  
pip install mock
```

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

```
python setup.py install
```


QUICK GUIDE

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
>>> from mock import Mock
>>> mock = Mock()
>>> mock.__str__ = Mock()
```

```
>>> mock.__str__.return_value = 'wheeeeeee'
>>> str(mock)
'wheeeeeee'
```

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

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


REFERENCES

Articles and blog entries on testing with Mock:

- [Mock recipes](#)
- [Mockity mock mock - some love for the mock module](#)
- [Python Unit Testing with 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](#)

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
```

On Python 2.7 and 3.2 you can use `unittest` module from the standard library.

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

On Python 2.4 you will see one failure - `testwith.py` will fail to import as it uses the `with` statement which is invalid syntax under Python 2.4.

On Python 3 the tests for unicode are skipped as they are not relevant.

OLDER VERSIONS

Documentation for older versions of mock:

- [mock 0.6.0](#)

INDEX

Symbols

`__call__`, 4
`__init__`, 3

A

`assert_called_with()` (Mock method), 4

C

`call_args` (Mock attribute), 6
`call_args_list` (Mock attribute), 7
`call_count` (Mock attribute), 5
`called` (Mock attribute), 5

D

`DEFAULT` (in module `mock`), 11

G

Getting Started, 19

M

`MagicMock` (class in `mock`), 14
`method_calls` (Mock attribute), 7
`Mock` (class in `mock`), 3
`mock` (module), 1
`mocksignature()` (in module `mock`), 16

P

`patch()` (in module `mock`), 8
`patch.dict()` (in module `mock`), 10
`patch.object()` (in module `mock`), 9

R

`reset_mock()` (Mock method), 4
`return_value` (Mock attribute), 5

S

`sentinel` (in module `mock`), 11
`side_effect` (Mock attribute), 5