:mod:`unittest.mock` --- mock object library

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 5)

Unknown directive type "module".

```
.. module:: unittest.mock
   :synopsis: Mock object library.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 8)

Unknown directive type "moduleauthor".

```
.. moduleauthor:: Michael Foord <michael@python.org>
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 9)

Unknown directive type "currentmodule".

```
.. currentmodule:: unittest.mock
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 11)

Unknown directive type "versionadded".

```
.. versionadded:: 3.3
```

Source code: :source:`Lib/unittest/mock.py`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 13); *backlink*

Unknown interpreted text role "source".

:mod:`unittest.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.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 17); *backlink*

Unknown interpreted text role "mod".

:mod:`unittest.mock` provides a core :class:`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.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 21); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 21); *backlink*

Unknown interpreted text role "class".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 27); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 27); *backlink*

Unknown interpreted text role "const".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 27); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 27); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 27); *backlink*

Unknown interpreted text role "func".

Mock is designed for use with :mod:`unittest` and is based on the 'action -> assertion' pattern instead of 'record -> replay' used by many mocking frameworks.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 33); *backlink*

Unknown interpreted text role "mod".

There is a backport of :mod:`unittest.mock` for earlier versions of Python, available as mock on PyPI.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 37); *backlink*

Unknown interpreted text role "mod".

Quick Guide

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 44)

Unknown directive type "testsetup".

```
.. testsetup::

    class ProductionClass:
        def method(self, a, b, c):
            pass

    class SomeClass:
        @staticmethod
        def static_method(args):
            return args

        @classmethod
        def class_method(cls, args):
            return args
```

:class:`Mock` and :class:`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:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 60); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 60); *backlink*

Unknown interpreted text role "class".

```
>>> from unittest.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')
```

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 72); *backlink*

Unknown interpreted text role "attr".

```
>>> 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 :exc:`AttributeError`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 92); *backlink*

Unknown interpreted text role "exc".

The :func:`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:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 97); *backlink*

Unknown interpreted text role "func".

```
>>> from unittest.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 :func:`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 :ref:`where to patch <where-to-patch>`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 121); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 121); *backlink*

Unknown interpreted text role "ref".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 125); *backlink*

Unknown interpreted text role "func".

```
>>> 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 :func:`patch.dict` for setting values in a dictionary just during a scope and restoring the dictionary to its original state when the test ends:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 135); *backlink*

Unknown interpreted text role "func".

```
>>> 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 :ref:`magic methods <magic-methods>`. The easiest way of using magic methods is with the :class:`MagicMock` class. It allows you to do things like:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 146); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 146); *backlink*

Unknown interpreted text role "class".

```
>>> 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 :class:`MagicMock` class is just a Mock variant that has all of the magic methods pre-created for you (well, all the useful ones anyway).

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 156); *backlink*

Unknown interpreted text role "class".

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 :ref:`auto-speccing <auto-speccing>`. Auto-speccing can be done through the *autospec* argument to patch, or the :func:`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.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst`, line 169); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst`, line 169); *backlink*

Unknown interpreted text role "func".

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

```
>>> from unittest.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)
```

:func:`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.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst`, line 193); *backlink*

Unknown interpreted text role "func".

The Mock Class

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst`, line 202)

Unknown directive type "testsetup".

```
.. testsetup::

    import asyncio
    import inspect
    import unittest
    from unittest.mock import sentinel, DEFAULT, ANY
    from unittest.mock import patch, call, Mock, MagicMock, PropertyMock, AsyncMock
    from unittest.mock import mock_open
```

: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.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst`, line 211); *backlink*

Unknown interpreted text role "class".

:class:`MagicMock` is a subclass of :class:`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: :class:`NonCallableMock` and :class:`NonCallableMagicMock`

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst`, line 217); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst`, line 217); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst`, line 217); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst`, line 217); *backlink*

Unknown interpreted text role "class".

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

new_callable argument to :func:`patch`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 222); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 222); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 222); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 222); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 222); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 222); *backlink*

Unknown interpreted text role "func".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 230); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 230); *backlink*

Unknown interpreted text role "class".

- *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 :exc:`AttributeError`.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 233); *backlink*

  Unknown interpreted text role "exc".

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

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 239); *backlink*

  Unknown interpreted text role "attr".

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 239); *backlink*

  Unknown interpreted text role "func".

- *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 :exc:`AttributeError`.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 243); *backlink*

  Unknown interpreted text role "exc".

- *side_effect*: A function to be called whenever the Mock is called. See the :attr:`~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 :data:`DEFAULT`, the return value of this function is used as the return value.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 247); *backlink*

  Unknown interpreted text role "attr".

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 247); *backlink*

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 0)

Unknown interpreted text role "data".

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 :attr:`return_value` attribute.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 261); *backlink*

  Unknown interpreted text role "attr".

- *unsafe*: By default, accessing any attribute whose name starts with *assert*, *assret*, *asert*, *aseert* or *assrt* will raise an :exc:`AttributeError`. Passing `unsafe=True` will allow access to these attributes.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 265); *backlink*

  Unknown interpreted text role "exc".

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 270)

  Unknown directive type "versionadded".

  ```
  .. versionadded:: 3.5
  ```

- *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). 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 :exc:`AttributeError`).

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 272); *backlink*

  Unknown interpreted text role "exc".

  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 :meth:`configure_mock` method for details.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 286); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 290)

Unknown directive type "method".

```
.. method:: assert_called()

    Assert that the mock was called at least once.

        >>> mock = Mock()
        >>> mock.method()
        <Mock name='mock.method()' id='...'>
        >>> mock.method.assert_called()

    .. versionadded:: 3.6
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 301)

Unknown directive type "method".

```
.. method:: assert_called_once()

    Assert that the mock was called exactly once.

        >>> mock = Mock()
        >>> mock.method()
        <Mock name='mock.method()' id='...'>
        >>> mock.method.assert_called_once()
        >>> mock.method()
        <Mock name='mock.method()' id='...'>
        >>> mock.method.assert_called_once()
        Traceback (most recent call last):
        ...
        AssertionError: Expected 'method' to have been called once. Called 2 times.

    .. versionadded:: 3.6
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 319)

Unknown directive type "method".

```
.. method:: assert_called_with(*args, **kwargs)

    This method is a convenient way of asserting that the last call has been
    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')
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 329)

Unknown directive type "method".

```
.. method:: assert_called_once_with(*args, **kwargs)

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

        >>> mock = Mock(return_value=None)
        >>> mock('foo', bar='baz')
        >>> mock.assert_called_once_with('foo', bar='baz')
        >>> mock('other', bar='values')
        >>> mock.assert_called_once_with('other', bar='values')
        Traceback (most recent call last):
          ...
        AssertionError: Expected 'mock' to be called once. Called 2 times.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 344)

Unknown directive type "method".

```
.. method:: 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
    :meth:`assert_called_with` and :meth:`assert_called_once_with` that
    only pass if the call is the most recent one, and in the case of
    :meth:`assert_called_once_with` it must also be the only call.

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 359)

Unknown directive type "method".

```
.. method:: assert_has_calls(calls, any_order=False)

    assert the mock has been called with the specified calls.
    The :attr:`mock_calls` list is checked for the calls.

    If *any_order* is false 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 :attr:`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)
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 381)

Unknown directive type "method".

```
.. method:: assert_not_called()

    Assert the mock was never called.

        >>> m = Mock()
        >>> m.hello.assert_not_called()
        >>> obj = m.hello()
        >>> m.hello.assert_not_called()
        Traceback (most recent call last):
          ...
        AssertionError: Expected 'hello' to not have been called. Called 1 times.

    .. versionadded:: 3.5
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 396)

Unknown directive type "method".

```
.. method:: reset_mock(*, return_value=False, side_effect=False)

    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

    .. versionchanged:: 3.6
       Added two keyword only argument to the reset_mock function.

    This can be useful where you want to make a series of assertions that
    reuse the same object. Note that :meth:`reset_mock` *doesn't* clear the
    return value, :attr:`side_effect` or any child attributes you have
    set using normal assignment by default. In case you want to reset
    *return_value* or :attr:`side_effect`, then pass the corresponding
    parameter as ``True``. Child mocks and the return value mock
    (if any) are reset as well.

    .. note:: *return_value*, and :attr:`side_effect` are keyword only
              argument.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 423)

Unknown directive type "method".

```
.. method:: 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.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 432)

Unknown directive type "method".

```
.. method:: 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
    :attr:`method_calls` and :attr:`mock_calls` attributes of this one.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 439)

Unknown directive type "method".

```
.. method:: 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

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 474)

Unknown directive type "method".

```
.. method:: __dir__()

    :class:`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 :data:`FILTER_DIR` for what this filtering does, and how to
    switch it off.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 484)

Unknown directive type "method".

```
.. method:: _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).
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 495)

Unknown directive type "attribute".

```
.. attribute:: 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
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 506)

Unknown directive type "attribute".

```
.. attribute:: 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
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 518)

Unknown directive type "attribute".

```
.. attribute:: 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()

    :attr:`return_value` can also be set in the constructor:

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 545)

Unknown directive type "attribute".

```
.. attribute:: side_effect

    This can either be a function to be called when the mock is called,
    an iterable 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 :data:`DEFAULT` singleton the
    call to the mock will then return whatever the function returns. If the
    function returns :data:`DEFAULT` then the mock will return its normal
    value (from the :attr:`return_value`).

    If you pass in an iterable, it is used to retrieve an iterator which
    must yield a value on every call.  This value can either be an exception
    instance to be raised, or a value to be returned from the call to the
```

```
mock (:data:`DEFAULT` handling is identical to the function case).

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 :attr:`side_effect` to return a sequence of values:

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

Using a callable:

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

:attr:`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 :attr:`side_effect` to ``None`` clears it:

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 610)

Unknown directive type "attribute".

```
.. attribute:: 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, which can also be accessed through
    the ``args`` property, is any ordered arguments the mock was
    called with (or an empty tuple) and the second member, which can
    also be accessed through the ``kwargs`` property, 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.call_args.args
        (3, 4)
        >>> mock.call_args.kwargs
        {}
        >>> mock(3, 4, 5, key='fish', next='w00t!')
        >>> mock.call_args
        call(3, 4, 5, key='fish', next='w00t!')
        >>> mock.call_args.args
        (3, 4, 5)
        >>> mock.call_args.kwargs
        {'key': 'fish', 'next': 'w00t!'}

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

    .. versionchanged:: 3.8
       Added ``args`` and ``kwargs`` properties.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 655)

Unknown directive type "attribute".

```
.. attribute:: 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
:data:`call` object can be used for conveniently constructing lists of
calls to compare with :attr:`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 :attr:`call_args_list` are :data:`call` objects. These can be
unpacked as tuples to get at the individual arguments. See
:ref:`calls as tuples <calls-as-tuples>`.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 678)

Unknown directive type "attribute".

```
.. attribute:: 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 :attr:`method_calls` are :data:`call` objects. These can be
    unpacked as tuples to get at the individual arguments. See
    :ref:`calls as tuples <calls-as-tuples>`.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 696)

Unknown directive type "attribute".

```
.. attribute:: mock_calls

    :attr:`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 :attr:`mock_calls` are :data:`call` objects. These can be
    unpacked as tuples to get at the individual arguments. See
    :ref:`calls as tuples <calls-as-tuples>`.

    .. note::

        The way :attr:`mock_calls` are recorded means that where nested
        calls are made, the parameters of ancestor calls are not recorded
        and so will always compare equal:

            >>> mock = MagicMock()
            >>> mock.top(a=3).bottom()
            <MagicMock name='mock.top().bottom()' id='...'>
            >>> mock.mock_calls
            [call.top(a=3), call.top().bottom()]
            >>> mock.mock_calls[-1] == call.top(a=-1).bottom()
            True
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 734)

Unknown directive type "attribute".

```
.. attribute:: __class__

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

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

    :attr:`__class__` is assignable to, this allows a mock to pass an
    :func:`isinstance` check without forcing you to use a spec:

        >>> mock = Mock()
        >>> mock.__class__ = dict
        >>> isinstance(mock, dict)
        True
```

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 755); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 755); *backlink*

Unknown interpreted text role "class".

Mock objects that use a class or an instance as a :attr:`spec` or :attr:`spec_set` are able to pass :func:`isinstance` tests:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 759); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 759); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 759); *backlink*

Unknown interpreted text role "func".

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

The :class:`Mock` classes have support for mocking magic methods. See :ref:`magic methods <magic-methods>` for the full details.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 769); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 769); *backlink*

Unknown interpreted text role "ref".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 772); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 772); *backlink*

Unknown interpreted text role "func".

```
>>> 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
```

A callable mock which was created with a *spec* (or a *spec_set*) will introspect the specification object's signature when matching calls to the mock. Therefore, it can match the actual call's arguments regardless of whether they were passed positionally or by name:

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, c=3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(1, 2, 3)
>>> mock.assert_called_with(a=1, b=2, c=3)
```

This applies to :meth:`~Mock.assert_called_with`, :meth:`~Mock.assert_called_once_with`, :meth:`~Mock.assert_has_calls` and :meth:`~Mock.assert_any_call`. When :ref:`auto-speccing`, it will also apply to method calls on the mock object.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 811); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 811); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 811); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 811); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 811); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 816)

Unknown directive type "versionchanged".

```
.. versionchanged:: 3.4
   Added signature introspection on specced and autospecced mock objects.
```

A mock intended to be used as a property, or other descriptor, on a class. :class:`PropertyMock` provides :meth:`__get__` and :meth:`__set__` methods so you can specify a return value when it is fetched.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 822); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 822); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 822); *backlink*

Unknown interpreted text role "meth".

Fetching a :class:`PropertyMock` instance from an object calls the mock, with no args. Setting it calls the mock with the value being set.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 826); *backlink*

Unknown interpreted text role "class".

```
>>> class Foo:
...     @property
...     def foo(self):
...         return 'something'
...     @foo.setter
...     def foo(self, value):
...         pass
...
>>> with patch('__main__.Foo.foo', new_callable=PropertyMock) as mock_foo:
...     mock_foo.return_value = 'mockity-mock'
...     this_foo = Foo()
...     print(this_foo.foo)
...     this_foo.foo = 6
...
mockity-mock
>>> mock_foo.mock_calls
[call(), call(6)]
```

Because of the way mock attributes are stored you can't directly attach a :class:`PropertyMock` to a mock object. Instead you can attach it to the mock type object:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 847); *backlink*

Unknown interpreted text role "class".

```
>>> m = MagicMock()
>>> p = PropertyMock(return_value=3)
>>> type(m).foo = p
>>> m.foo
3
>>> p.assert_called_once_with()
```

An asynchronous version of :class:`MagicMock`. The :class:`AsyncMock` object will behave so the object is recognized as an async function, and the result of a call is an awaitable.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 861); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 861); *backlink*

Unknown interpreted text role "class".

```
>>> mock = AsyncMock()
>>> asyncio.iscoroutinefunction(mock)
True
>>> inspect.isawaitable(mock())  # doctest: +SKIP
True
```

The result of `mock()` is an async function which will have the outcome of `side_effect` or `return_value` after it has been awaited:

- if `side_effect` is a function, the async function will return the result of that function,
- if `side_effect` is an exception, the async function will raise the exception,
- if `side_effect` is an iterable, the async function will return the next value of the iterable, however, if the sequence of result is exhausted, `StopAsyncIteration` is raised immediately,
- if `side_effect` is not defined, the async function will return the value defined by `return_value`, hence, by default, the async function returns a new :class:`AsyncMock` object.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 881); *backlink*

Unknown interpreted text role "class".

Setting the *spec* of a :class:`Mock` or :class:`MagicMock` to an async function will result in a coroutine object being returned after calling.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 886); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 886); *backlink*

Unknown interpreted text role "class".

```
>>> async def async_func(): pass
...
>>> mock = MagicMock(async_func)
>>> mock
<MagicMock spec='function' id='...'>
>>> mock()  # doctest: +SKIP
<coroutine object AsyncMockMixin._mock_call at ...>
```

Setting the *spec* of a :class:`Mock`, :class:`MagicMock`, or :class:`AsyncMock` to a class with asynchronous and synchronous functions will automatically detect the synchronous functions and set them as :class:`MagicMock` (if the parent mock is :class:`AsyncMock` or :class:`MagicMock`) or :class:`Mock` (if the parent mock is :class:`Mock`). All asynchronous functions will be :class:`AsyncMock`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 898); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 898); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 898); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 898); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 898); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 898); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst, line 898); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 898); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 898); *backlink*

Unknown interpreted text role "class".

```
>>> class ExampleClass:
...     def sync_foo():
...         pass
...     async def async_foo():
...         pass
...
>>> a_mock = AsyncMock(ExampleClass)
>>> a_mock.sync_foo
<MagicMock name='mock.sync_foo' id='...'>
>>> a_mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
>>> mock = Mock(ExampleClass)
>>> mock.sync_foo
<Mock name='mock.sync_foo' id='...'>
>>> mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 922)

Unknown directive type "versionadded".

```
.. versionadded:: 3.8
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 924)

Unknown directive type "method".

```
.. method:: assert_awaited()

    Assert that the mock was awaited at least once. Note that this is separate
    from the object having been called, the ``await`` keyword must be used:

        >>> mock = AsyncMock()
        >>> async def main(coroutine_mock):
        ...     await coroutine_mock
        ...
        >>> coroutine_mock = mock()
        >>> mock.called
        True
        >>> mock.assert_awaited()
        Traceback (most recent call last):
        ...
        AssertionError: Expected mock to have been awaited.
        >>> asyncio.run(main(coroutine_mock))
        >>> mock.assert_awaited()
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 943)

Unknown directive type "method".

```
.. method:: assert_awaited_once()

    Assert that the mock was awaited exactly once.

      >>> mock = AsyncMock()
      >>> async def main():
      ...     await mock()
      ...
      >>> asyncio.run(main())
      >>> mock.assert_awaited_once()
      >>> asyncio.run(main())
      >>> mock.method.assert_awaited_once()
      Traceback (most recent call last):
      ...
      AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 959)

Unknown directive type "method".

```
.. method:: assert_awaited_with(*args, **kwargs)

    Assert that the last await was with the specified arguments.

      >>> mock = AsyncMock()
      >>> async def main(*args, **kwargs):
      ...     await mock(*args, **kwargs)
      ...
      >>> asyncio.run(main('foo', bar='bar'))
      >>> mock.assert_awaited_with('foo', bar='bar')
      >>> mock.assert_awaited_with('other')
      Traceback (most recent call last):
      ...
      AssertionError: expected call not found.
      Expected: mock('other')
      Actual: mock('foo', bar='bar')
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-

main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 976)

Unknown directive type "method".

```
.. method:: assert_awaited_once_with(*args, **kwargs)

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

      >>> mock = AsyncMock()
      >>> async def main(*args, **kwargs):
      ...     await mock(*args, **kwargs)
      ...
      >>> asyncio.run(main('foo', bar='bar'))
      >>> mock.assert_awaited_once_with('foo', bar='bar')
      >>> asyncio.run(main('foo', bar='bar'))
      >>> mock.assert_awaited_once_with('foo', bar='bar')
      Traceback (most recent call last):
      ...
      AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 993)

Unknown directive type "method".

```
.. method:: assert_any_await(*args, **kwargs)

    Assert the mock has ever been awaited with the specified arguments.

      >>> mock = AsyncMock()
      >>> async def main(*args, **kwargs):
      ...     await mock(*args, **kwargs)
      ...
      >>> asyncio.run(main('foo', bar='bar'))
      >>> asyncio.run(main('hello'))
      >>> mock.assert_any_await('foo', bar='bar')
      >>> mock.assert_any_await('other')
      Traceback (most recent call last):
      ...
      AssertionError: mock('other') await not found
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1009)

Unknown directive type "method".

```
.. method:: assert_has_awaits(calls, any_order=False)

    Assert the mock has been awaited with the specified calls.
    The :attr:`await_args_list` list is checked for the awaits.

    If *any_order* is false then the awaits must be
    sequential. There can be extra calls before or after the
    specified awaits.

    If *any_order* is true then the awaits can be in any order, but
    they must all appear in :attr:`await_args_list`.

      >>> mock = AsyncMock()
      >>> async def main(*args, **kwargs):
      ...     await mock(*args, **kwargs)
      ...
      >>> calls = [call("foo"), call("bar")]
      >>> mock.assert_has_awaits(calls)
      Traceback (most recent call last):
      ...
      AssertionError: Awaits not found.
      Expected: [call('foo'), call('bar')]
      Actual: []
      >>> asyncio.run(main('foo'))
      >>> asyncio.run(main('bar'))
      >>> mock.assert_has_awaits(calls)
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1036)

Unknown directive type "method".

```
.. method:: assert_not_awaited()

  Assert that the mock was never awaited.

      >>> mock = AsyncMock()
      >>> mock.assert_not_awaited()
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1043)

Unknown directive type "method".

```
.. method:: reset_mock(*args, **kwargs)

  See :func:`Mock.reset_mock`. Also sets :attr:`await_count` to 0,
  :attr:`await_args` to None, and clears the :attr:`await_args_list`.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1048)

Unknown directive type "attribute".

```
.. attribute:: await_count

  An integer keeping track of how many times the mock object has been awaited.
```

```
>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.await_count
1
>>> asyncio.run(main())
>>> mock.await_count
2
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1063)

Unknown directive type "attribute".

```
.. attribute:: await_args

  This is either ``None`` (if the mock hasnâ€™t been awaited), or the arguments that
  the mock was last awaited with. Functions the same as :attr:`Mock.call_args`.

    >>> mock = AsyncMock()
    >>> async def main(*args):
    ...     await mock(*args)
    ...
    >>> mock.await_args
    >>> asyncio.run(main('foo'))
    >>> mock.await_args
    call('foo')
    >>> asyncio.run(main('bar'))
    >>> mock.await_args
    call('bar')
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1081)

Unknown directive type "attribute".

```
.. attribute:: await_args_list

  This is a list of all the awaits made to the mock object in sequence (so the
  length of the list is the number of times it has been awaited). Before any
  awaits have been made it is an empty list.

    >>> mock = AsyncMock()
    >>> async def main(*args):
    ...     await mock(*args)
    ...
    >>> mock.await_args_list
    []
    >>> asyncio.run(main('foo'))
    >>> mock.await_args_list
    [call('foo')]
    >>> asyncio.run(main('bar'))
    >>> mock.await_args_list
    [call('foo'), call('bar')]
```

Calling

Mock objects are callable. The call will return the value set as the :attr:`~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.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1104); *backlink*

Unknown interpreted text role "attr".

Calls made to the object will be recorded in the attributes like :attr:`~Mock.call_args` and :attr:`~Mock.call_args_list`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1110); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1110); *backlink*

Unknown interpreted text role "attr".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1113); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1113); *backlink*

Unknown interpreted text role "attr".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1117); *backlink*

Unknown interpreted text role "attr".

```
>>> 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 :attr:`side_effect` is a function then whatever that function returns is what calls to the mock return. The :attr:`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:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1135); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1135); *backlink*

Unknown interpreted text role "attr".

```
>>> 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 :attr:`mock.return_value` from inside :attr:`side_effect`, or return :data:`DEFAULT`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1151); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1151); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1151); *backlink*

Unknown interpreted text role "data".

```
>>> 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 :attr:`side_effect`, and return to the default behaviour, set the :attr:`side_effect` to `None`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1170); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1170); *backlink*

Unknown interpreted text role "attr".

```
>>> 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 :attr:`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 :exc:`StopIteration` is raised):

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1184); backlink

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1184); backlink

Unknown interpreted text role "exc".

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

If any members of the iterable are exceptions they will be raised instead of returned:

```
>>> iterable = (33, ValueError, 66)
>>> m = MagicMock(side_effect=iterable)
>>> m()
33
>>> m()
Traceback (most recent call last):
 ...
ValueError
>>> m()
66
```

Deleting Attributes

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

You may want a mock object to return `False` to a :func:`hasattr` call, or raise an :exc:`AttributeError` when an attribute is fetched. You can do this by providing an object as a :attr:`spec` for a mock, but that isn't always convenient.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1223); backlink

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1223); backlink

Unknown interpreted text role "exc".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1223); backlink

Unknown interpreted text role "attr".

You "block" attributes by deleting them. Once deleted, accessing an attribute will raise an :exc:`AttributeError`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1227); backlink

Unknown interpreted text role "exc".

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

Mock names and the name attribute

Since "name" is an argument to the :class:`Mock` constructor, if you want your mock object to have a "name" attribute you can't just pass it in at creation time. There are two alternatives. One option is to use :meth:`~Mock.configure_mock`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1246); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1246); backlink

Unknown interpreted text role "meth".

```
>>> mock = MagicMock()
>>> mock.configure_mock(name='my_name')
>>> mock.name
'my_name'
```

A simpler option is to simply set the "name" attribute after mock creation:

```
>>> mock = MagicMock()
>>> mock.name = "foo"
```

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 :attr:`~Mock.method_calls` and :attr:`~Mock.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:

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1265); *backlink*
>
> Unknown interpreted text role "attr".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1265); *backlink*
>
> Unknown interpreted text role "attr".

```
>>> 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 :func:`patch` are automatically given names. To attach mocks that have names to a parent you use the :meth:`~Mock.attach_mock` method:

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1294); *backlink*
>
> Unknown interpreted text role "func".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1294); *backlink*
>
> Unknown interpreted text role "meth".

```
>>> 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] The only exceptions are magic methods and attributes (those that have leading and trailing double underscores). Mock doesn't create these but instead raises an :exc:`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 :ref:`magic methods <magic-methods>`.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1312); *backlink*
>
> Unknown interpreted text role "exc".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1312); *backlink*
>
> Unknown interpreted text role "ref".

The patchers

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.

patch

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

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1336)
>
> Unknown directive type "function".
>
> ```
> .. function:: patch(target, new=DEFAULT, spec=None, create=False, spec_set=None, autospec=None, new_callable=None, **kwa
> ```

```
        :func:`patch` acts as a function decorator, class decorator or a context
        manager. Inside the body of the function or with statement, the *target*
        is patched with a *new* object. When the function/with statement exits
        the patch is undone.

        If *new* is omitted, then the target is replaced with an
        :class:`AsyncMock` if the patched object is an async function or
        a :class:`MagicMock` otherwise.
        If :func:`patch` is used as a decorator and *new* is
        omitted, the created mock is passed in as an extra argument to the
        decorated function. If :func:`patch` is used as a context manager the created
        mock is returned by the context manager.

        *target* should be a string in the form ``'package.module.ClassName'``. The
        *target* is imported and the specified object replaced with the *new*
        object, so the *target* must be importable from the environment you are
        calling :func:`patch` from. The target is imported when the decorated function
        is executed, not at decoration time.

        The *spec* and *spec_set* keyword arguments are passed to the :class:`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 :class:`AsyncMock`
        is used for async functions and :class:`MagicMock` for the rest.

        A more powerful form of *spec* is *autospec*. If you set ``autospec=True``
        then the mock will 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 :exc:`TypeError` if they are
        called with the wrong signature. For mocks
        replacing a class, their return value (the 'instance') will have the same
        spec as the class. See the :func:`create_autospec` function and
        :ref:`auto-speccing`.

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

        By default :func:`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 after the patched function has exited. 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!

        .. note::

            .. versionchanged:: 3.5
               If you are patching builtins in a module then you don't
               need to pass ``create=True``, it will be added by default.

        Patch can be used as a :class:`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. :func:`patch` finds
        tests by looking for method names that start with ``patch.TEST_PREFIX``.
        By default this is ``'test'``, which matches the way :mod:`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 :func:`patch` is creating a mock object for you.

        :func:`patch` takes arbitrary keyword arguments. These will be passed to
        :class:`AsyncMock` if the patched object is asynchronous, to
        :class:`MagicMock` otherwise or to *new_callable* if specified.

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

:func:`patch` as function decorator, creating the mock for you and passing it into the decorated function:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1413); *backlink*

Unknown interpreted text role "func".

```
>>> @patch('__main__.SomeClass')
... def function(normal_argument, mock_class):
...     print(mock_class is SomeClass)
...
>>> function(None)
True
```

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1423); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1423); *backlink*

Unknown interpreted text role "attr".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1427); *backlink*

Unknown interpreted text role "attr".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1431); *backlink*

Unknown interpreted text role "attr".

```
>>> class Class:
...     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 :func:`patch` is replacing a *class*, then the return value of the created mock will have the same spec.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1445); *backlink*

Unknown interpreted text role "func".

```
>>> 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 :class:`MagicMock` for the created mock. For example, if you wanted a :class:`NonCallableMock` to be used:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1455); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1455); *backlink*

Unknown interpreted text role "class".

```
>>> 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 an :class:`io.StringIO` instance:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1468); *backlink*

Unknown interpreted text role "class".

```
>>> from io 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 :func:`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:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1481); *backlink*

Unknown interpreted text role "func".

```
>>> 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 :attr:`~Mock.return_value` and :attr:`~Mock.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 :func:`patch` call using `**`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1493); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1493); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1493); backlink

Unknown interpreted text role "func".

```
>>> 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
```

By default, attempting to patch a function in a module (or a method or an attribute in a class) that does not exist will fail with :exc:`AttributeError`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1509); backlink

Unknown interpreted text role "exc".

```
>>> @patch('sys.non_existing_attribute', 42)
... def test():
...     assert sys.non_existing_attribute == 42
...
>>> test()
Traceback (most recent call last):
  ...
AttributeError: <module 'sys' (built-in)> does not have the attribute 'non_existing_attribute'
```

but adding `create=True` in the call to :func:`patch` will make the previous example work as expected:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1521); backlink

Unknown interpreted text role "func".

```
>>> @patch('sys.non_existing_attribute', 42, create=True)
... def test(mock_stdout):
...     assert sys.non_existing_attribute == 42
...
>>> test()
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1530)

Unknown directive type "versionchanged".

```
.. versionchanged:: 3.8

    :func:`patch` now returns an :class:`AsyncMock` if the target is an async function
```

patch.object

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1538)

Unknown directive type "function".

```
.. function:: patch.object(target, attribute, new=DEFAULT, spec=None, create=False, spec_set=None, autospec=None, new_ca

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

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

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

You can either call :func:`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.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1552); backlink

Unknown interpreted text role "func".

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 :func:`patch.object` have the same meaning as they do for :func:`patch`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1567); backlink

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1567); *backlink*

Unknown interpreted text role "func".

patch.dict

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1574)

Unknown directive type "function".

```
.. function:: 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.

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

    .. versionchanged:: 3.8

        :func:`patch.dict` now returns the patched dictionary when used as a context
        manager.
```

:func:`patch.dict` can be used as a context manager, decorator or class decorator:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1600); *backlink*

Unknown interpreted text role "func".

```
>>> foo = {}
>>> @patch.dict(foo, {'newkey': 'newvalue'})
... def test():
...     assert foo == {'newkey': 'newvalue'}
>>> test()
>>> assert foo == {}
```

When used as a class decorator :func:`patch.dict` honours `patch.TEST_PREFIX` (default to `'test'`) for choosing which methods to wrap:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1610); *backlink*

Unknown interpreted text role "func".

```
>>> import os
>>> import unittest
>>> from unittest.mock import patch
>>> @patch.dict('os.environ', {'newkey': 'newvalue'})
... class TestSample(unittest.TestCase):
...     def test_sample(self):
...         self.assertEqual(os.environ['newkey'], 'newvalue')
```

If you want to use a different prefix for your test, you can inform the patchers of the different prefix by setting `patch.TEST_PREFIX`. For more details about how to change the value of see :ref:`test-prefix`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1621); *backlink*

Unknown interpreted text role "ref".

:func:`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.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1625); *backlink*

Unknown interpreted text role "func".

```
>>> foo = {}
>>> with patch.dict(foo, {'newkey': 'newvalue'}) as patched_foo:
...     assert foo == {'newkey': 'newvalue'}
...     assert patched_foo == {'newkey': 'newvalue'}
...     # You can add, update or delete keys of foo (or patched_foo, it's the same dict)
...     patched_foo['spam'] = 'eggs'
...
>>> assert foo == {}
>>> assert patched_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 :func:`patch.dict` call to set values in the dictionary:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst`, line 1646); *backlink*

Unknown interpreted text role "func".

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

:func:`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 :meth:`__getitem__`, :meth:`__setitem__`, :meth:`__delitem__` and either :meth:`__iter__` or :meth:`__contains__`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst`, line 1656); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst`, line 1656); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst`, line 1656); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst`, line 1656); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst`, line 1656); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst`, line 1656); *backlink*

Unknown interpreted text role "meth".

```
>>> class Container:
...     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']
```

patch.multiple

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main) (Doc) (library)unittest.mock.rst`, line 1687)

Unknown directive type "function".

```
.. function:: patch.multiple(target, spec=None, create=False, spec_set=None, autospec=None, 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 :data:`DEFAULT` as the value if you want :func:`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 :func:`patch.multiple` is
    used as a context manager.

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

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

If you want :func:`patch.multiple` to create mocks for you, then you can use :data:`DEFAULT` as the value. If you use

:func:`patch.multiple` as a decorator then the created mocks are passed into the decorated function by keyword.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1709); backlink

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1709); backlink

Unknown interpreted text role "data".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1709); backlink

Unknown interpreted text role "func".

```
>>> 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()
```

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1723); backlink

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1723); backlink

Unknown interpreted text role "func".

```
>>> @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 :func:`patch.multiple` is used as a context manager, the value returned by the context manager is a dictionary where created mocks are keyed by name:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1735); backlink

Unknown interpreted text role "func".

```
>>> 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
...
```

patch methods: start and stop

All the patchers have :meth:`start` and :meth:`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.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1751); backlink

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1751); backlink

Unknown interpreted text role "meth".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1755); backlink

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1755); backlink

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1755); backlink

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1755); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1755); *backlink*

Unknown interpreted text role "meth".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1759); *backlink*

Unknown interpreted text role "func".

```
>>> 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 :class:`TestCase`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1773); *backlink*

Unknown interpreted text role "class".

```
>>> class MyTest(unittest.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. :meth:`unittest.TestCase.addCleanup` makes this easier:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1795); *backlink*

Unknown interpreted text role "meth".

```
>>> class MyTest(unittest.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
...
```

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

It is also possible to stop all patches which have been started by using :func:`patch.stopall`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1813); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1816)

Unknown directive type "function".

```
.. function:: patch.stopall

    Stop all active patches. Only stops patches started with ``start``.
```

patch builtins

You can patch any builtins within a module. The following example patches builtin :func:`ord`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1825); *backlink*

Unknown interpreted text role "func".

```
>>> @patch('__main__.ord')
... def test(mock_ord):
...     mock_ord.return_value = 101
...     print(ord('c'))
...
>>> test()
101
```

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 :class:`unittest.TestLoader` finds test methods by default.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1842); *backlink*
>
> Unknown interpreted text role "class".

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:
...     def foo_one(self):
...         print(value)
...     def foo_two(self):
...         print(value)
...
>>>
>>> Thing().foo_one()
not three
>>> Thing().foo_two()
not three
>>> value
3
```

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.

Where to patch

:func:`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.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1901); *backlink*
>
> Unknown interpreted text role "func".

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 :func:`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 :func:`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.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1919); *backlink*
>
> Unknown interpreted text role "func".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 1919); *backlink*
>
> Unknown interpreted text role "func".

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 in the module and so we have to patch `a.SomeClass` instead:

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

Patching Descriptors and Proxy Objects

Both patch and patch.object correctly patch and restore descriptors: class methods, static methods and properties. You should patch these on the *class* rather than an instance. They also work with *some* objects that proxy attribute access, like the django settings object.

MagicMock and magic method support

Mocking Magic Methods

:class:`Mock` supports mocking the Python protocol methods, also known as "magic methods". This allows mock objects to replace containers or other objects that implement Python protocols.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst`, **line 1958**); *backlink*
>
> Unknown interpreted text role "class".

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

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

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

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

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

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

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst`, **line 1990**); *backlink*
>
> Unknown interpreted text role "keyword".

```
>>> 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 :attr:`~Mock.method_calls`, but they are recorded in :attr:`~Mock.mock_calls`.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst`, **line 2002**); *backlink*
>
> Unknown interpreted text role "attr".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst`, **line 2002**); *backlink*
>
> Unknown interpreted text role "attr".

> **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 :exc:`AttributeError`.
>
> > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst`, **line 2007**); *backlink*
> >
> > Unknown interpreted text role "exc".

The full list of supported magic methods is:

- `__hash__`, `__sizeof__`, `__repr__` and `__str__`
- `__dir__`, `__format__` and `__subclasses__`
- `__round__`, `__floor__`, `__trunc__` and `__ceil__`
- Comparisons: `__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__` and `__ne__`
- Container methods: `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`, `__len__`, `__iter__`, `__reversed__` and `__missing__`
- Context manager: `__enter__`, `__exit__`, `__aenter__` and `__aexit__`
- Unary numeric methods: `__neg__`, `__pos__` and `__invert__`

- The numeric methods (including right hand and in-place variants): `__add__`, `__sub__`, `__mul__`, `__matmul__`, `__truediv__`, `__floordiv__`, `__mod__`, `__divmod__`, `__lshift__`, `__rshift__`, `__and__`, `__xor__`, `__or__`, and `__pow__`
- Numeric conversion methods: `__complex__`, `__int__`, `__float__` and `__index__`
- Descriptor methods: `__get__`, `__set__` and `__delete__`
- Pickling: `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` and `__setstate__`
- File system path representation: `__fspath__`
- Asynchronous iteration methods: `__aiter__` and `__anext__`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2034)

Unknown directive type "versionchanged".

```
.. versionchanged:: 3.8
   Added support for :func:`os.PathLike.__fspath__`.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2037)

Unknown directive type "versionchanged".

```
.. versionchanged:: 3.8
   Added support for ``__aenter__``, ``__aexit__``, ``__aiter__`` and ``__anext__``.
```

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__`

Magic Mock

There are two `MagicMock` variants: :class:`MagicMock` and :class:`NonCallableMagicMock`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2052); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2052); *backlink*

Unknown interpreted text role "class".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2057); *backlink*

Unknown interpreted text role "class".

The constructor parameters have the same meaning as for :class:`Mock`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2061); *backlink*

Unknown interpreted text role "class".

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

A non-callable version of :class:`MagicMock`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2069); *backlink*

Unknown interpreted text role "class".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2071); *backlink*

Unknown interpreted text role "class".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2075); *backlink*

Unknown interpreted text role "class".

```
>>> 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:

- `__lt__`: NotImplemented
- `__gt__`: NotImplemented
- `__le__`: NotImplemented
- `__ge__`: NotImplemented
- `__int__`: 1
- `__contains__`: False
- `__len__`: 0
- `__iter__`: iter([])
- `__exit__`: False
- `__aexit__`: False
- `__complex__`: 1j
- `__float__`: 1.0
- `__bool__`: True
- `__index__`: 1
- `__hash__`: default hash for the mock
- `__str__`: default str for the mock
- `__sizeof__`: default sizeof for the mock

For example:

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

The two equality methods, :meth:`__eq__` and :meth:`__ne__`, are special. They do the default equality comparison on identity, using the :attr:`~Mock.side_effect` attribute, unless you change their return value to return something else:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst`, line 2123); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst`, line 2123); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst`, line 2123); *backlink*

Unknown interpreted text role "attr".

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

The return value of :meth:`MagicMock.__iter__` can be any iterable object and isn't required to be an iterator:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst`, line 2137); *backlink*

Unknown interpreted text role "meth".

```
>>> 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:

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

[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.

Helpers

sentinel

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2186)

Unknown directive type "data".

```
.. data:: 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.

   .. versionchanged:: 3.7
      The ``sentinel`` attributes now preserve their identity when they are
      :mod:`copied <copy>` or :mod:`pickled <pickle>`.
```

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. :data:`sentinel` provides a convenient way of creating and testing the identity of objects like this.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2199); *backlink*

Unknown interpreted text role "data".

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
>>> result
sentinel.some_object
```

DEFAULT

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2219)

Unknown directive type "data".

```
.. data:: DEFAULT

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

call

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2229)

Unknown directive type "function".

```
.. function:: call(*args, **kwargs)

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2242)

Unknown directive type "method".

```
.. method:: call.call_list()

    For a call object that represents multiple calls, :meth:`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 :attr:`~Mock.mock_calls` on a mock. Manually constructing the sequence of calls can be tedious.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2248); *backlink*

Unknown interpreted text role "attr".

:meth:`~call.call_list` can construct the sequence of calls from the same chained call:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2253); *backlink*

Unknown interpreted text role "meth".

```
>>> 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 :attr:`Mock.call_args`, :attr:`Mock.call_args_list` and :attr:`Mock.mock_calls` attributes can be introspected to get at the individual arguments they contain.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2270); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2270); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2270); *backlink*

Unknown interpreted text role "attr".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2277); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2277); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2277); *backlink*

Unknown interpreted text role "attr".

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
>>> kall.args
(1, 2, 3)
>>> kall.kwargs
{'arg': 'one', 'arg2': 'two'}
>>> kall.args is kall[0]
True
>>> kall.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
{'arg': 'two', 'arg2': 'three'}
>>> name is m.mock_calls[0][0]
True
```

create_autospec

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2317)

Unknown directive type "function".

```
.. function:: 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 to
    ensure 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 :exc:`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.

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

See :ref`auto-speccing` for examples of how to use auto-speccing with :func:`create_autospec` and the *autospec* argument to :func:`patch`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2337); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2337); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2337); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2341)

Unknown directive type "versionchanged".

```
.. versionchanged:: 3.8

    :func:`create_autospec` now returns an :class:`AsyncMock` if the target is
    an async function.
```

ANY

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2350)

Unknown directive type "data".

```
.. data:: 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 :attr:`~Mock.call_args` and make more complex assertions on them.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2352); *backlink*

Unknown interpreted text role "attr".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2357); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2357); *backlink*

Unknown interpreted text role "meth".

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

:data:`ANY` can also be used in comparisons with call lists like :attr:`~Mock.mock_calls`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2366); *backlink*

Unknown interpreted text role "data".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2366); *backlink*

Unknown interpreted text role "attr".

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

FILTER_DIR

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2381)

Unknown directive type "data".

```
.. data:: FILTER_DIR
```

:data:`FILTER_DIR` is a module level variable that controls the way mock objects respond to :func:`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`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2383); *backlink*

Unknown interpreted text role "data".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2383); *backlink*

Unknown interpreted text role "func".

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:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2395)

Unknown directive type "doctest".

```
.. doctest::
    :options: +ELLIPSIS,+NORMALIZE_WHITESPACE

    >>> dir(Mock())
    ['assert_any_call',
     'assert_called',
     'assert_called_once',
     'assert_called_once_with',
     'assert_called_with',
     'assert_has_calls',
     'assert_not_called',
     'attach_mock',
     ...
    >>> from urllib import request
    >>> dir(Mock(spec=request))
    ['AbstractBasicAuthHandler',
     'AbstractDigestAuthHandler',
     'AbstractHTTPHandler',
     'BaseHandler',
     ...
```

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2416); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2416); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2416); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2416); *backlink*

Unknown interpreted text role "data".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2422)

Unknown directive type "doctest".

```
.. doctest::
    :options: +ELLIPSIS,+NORMALIZE_WHITESPACE

    >>> from unittest 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 :data:`mock.FILTER_DIR`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2437); *backlink*

Unknown interpreted text role "data".

mock_open

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2445)

Unknown directive type "function".

```
.. function:: mock_open(mock=None, read_data=None)

   A helper function to create a mock to replace the use of :func:`open`. It works
   for :func:`open` called directly or used as a context manager.

   The *mock* argument is the mock object to configure. If ``None`` (the
   default) then a :class:`MagicMock` will be created for you, with the API limited
   to methods or attributes available on standard file handles.

   *read_data* is a string for the :meth:`~io.IOBase.read`,
   :meth:`~io.IOBase.readline`, and :meth:`~io.IOBase.readlines` methods
   of the file handle to return.  Calls to those methods will take data from
   *read_data* until it is depleted.  The mock of these methods is pretty
   simplistic: every time the *mock* is called, the *read_data* is rewound to
   the start.  If you need more control over the data that you are feeding to
   the tested code you will need to customize this mock for yourself.  When that
   is insufficient, one of the in-memory filesystem packages on `PyPI
   <https://pypi.org>`_ can offer a realistic filesystem for testing.

   .. versionchanged:: 3.4
      Added :meth:`~io.IOBase.readline` and :meth:`~io.IOBase.readlines` support.
      The mock of :meth:`~io.IOBase.read` changed to consume *read_data* rather
      than returning it on each call.

   .. versionchanged:: 3.5
      *read_data* is now reset on each call to the *mock*.

   .. versionchanged:: 3.8
      Added :meth:`__iter__` to implementation so that iteration (such as in for
      loops) correctly consumes *read_data*.
```

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2476); *backlink*

Unknown interpreted text role "func".

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

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2482); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2482); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2482); *backlink*

Unknown interpreted text role "meth".

Mocking context managers with a :class:`MagicMock` is common enough and fiddly enough that a helper function is useful.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2486); *backlink*

Unknown interpreted text role "class".

```
>>> m = mock_open()
>>> with patch('__main__.open', m):
...     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:

```
>>> with patch('__main__.open', mock_open(read_data='bibble')) as m:
...     with open('foo') as h:
...         result = h.read()
...
>>> m.assert_called_once_with('foo')
>>> assert result == 'bibble'
```

Autospeccing

Autospeccing is based on the existing :attr:`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 :exc:`TypeError` if they are called incorrectly.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2518); backlink

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2518); backlink

Unknown interpreted text role "exc".

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

:class:`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 :class:`Mock` api and the other is a more general problem with using mock objects.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2527); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2527); backlink

Unknown interpreted text role "class".

First the problem specific to :class:`Mock`. :class:`Mock` has two assert methods that are extremely handy: :meth:`~Mock.assert_called_with` and :meth:`~Mock.assert_called_once_with`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2532); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2532); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2532); backlink

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2532); backlink

Unknown interpreted text role "meth".

```
>>> 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 'mock' 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)  # Intentional typo!
```

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.

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2567); backlink

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2567); backlink

Unknown interpreted text role "attr".

```
>>> from urllib import request
>>> mock = Mock(spec=request.Request)
>>> mock.assret_called_with  # Intentional typo!
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()  # Intentional typo!
```

Auto-speccing solves this problem. You can either pass `autospec=True` to :func:`patch` / :func:`patch.object` or use the

:func:`create_autospec` function to create a mock with a spec. If you use the `autospec=True` argument to :func:`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.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2587); backlink

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2587); backlink

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2587); backlink

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2587); backlink

Unknown interpreted text role "func".

Here's an example of it in use:

```
>>> from urllib import request
>>> patcher = patch('__main__.request', autospec=True)
>>> mock_request = patcher.start()
>>> request is mock_request
True
>>> mock_request.Request
<MagicMock name='request.Request' spec='Request' id='...'>
```

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2606); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2606); backlink

Unknown interpreted text role "class".

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

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2622); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2622); backlink

Unknown interpreted text role "class".

```
>>> req.add_header('spam', 'eggs')
<MagicMock name='request.Request().add_header()' id='...'>
>>> req.add_header.assret_called_with  # Intentional typo!
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 :func:`patch` calls and then be protected against bugs due to typos and api changes.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2634); backlink

Unknown interpreted text role "func".

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2638); backlink

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2638); backlink

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2655); backlink

Unknown interpreted text role "func".

```
>>> from urllib import request
>>> mock_request = create_autospec(request)
>>> mock_request.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 [4].

A more serious problem is that it is common for instance attributes to be created in the :meth:`__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.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2655); backlink

Unknown interpreted text role "meth".

```
>>> class Something:
...   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()
...   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 :meth:`__init__`. Note that if you are only setting default attributes in :meth:`__init__` then providing them via class attributes (shared between instances of course) is faster too. e.g.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2696); backlink

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2696); backlink

Unknown interpreted text role "meth".

```
class Something:
    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:
...     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 :func:`patch` supports this - you can simply pass the alternative object as the *autospec* argument:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2721); backlink

Unknown interpreted text role "func".

```
>>> class Something:
...   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='...'>
```

[4] 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 :func:`dir` - that are done.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2742); *backlink*

Unknown interpreted text role "func".

Sealing mocks

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2750)

Unknown directive type "testsetup".

```
.. testsetup::

    from unittest.mock import seal
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\(cpython-main)(Doc)(library)unittest.mock.rst, line 2754)

Unknown directive type "function".

```
.. function:: seal(mock)

    Seal will disable the automatic creation of mocks when accessing an attribute of
    the mock being sealed or any of its attributes that are already mocks recursively.

    If a mock instance with a name or a spec is assigned to an attribute
    it won't be considered in the sealing chain. This allows one to prevent seal from
    fixing part of the mock object. ::

        >>> mock = Mock()
        >>> mock.submock.attribute1 = 2
        >>> mock.not_submock = mock.Mock(name="sample_name")
        >>> seal(mock)
        >>> mock.new_attribute  # This will raise AttributeError.
        >>> mock.submock.attribute2  # This will raise AttributeError.
        >>> mock.not_submock.attribute2  # This won't raise.

    .. versionadded:: 3.7
```