Mockify Documentation

Release 0...5

Maciej Wiatrzyk

Jul 27, 2019

Contents

Welcome to Mockify library documentation!

Mockify is a mocking toolkit for Python inspired by GMock (Google Mock) C++ framework. I was using GMock a lot during my 5 years of work as a C++ developer and really liked it for its expressive API. During that days I was still writing some Python code (mostly in Python 2.x) and for testing it I was using hand-written stubs when needed. When I used `unittest.mock` for the first time I noticed that it uses a very different approach than GMock I got used to, so I decided to start writing my own toolkit.

Currently, Mockify is supplied with following features:

- Creating mocks of standalone functions and Python objects
- Recording call expectations with fixed arguments and using **matchers**
- Checking if expectations are satisfied using one single `assert_satisfied` assertion method
- **Configuring recorded expectations:**
  - setting expected call count
  - recording single and repeated actions (a.k.a. side effects)
  - chaining actions

I hope you will find this library useful :-)

CHAPTER 1

User's Guide

1.1 Changelog

1.1.1 0.5.0

- Dependency management provided by pipenv
- Project's CLI provided by Invoke library
- Added `mockify.mock.Namespace` mock class
- Use Sphinx Read The Docs theme for documentation
- Class `mockify.mock.Object` can now be used without subclassing and has API similar to other mock classes
- Module *mockify.helpers* (was merged to library core)
- Module *mockify.times* (renamed to `mockify.cardinality`)
- Module *mockify.engine* is now available via `mockify`
- Modules *mockify.mock.function* and *mockify.mock.object* are now merged into `mockify.mock`

1.1.2 0.4.0

- Added strategies for dealing with unexpected calls

1.1.3 0.3.1

- Added frontend for mocking Python objects

1.1.4 0.2.1

- Updated copyright notice
- Added description to Alabaster Sphinx theme used for docs
- Added FunctionFactory mocking utility
- Changed Registry.assert_satisfied method to allow it to get mock names to check using positional args
- Script for running tests added (pytest wrapper)
- Updated copyright.py script and hardcode year the project was started and author's name

1.1.5 0.1.12

- First release published to PyPI

1.2 Installation

1.2.1 From PyPI using *virtualenv* and *pip*

Mockify can be installed by simply invoking this inside active virtual Python environment:

```
$ pip install mockify
```

This will install most recently deployed version of the library.

You can also add Mockify to your *requirements.txt* file if your project already has one. After that, you can install all dependencies at once using this command:

```
$ pip install -r requirements.txt
```

1.2.2 Directly from source using *virtualenv* and *pip*

You can also install Mockify directly from source code by simply invoking this command inside active virtual Python environment:

```
$ pip install git+https://gitlab.com/zef1r/mockify.git@[branch-or-tag]
```

This will allow you to install most recent version of the library that may not be released yet to PyPI. And also you will be able to install from any branch and tag.

1.2.3 Verifying installation

Once Mockify is installed, you can simply check if it works by invoking this code to print version of installed Mockify library:

```
import mockify

print(mockify.version)
```

And you're now ready to use Mockify for mocking things in your tests. Please proceed to *Tutorial* section of this documentation to learn how to use it.

1.3 Tutorial

1.3.1 Mocking functions

Using `Function` class

This is the most basic mocking utility. Instances of *mockify.mock.Function* are simply used to mock normal Python functions. You'll need such mocks for example to test code that uses callbacks.

To create function mock you need to import function mock utility:

```
>>> from mockify.mock import Function
```

Now you can create function mock using following boilerplate pattern:

```
>>> foo = Function('foo')
```

In the code above we have created function mock named *foo* and assigned it to variable of same name. Now object `foo` can be used like a normal Python function.

Most examples in this tutorial use function mocks.

Using `FunctionFactory` class

New in version 0.2.

You can also create function mocks in easier way by using *mockify.mock.FunctionFactory* class. Objects of this class simplify function mock creation by allowing it to be created by just attribute reading. For example, to create `foo` and `bar` function mocks you just need to execute following code:

```
>>> from mockify.mock import FunctionFactory
>>> factory = FunctionFactory()
>>> foo = factory.foo
>>> bar = factory.bar
```

Now both `foo` and `bar` are instances of *mockify.mock.Function* class. Of course you do not have to assign factory attribute to a variable - you can pass it directly, or even pass entire factory object to code being under test if needed.

Besides simplified mock creation this class also provides *mockify.mock.FunctionFactory.assert_satisfied()* method that checks if all mocks created by the factory are satisfied. Of course you can still do this by checking each individually:

```
>>> foo.assert_satisfied()
>>> bar.assert_satisfied()
```

But you will also achieve same result with this:

```
>>> factory.assert_satisfied()
```

1.3.2 Mocking objects

New in version 0.3.

Changed in version 0.5: Now you don't need to subclass, and the API is the same as for other mock classes.

To mock Python objects you need *mockify.mock.Object* class:

```
>>> from mockify.mock import Object
```

Now you can instantiate like any other mocking utility:

```
>>> mock = Object('mock')
```

Once you have a `mock` object, you can inject it into some code being under test. For example, let's have following function that interacts with some `obj` object:

```
>>> def uut(obj):
...     for x in obj.spam:
...         obj.foo(x)
...     return obj.bar()
```

To make *uut* function pass, we have to record expectations for:

- `spam` property to be read once
- `foo` to be called zero or more times (depending on what `spam` returns)
- `bar` to be called once and to return value that will also be used as *uut* function return value

We can of course create several combinations of expectations listed above (due to use of loop by *uut* function), but for the sake of simplicity let's configure `spam` to return `[1]` list, forcing `foo` to be called once with `1`:

```
>>> from mockify.actions import Return
>>> mock.spam.fget.expect_call().will_once(Return([1]))
<mockify.Expectation: mock.spam.fget()>
>>> mock.foo.expect_call(1)
<mockify.Expectation: mock.foo(1)>
>>> mock.bar.expect_call().will_once(Return(True))
<mockify.Expectation: mock.bar()>
```

Let's now call our `uut` function. Since we have covered all methods by our expectations, the mock call will now pass returning `True` (as we've set `bar` to return `True`):

```
>>> uut(mock)
True
```

And our mock is of course satisfied:

```
>>> mock.assert_satisfied()
```

1.3.3 Recording and verifying expectations

Mocks with no expectations

When mock is created, it has no expectations set, so it already is satisfied:

```
>>> foo = Function('foo')
>>> foo.assert_satisfied()
```

Mockify requires each mock to have all needed expectations recorded. But since `foo` has no expectations recorded yet, it cannot be called with any arguments and doing so will result in *mockify.exc.UninterestedCall* exception being raised when call is made. For example:

```
>>> foo(1, 2)
Traceback (most recent call last):
    ...
mockify.exc.UninterestedCall: foo(1, 2)
```

In order to allow `foo` to be called with `(1, 2)` as parameters, a matching expectation have to be recorded.

Mocks with one expectation

Let's go back to our mock `foo` defined in previous example and record a matching expectation:

```
>>> foo.expect_call(1, 2)
<mockify.Expectation: foo(1, 2)>
```

Now we've recorded that `foo` is expected to be called once with `(1, 2)` as positional arguments. Since the mock now has expectation, it is not satisfied now, as the expectation was not yet satisfied (previous failed call does not count):

```
>>> foo.assert_satisfied()
Traceback (most recent call last):
    ...
mockify.exc.Unsatisfied: following expectation is not satisfied:

at <doctest tutorial.rst[...]>:1
-----------------------------...
    Pattern: foo(1, 2)
   Expected: to be called once
     Actual: never called
```

As you can see, Mockify is presenting explanatory assertion message. You will know that only one expectation has failed and will no exactly which expectation it is as exact file and line number where the expectation was created are presented. Besides, you will also know how many times the mock is expected to be called with params matching *Pattern* and how many times it was actually called.

Each expectation can be in one of three states:

- **unsatisfied**,
- **satisfied**
- and **oversaturated**.

Currently, expectation from example above is in **unsatisfied** state, as it can still be satisfied by adequate number of matching mock calls. Let's then call a mock once to make it satisfied:

```
>>> foo(1, 2)
>>> foo.assert_satisfied()
```

Calling a mock more times than expected is possible and will not cause *mockify.exc.UninterestedCall* exception, as this is only used to point out that there were no expectations found that match given call parameters. But if expectation is already satisfied and is called again, it becomes **oversaturated** and the mock will stay unsatisfied for entire its lifetime:

```
>>> foo(1, 2)
>>> foo.assert_satisfied()
Traceback (most recent call last):
    ...
```

(continues on next page)

(continued from previous page)

```
mockify.exc.Unsatisfied: following expectation is not satisfied:

at <doctest tutorial.rst[...]>:1
-----------------------------...
    Pattern: foo(1, 2)
   Expected: to be called once
     Actual: called twice
>>> foo(1, 2)
>>> foo.assert_satisfied()
Traceback (most recent call last):
    ...
mockify.exc.Unsatisfied: following expectation is not satisfied:

at <doctest tutorial.rst[...]>:1
-----------------------------...
    Pattern: foo(1, 2)
   Expected: to be called once
     Actual: called 3 times
```

Mocks with many expectations

Usually each mock will have many expectations recorded, as the code being under test will usually use its dependencies more than once and with many different parameters. Let's have a look at following simple function:

```
>>> def example(count, callback):
...     for i in range(count):
...         callback(i)
```

This function is simply calling `callback` given number of times and passes current loop index as an argument on each iteration. If we want to test such function we basically need 3 tests:

1) Check if `callback` is not called when `count` is 0
2) Check if `callback` is called once with 0 when `count` is 1
3) Check if `callback` is triggered with 0, 1, ..., N-1 if `count` is N

First test can be written as simple as this one:

```
>>> callback = Function('callback')
>>> example(0, callback)
>>> callback.assert_satisfied()
```

If `callback` gets called, the test will fail with *`mockify.exc.UninterestedCall`* exception. There is also a nicer way to expect something to not happen but we'll talk about this a bit later.

Second test will look similar to what we've already used in previous examples:

```
>>> callback = Function('callback')
>>> callback.expect_call(0)
<mockify.Expectation: callback(0)>
>>> example(1, callback)
>>> callback.assert_satisfied()
```

And third test would look like this. For the sake of simplicity let's test our `example` function for N=2:

```
>>> callback = Function('callback')
>>> callback.expect_call(0)
<mockify.Expectation: callback(0)>
>>> callback.expect_call(1)
<mockify.Expectation: callback(1)>
>>> example(2, callback)
>>> callback.assert_satisfied()
```

As you can see, we have recorded two expectations. Mockify by default does not care about order of expectations, so the same can also be achieved if those expectations are reversed:

```
>>> callback = Function('callback')
>>> callback.expect_call(1)
<mockify.Expectation: callback(1)>
>>> callback.expect_call(0)
<mockify.Expectation: callback(0)>
>>> example(2, callback)
>>> callback.assert_satisfied()
```

Note: There are plans of implementing ordered expectations in future releases of Mockify.

Let's now leave our `example` function for a while and have a look at how unsatisfied assertion is rendered in case of multiple failed expectations. Let's create another mock with two expectations and call `assert_satisfied` on it:

```
>>> foo = Function('foo')
>>> foo.expect_call(1)
<mockify.Expectation: foo(1)>
>>> foo.expect_call(2)
<mockify.Expectation: foo(2)>
>>> foo.assert_satisfied()
Traceback (most recent call last):
    ...
mockify.exc.Unsatisfied: following 2 expectations are not satisfied:

at <doctest tutorial.rst[...]>:1
------------------------------...
    Pattern: foo(1)
   Expected: to be called once
     Actual: never called

at <doctest tutorial.rst[...]>:1
------------------------------...
    Pattern: foo(2)
   Expected: to be called once
     Actual: never called
```

If you now call a mock for the first time and check if it is satisfied, you'll see that only one unsatisfied expectation has left:

```
>>> foo(1)
>>> foo.assert_satisfied()
Traceback (most recent call last):
    ...
mockify.exc.Unsatisfied: following expectation is not satisfied:
```

(continues on next page)

(continued from previous page)

```
at <doctest tutorial.rst[...]>:1
------------------------------...
    Pattern: foo(2)
   Expected: to be called once
     Actual: never called
```

And if call one remaining expected call, the mock will become satisfied:

```
>>> foo(2)
>>> foo.assert_satisfied()
```

Using matchers

Sometimes you will need to write single expectation that is supposed to match multiple argument values. For this purpose, you will need **matchers**. Matchers are simple objects with overloaded `object.__eq__()` method. Thanks to matchers you will be able to write expectations that match entire classes of values, not exact ones. You will find predefined matchers in *`mockify.matchers`* module.

Let's now use *`mockify.matchers.Any`* matcher to show how it would look in practice:

```
>>> from mockify.matchers import _
>>> foo = Function('foo')
>>> foo.expect_call(_)
<mockify.Expectation: foo(_)>
>>> foo.expect_call(_)
<mockify.Expectation: foo(_)>
```

We've just recorded that we expect `foo` to be called twice with exactly one argument of any kind. So, for example, we can satisfy our mock with this:

```
>>> foo([])
>>> foo('spam')
>>> foo.assert_satisfied()
```

Matchers will also allow us to write complex patterns. For example, if mock is called with dict as an argument and the dict represents JSONRPC request (see: https://www.jsonrpc.org/specification), we could write expectation that we want our mock to be execute with request object, but no matter what is the method, params and ID:

```
>>> foo = Function('foo')
>>> foo.expect_call({'jsonrpc': '2.0', 'method': _, 'params': _, 'id': _})
<mockify.Expectation: foo({...})>
>>> foo({'jsonrpc': '2.0', 'method': 'spam', 'params': 123, 'id': 1})
>>> foo.assert_satisfied()
```

But if now the mock is called with different dict structure, the call will fail:

```
>>> foo({'jsonrpc': '2.0'})
Traceback (most recent call last):
    ...
mockify.exc.UninterestedCall: foo({'jsonrpc': '2.0'})
```

Dealing with unexpected calls

New in version 0.4.

Now you can change a default strategy for handling uninterested calls for your mocks.

To change a strategy you need to create a custom *mockify.Registry* object and use it as a **registry** for your mock classes.

For example, you can change the strategy to *ignore*, so all unexpected mock calls will simply be ignored:

```
>>> from mockify import Registry

>>> registry = Registry(uninterested_call_strategy='ignore')

>>> mock = Function('mock', registry=registry)
>>> mock(1, 2)
>>> mock(1, 2, c=3)
>>> mock()

>>> mock.assert_satisfied()
```

And now your mock will only fail if you have an unsatisfied expectation:

```
>>> mock.expect_call('spam')
<mockify.Expectation: mock('spam')>
>>> mock.assert_satisfied()
Traceback (most recent call last):
    ...
mockify.exc.Unsatisfied: following expectation is not satisfied:
<BLANKLINE>
at <doctest tutorial.rst[74]>:1
------------------------------
    Pattern: mock('spam')
   Expected: to be called once
     Actual: never called
```

1.3.4 Configuring expectation objects

So far, we've done nothing with `mockify..Expectation` object `expect_call` method returns. But it has a lot of very handy features that we are going to discuss right now.

Expecting a mock to be never called

It is very tricky to expect something to never happen as there are infinite number of possibilities. Besides, especially if it takes time to execute test, after how many seconds should we say that somethid *did not happpen*? But sometimes you may need to expect a mock to be never called.

Let's go back to our `example` function defined before. There was a test that callback is never called. The test looked like this:

```
>>> callback = Function('callback')
>>> example(0, callback)
>>> callback.assert_satisfied()
```

Although it works fine, there is not visible what we are expecting. Same test can be done like this:

```
>>> from mockify.matchers import _
>>> callback = Function('callback')
>>> callback.expect_call(_).times(0)
```

(continues on next page)

(continued from previous page)

```
<mockify.Expectation: callback(_)>
>>> example(0, callback)
>>> callback.assert_satisfied()
```

As you can see, we've used *mockify.Expectation.times()* method and called it with 0, meaning that we expect `callback` to be called 0 times. Now the test looks more expressive, but as stated in the beginning, expecting something to never happen is tricky. No matter if we call `example` function, other function or even nothing instead, the test will still pass:

```
>>> from mockify.matchers import _
>>> callback = Function('callback')
>>> callback.expect_call(_).times(0)
<mockify.Expectation: callback(_)>
>>> callback.assert_satisfied()
```

Just like normally expectation has expected call count set to one, modifying it with `times(0)` sets this counter to 0, so mock is already satisfied. Situtation changes when mock gets called:

```
>>> callback(0)
>>> callback.assert_satisfied()
Traceback (most recent call last):
    ...
mockify.exc.Unsatisfied: following expectation is not satisfied:

at <doctest tutorial.rst[...]>:1
----------------------------...
    Pattern: callback(_)
   Expected: to be never called
     Actual: called once
```

Expecting a mock to be called given number of times

So far, if we needed to expect a mock to be called more than once we've recorded two or more expectations with same parameters. But there is a better way of doing this.

Let's go back to our `example` function and third test. We can rewrite it in following way:

```
>>> callback = Function('callback')
>>> callback.expect_call(_).times(2)
<mockify.Expectation: callback(_)>
>>> example(2, callback)
>>> callback.assert_satisfied()
```

But actually we've verified only that mock is called twice each time with any argument. So in fact, if `example` calls a mock with fixed argument, then the test above will still pass. Therefore, we need another matcher to ensure that mock is called with valid arguments. For that purpose, we'll use *mockify.matchers.SaveArg*:

```
>>> from mockify.matchers import SaveArg
>>> count = SaveArg()
>>> callback = Function('callback')
>>> callback.expect_call(count).times(2)
<mockify.Expectation: callback(SaveArg)>
>>> example(2, callback)
>>> callback.assert_satisfied()
```

(continues on next page)

(continued from previous page)

```
>>> count.called_with == [0, 1]
True
```

Using *mockify.matchers.SaveArg* you will also have to do some additional assertions like in example above.

Method `mockify..Expectation.times()` allows to configure more then just fixed expected number of calls. For more information go to the `mockify.times` module documentation.

Single actions

Besides setting how many times each mock is expected to be called and with what arguments, you can also record actions to be executed on each mock call. For example, we can tell a mock to return given value when it gets called. To do this, we need to use `mockify..Expectation.will_once()` method:

```
>>> from mockify.actions import Return
>>> foo = Function('foo')
>>> foo.expect_call().will_once(Return(1))
<mockify.Expectation: foo()>
```

If you now check if mock is satisfied, you'll notice that there is additional information of what action is going to be executed next:

```
>>> foo.assert_satisfied()
Traceback (most recent call last):
    ...
mockify.exc.Unsatisfied: following expectation is not satisfied:

at <doctest tutorial.rst[...]>:1
-----------------------------...
    Pattern: foo()
     Action: Return(1)
   Expected: to be called once
     Actual: never called
```

So if you now call a mock, it will return 1 and will be satisfied:

```
>>> foo()
1
>>> foo.assert_satisfied()
```

But if you now call a mock again it will end up with an exception:

```
>>> foo()
Traceback (most recent call last):
    ...
mockify.exc.OversaturatedCall: at <doctest tutorial.rst[...]>:1: foo(): no more␣
↪actions recorded for call: foo()
```

This is a very special situation, as when actions are recorded it is assumed that the mock should always return *something*. Therefore, failing to do that is treated as exception currently.

Note: There are plans to implement default actions, so there will be no such exception in that case, but a default action will be executed instead. But mock will not be satisfied anyway.

For more actions please proceed to the *mockify.actions* documentation.

Action chains

You can chain `mockify..Expectation.will_once()` method invocations to end up with action chains being recorded, so each time when mock is called, next action in a chain is executed. For example, you can record expectation that mock is going to be called twice, returning 1 on first call and 2 on second call:

```
>>> foo = Function('foo')
>>> foo.expect_call().will_once(Return(1)).will_once(Return(2))
<mockify.Expectation: foo()>
```

When you now check if mock is satisfied, you will be informed that it is expected to be called twice and that next action is `Return(1)`:

```
>>> foo.assert_satisfied()
Traceback (most recent call last):
    ...
mockify.exc.Unsatisfied: following expectation is not satisfied:

at <doctest tutorial.rst[...]>:1
-----------------------------...
    Pattern: foo()
     Action: Return(1)
   Expected: to be called twice
     Actual: never called
```

If you now call a mock, it will return 1:

```
>>> foo()
1
```

If you now check if it is satisfied, you will notice that one more call is needed and that next action will be `Return(2)`:

```
>>> foo.assert_satisfied()
Traceback (most recent call last):
    ...
mockify.exc.Unsatisfied: following expectation is not satisfied:

at <doctest tutorial.rst[...]>:1
-----------------------------...
    Pattern: foo()
     Action: Return(2)
   Expected: to be called twice
     Actual: called once
```

Finally, if you call a mock for the second time it will return 2 and mock will become satisfied:

```
>>> foo()
2
>>> foo.assert_satisfied()
```

You can of course record different actions type for each call. For list of available built-in actions or instructions of how to make custom ones please refer to the *`mockify.actions`* module documentation.

Repeated actions

Repeated actions allow to set single action that will keep being executed each time the mock is called. By default, if mock has repeated action set it can be called any number of times, so mock with repeated action set is initially

satisfied. Repeated actions are recorded using `mockify..Expectation.will_repeatedly()` method:

```
>>> foo = Function('foo')
>>> foo.expect_call().will_repeatedly(Return(1))
<mockify.Expectation: foo()>
>>> foo.assert_satisfied()
```

And you can call mock with such defined expectation any times you want. For example, lets call it 3 times. The mock will return 1 on each call and still will be satisfied:

```
>>> for _ in range(3):
...     foo()
1
1
1
>>> foo.assert_satisfied()
```

You can also use `mockify..Expectation.times()` method to set expected call count on a repeated action. For example, if you want to record repeated action that can be executed at most twice, you would write following:

```
>>> from mockify.times import AtMost
>>> foo = Function('foo')
>>> foo.expect_call().will_repeatedly(Return(1)).times(AtMost(2))
<mockify.Expectation: foo()>
```

Such expectation is already satisfied (as at most twice is 0, 1 or 2 calls):

```
>>> foo.assert_satisfied()
```

But right now if you call a mock 3 times, the mock will no longer be satisfied:

```
>>> for _ in range(3):
...     foo()
1
1
1
>>> foo.assert_satisfied()
Traceback (most recent call last):
    ...
mockify.exc.Unsatisfied: following expectation is not satisfied:

at <doctest tutorial.rst[...]>:1
----------------------------...
    Pattern: foo()
     Action: Return(1)
   Expected: to be called at most twice
     Actual: called 3 times
```

1.3.5 Recording complex expectations

Currently we've used all of the features independently, but actually it is possible to record expectations that are combination of those. For example, you can record few single actions, and one repeated:

```
>>> foo = Function('foo')
>>> foo.expect_call().will_once(Return(1)).will_once(Return(2)).will_
↪repeatedly(Return(3))
```

(continues on next page)

(continued from previous page)

```
<mockify.Expectation: foo()>
```

Such mock will be expected to be called at least twice, as there are two single actions in the chain recorded:

```
>>> foo.assert_satisfied()
Traceback (most recent call last):
    ...
mockify.exc.Unsatisfied: following expectation is not satisfied:

at <doctest tutorial.rst[...]>:1
------------------------------...
    Pattern: foo()
     Action: Return(1)
   Expected: to be called at least twice
     Actual: never called
```

If now the mock is called for the fist time it will return 1, for the second time - 2, and after that it will keep returning 3. And of course it will be satisfied, as all single actions were consumed:

```
>>> foo()
1
>>> foo()
2
>>> for _ in range(3):
...     foo()
3
3
3
>>> foo.assert_satisfied()
```

You can also set expected call count for repeated action:

```
>>> foo = Function('foo')
>>> foo.expect_call().will_once(Return(1)).will_repeatedly(Return(2)).times(2)
<mockify.Expectation: foo()>
```

Now the mock will have to be called exactly 3 times:

```
>>> foo.assert_satisfied()
Traceback (most recent call last):
    ...
mockify.exc.Unsatisfied: following expectation is not satisfied:

at <doctest tutorial.rst[...]>:1
------------------------------...
    Pattern: foo()
     Action: Return(1)
   Expected: to be called 3 times
     Actual: never called
>>> foo()
1
>>> foo()
2
>>> foo()
2
>>> foo.assert_satisfied()
```

Even such combinations are possible:

```
>>> foo = Function('foo')
>>> foo.expect_call().will_once(Return(1)).will_repeatedly(Return(2)).times(2).will_
↪once(Return(3))
<mockify.Expectation: foo()>
```

And this time the mock is expected to be called 4 times:

```
>>> foo.assert_satisfied()
Traceback (most recent call last):
    ...
mockify.exc.Unsatisfied: following expectation is not satisfied:

at <doctest tutorial.rst[...]>:1
------------------------------...
    Pattern: foo()
     Action: Return(1)
   Expected: to be called 4 times
     Actual: never called
>>> foo()
1
>>> for _ in range(2):
...     foo()
2
2
>>> foo()
3
>>> foo.assert_satisfied()
```

1.4 API Reference

1.4.1 mockify - Library core

Library core module.

class mockify.**Call**(*name*, *args=None*, *kwargs=None*)

Bases: object

Binds mock name with arguments it was called with or it is expected to be called with.

Call objects are created in mock frontends (like *mockify.mock.Function* mock class) by methods expected_call and __call__ by simply passing their argument to *Call* constructor.

Instances of this class are comparable. Two *Call* objects are equal if and only if all attributes (name, args and kwargs) are the same. For example:

```
>>> Call('foo') == Call('foo')
True
>>> Call('foo') != Call('bar')
True
>>> Call('foo', (1, 2), {'c': 3}) == Call('foo', (1, 2), {'c': 3})
True
```

Call objects can also be created with use of **matchers**, for example *mockify.matchers.Any*, that will match any value:

```
>>> from mockify.matchers import _
>>> Call('foo', (_, _)) == Call('foo', (1, 2))
True
>>> Call('foo', (_, _)) == Call('foo', (3, 4))
True
```

Parameters

- **name** – Function or method name.
- **args** – Positional arguments
- **kwargs** – Named arguments

args
Mock positional args.

classmethod create(**args*, ***kwargs*)
Factory method for easier *Call* object creating.

You must give at least one positional argument - the name. All other will be passed to constructor's **args** and **kwargs** parameters.

New in version 0.5.

kwargs
Mock named args.

name
Mock name.

class mockify.**Expectation**(*expected_call*, *filename*, *lineno*)
Bases: object

Class representing single expectation.

Instances of this class are normally created by registry objects using *Registry.expect_call()* method. Each instance of this class is correlated with exactly one mockify.engine.Call object representing expected mock call pattern.

After *Expectation* object is created by call to some expect_call method, it can be mutated using following methods:

- *times()*
- *will_once()*
- *will_repeatedly()*

Parameters

- **call** – Instance of mockify.engine.Call representing expected mock call pattern
- **filename** – File name were this expectation was created
- **lineno** – Line number where this expectation was created

__call__(*call*)
Call this expectation object.

If given call object does not match *expected_call* then this method will raise TypeError exception.

Otherwise, total call count is increased by one and:

- if actions are recorded, then next action is executed and its result returned or *mockify.exc.OversaturatedCall* exception is raised if there are no more actions
- if there are no actions recorded, just `None` is returned

expected_call
Instance of `mockify.engine.Call` representing expected mock call pattern.

This basically is exactly the same *Call* object as was passed to *Expectation* constructor.

format_action()
Return textual representation of next action to be executed.

This method uses action's `__str__` method to render action name.

Returns `None` if there were no actions recorded or all were consumed.

This is used by *mockify.exc.Unsatisfied* exception when rendering error message.

format_actual()
Return textual representation of how many times this expectation was called so far.

This is used by *mockify.exc.Unsatisfied* exception when rendering error message.

format_expected()
Return textual representation of how many times this expectation is expected to be called.

This is used by *mockify.exc.Unsatisfied* exception when rendering error message.

format_location()
Return textual representation of place (filename and lineno) where this expectation was created.

Basically, it just returns `[filename]:[lineno]` string, where `filename` and `lineno` are given via *Expectation* constructor.

is_satisfied()
Check if this expectation is satisfied.

match(*call*)
Check if *expected_call* matches `call`.

times(*expected_count*)
Record how many times this expectation is expected to be called.

Parameters **expected_count** – Expected call count.

This can be either integer number (exact call count) or instance of one of classes from `mockify.times` module.

will_once(*action*)
Attach action to be executed when this expectation gets consumed.

This method can be used several times, making action chains. Once expectation is consumed, next action is executed and removed from the list. If there are no more actions, another call will fail with *mockify.exc.OversaturatedCall* exception.

After this method is used, you can also use *will_repeatedly()* to record repeated action that will get executed after all single actions are consumed.

Parameters **action** – Action to be executed.

See *mockify.actions* for details.

will_repeatedly(*action*)
Attach repeated action to be executed when this expectation is called.

This method is used to record one action that gets executed each time this expectation object is called. By default, when repeated action is recorded, expectation can be called any number of times (including zero).

After setting repeated action, you can also set expected call count using *times()*.

Parameters **action** – Action to be executed.

See *mockify.actions* for details.

class mockify.**Registry**(*expectation_class=None*, *uninterested_call_strategy='fail'*)
Bases: object

Acts like a database for *Expectation* objects.

This class is used as a backend for higher level mocking utilities (a.k.a. frontends), like *mockify.mock.Function* mocking class. It provides methods to record, lookup and verifying of expectations.

There can be many instances of registry classes, or one that can be shared between various mock frontends. For example, you can create one registry in setup code, then create various mocks inside your tests, to finally trigger *assert_satisfied()* of that single registry in test's teardown code. Or you can just use frontends with their defaults. It is completely up to you.

Parameters

- **expectation_class** – This is optional.

  Used to give custom subclass of *Expectation* to be used inside this registry.

- **uninterested_call_strategy** – Setup the way how uninterested calls are treated.

  Following values are available:

  – *fail* - issue *mockify.exc.UninterestedCall* exception on each unexpectedly called mock (default)

  – *ignore* - do nothing with uninterested calls

  – *warn* - issue a warning on each uninterested call

  New in version 0.4.

__call__(*call*)
Call a mock.

When this method is called, registry performs a lookup of matching unsatisfied expectations and calls first expectation found. If there are no matching expectation, then *mockify.exc.UninterestedCall* exception is raised. If there are matching expectations but all are satisfied, then last is called (making it oversaturated).

Parameters **call** – Instance of mockify.engine.Call class representing mock being called

assert_satisfied(**names*)
Assert that all expectations are satisfied.

If there is at least one unsatisfied expectation, then this method will raise *mockify.exc.Unsatisfied* exception containing list of failed expectations.

This method can be called as many times as you want.

Changed in version 0.2: Accepts names of mocks to check as positional args. If one or more names are given, then this method limits checking only to mocks of matching names.

expect_call(*call*, *filename*, *lineno*)
Register expectation.

Returns instance of `expectation_class` (usually *Expectation*) representing newly created expectation.

Parameters

- **call** – Instance of `mockify.engine.Call` class representing exact mock call or a pattern (if created with matchers) that is expected to be executed
- **filename** – Path to file were expectation is created
- **lineno** – Line number (inside `filename`) where expectation is created

`mockify.`**assert_satisfied**(**subjects*)
Context manager for verifying multiple subjects at once.

Each passed subject must have `assert_satisfied` method defined, so it can be used with *mockify.mock.Function* or `mockify.engine.Registry` instances for example.

Basically, the role of this helper is to ensure that all subjects become satisfied after leaving wrapped context. For example:

```
>>> from mockify.mock import Function
>>> foo = Function('foo')
>>> bar = Function('bar')
>>> foo.expect_call()
<mockify.Expectation: foo()>
>>> bar.expect_call().times(0)
<mockify.Expectation: bar()>
>>> with assert_satisfied(foo, bar):
...     foo()
```

And that's it - you don't have to explicitly check if `foo` and `bar` are satisfied, because the helper will do it for you. And also it emphasizes part of code that actually uses given mocks.

1.4.2 mockify.mock - Classes for mocking things

Classes for mocking things.

class `mockify.mock.`**Object**(*name*, *methods=None*, *properties=None*, *registry=None*)
Bases: `object`

Class for mocking Python objects.

Changed in version 0.5: New API introduced.

New in version 0.3.

Since version 0.5 this class provides a new API that complies with the one used by other mock classes.

You can now create mock objects directly, without subclassing:

```
mock = Object('mock')
```

Method calls are now recorded like this:

```
mock.foo.expect_call(1, 2)
```

And for recording property get/set expectations you write:

```
mock.bar.fset.expect_call(123)
mock.bar.fget.expect_call().will_once(Return(123))
mock.baz.fget.expect_call().will_once(Return(456))
```

And you can still subclass this class and provide a set of methods and properties, like in this example:

```
class Dummy(Object):
    __methods__ = ['foo']
    __properties__ = ['bar']
```

Parameters

- **name** – Name of mocked object
- **methods** – Sequence of names of methods to be mocked.

  If this is given, then the only allowed methods will be the ones from given sequence. Attempt to access any other will result in `AttributeError` being raised.
- **properties** – Sequence of names of properties to be mocked.

  Use is the same as for **methods** parameter.
- **registry** – Instance of *mockify.Registry* class.

  If not given, a default one will be created for this mock object.

assert_satisfied()
Assert that all expected method/property calls are satisfied.

expect_call(*__name__*, **args*, ***kwargs*)
Record method call expectation.

Deprecated since version 0.5: See *Object* for a brief example of how to use new API.

expect_get(*__name__*)
Record property get expectation.

Deprecated since version 0.5: See *Object* for a brief example of how to use new API.

expect_set(*__name__*, *value*)
Record property set expectation.

Deprecated since version 0.5: See *Object* for a brief example of how to use new API.

class mockify.mock.**Function**(*name*, *registry=None*)
Bases: `object`

Class for mocking Python functions.

Example usage:

```
>>> foo = Function('foo')
>>> foo.expect_call(1, 2).times(2)
<mockify.Expectation: foo(1, 2)>
>>> for _ in range(2):
...     foo(1, 2)
>>> foo.assert_satisfied()
```

Parameters

- **name** – Mock function name

- **registry** – This is optional.

  Use this to pass custom instance of `mockify.engine.Registry` class if you need to share it between multiple frontends. Sharing is useful for example to check if all mocks are satisfied using one `assert_satisfied` call:

```
>>> from mockify import Registry
>>> reg = Registry()
>>> foo = Function('foo', registry=reg)
>>> bar = Function('bar', registry=reg)
>>> foo.expect_call()
<mockify.Expectation: foo()>
>>> bar.expect_call()
<mockify.Expectation: bar()>
>>> foo()
>>> bar()
>>> reg.assert_satisfied()
```

assert_satisfied()
Assert that this function mock is satisfied.

This method just calls `mockify.engine.Registry.assert_satisfied()` with `name` given via constructor as an argument.

expect_call(**args*, ***kwargs*)
Record call expectation.

This method creates `mockify.engine.Call` instance giving it `args` and `kwargs`, fetches file and line number from current call stack and triggers `mockify.engine.Registry.expect_call()` and returns expectation object it produces.

class mockify.mock.**FunctionFactory**(*registry=None*)
Bases: `object`

Helper factory class for easier function mocks creating.

This helper can be created with no params or with `mockify.engine.Registry` instance as parameter. It provides an easy way of function mock creating by simply getting factory attributes that become function mock names. Once such attribute is get for the first time, *`Function`* instance is created, and later it is just returned.

This allows to create function mocks as easy as in this example:

```
>>> factory = FunctionFactory()
>>> factory.foo.expect_call()
<mockify.Expectation: foo()>
>>> factory.bar.expect_call(1, 2)
<mockify.Expectation: bar(1, 2)>
```

Then pass to some unit under test:

```
>>> def unit_under_test(foo, bar):
...     foo()
...     bar(1, 2)
>>> unit_under_test(factory.foo, factory.bar)
```

To finally check if all mocks registered in one *`FunctionFactory`* object are satisfied using one single call:

```
>>> factory.assert_satisfied()
```

assert_satisfied()
Check if all function mocks registered by this factory are satisfied.

This method simply calls `mockify.engine.Registry.assert_satisfied()` with names of all created mocks as arguments.

class mockify.mock.**Namespace**(*name*, *registry=None*, *mock_class=None*)
Bases: `object`

Used to mock functions that are behind some kind of a namespace.

New in version 0.5.

Look at following code:

```
if os.path.isfile(path):
    do_something_with_file(path)
```

It is very common pattern of how `os` is used by Python applications. And basically *Namespace* can be used to make it easier to mock such statements. You simply do it like this:

```
os = Namespace('os')
os.path.isfile.expect_call('/foo/bar/baz.txt').will_once(Return(True))
```

And now you can call such mocked statement:

```
assert os.path.isfile('/foo/bar/baz.txt')
```

There is no namespace nesting limit.

Parameters

- **name** – Mock name.

  This will be used as a prefix for all namespaced mocks created by this class.

- **registry** – Instance of *mockify.Registry* to be used.

  If not given, a default one will be created for this mock object.

- **mock_class** – Mock class to be used for "leaf" nodes.

  By default, this is *mockify.mock.Function*, but you can give any other mock class here.

assert_satisfied()
Check if all mocks created within this namespace are satisfied.

name
Name of this namespace mock.

This will be a root for all nested namespaces.

1.4.3 mockify.actions - Classes for recording side effects

Module containing predefined actions that can be used as argument for `Expectation.will_once()` or `Expectation.will_repeatedly()`.

Basically, any class containing following methods is considered an **action**:

`__str__(self)`

Returning string representation of an action.

This is used for error reporting.

`__call__(self, *args, **kwargs)`

Method that is called when mock is called.

Entire action logic goes in here.

class mockify.actions.**Invoke**(*func*)
Bases: object

Makes mock invoking given function when called.

When mock is called, all arguments (if there are any) are passed to the `func` and its return value is returned as mock's return value.

Parameters **func** – Function to be executed

class mockify.actions.**Raise**(*exc*)
Bases: object

Makes mock raising given exception when called.

Parameters **exc** – Instance of exception to be raised

class mockify.actions.**Return**(*value*)
Bases: object

Makes mock returning given value when called.

Parameters **value** – Value to be returned

1.4.4 mockify.cardinality - Classes for setting expected call cardinality

Module containing set of classes to be used with `mockify.engine.Expectation.times()` method.

You can also create your own classes to be used with that method. The only thing required from such class is to implement following interface:

is_satisfied(self, actual) Return `True` if `actual` call count is satisfied by `self`, or `False` otherwise.

Here, `actual` is absolute call count expectation received so far. It is completely implementation-specific of which values of `actual` are said to be *satisfied* and which are not. For example, *`Exactly`* will compare `actual` with fixed value (given via constructor) and return `True` only if those two are equal.

adjust_by(self, minimal) Adjust `self` by current `minimal` expected call count and return new instance of `type(self)`.

In some complex expectation there could be a situation in which expectation must be computed again. This is not visible for library user, but must be done behind the scenes to properly process expectations. Such situation can be presented in this example:

```
>>> from mockify.actions import Return
>>> from mockify.mock import Function
>>> foo = Function('foo')
>>> foo.expect_call(1, 2).will_once(Return(1)).will_
↪repeatedly(Return(2)).times(2)
<mockify.Expectation: foo(1, 2)>
```

(continues on next page)

(continued from previous page)

```
>>> foo(1, 2)
1
>>> foo(1, 2)
2
>>> foo(1, 2)
2
>>> foo.assert_satisfied()
```

In example above we've used `times(2)` to tell that last repeated action is expected to be called twice, but real expected call count is 3 times, as `will_once` is used. Behind the scenes, this is recalculated using this metho.

format_expected(self) Return textual representation of expected call count.

This is used by *mockify.exc.Unsatisfied* exception when error message is being rendered.

minimal(sefl) ***(property)*** Property containing minimal call count that is considered valid for given instance.

For example, for *AtLeast* or *Exactly* it would be just its constructor argument, for :class'AtMost' it will be 0, for *Between* it will be its `minimal` argument.

class mockify.cardinality.**AtLeast**(*minimal*)
Bases: object

Used to set minimal expected call count.

If this is used, then expectation is said to be satisfied if actual call count is not less that `minimal`.

Parameters **minimal** – Integer value representing minimal expected call count

class mockify.cardinality.**AtMost**(*maximal*)
Bases: object

Used to set maximal expected call count.

If this is used, then expectation is said to be satisfied if actual call count is not greater than `maximal`.

Parameters **maximal** – Integer value representing maximal expected call count

class mockify.cardinality.**Between**(*minimal*, *maximal*)
Bases: object

Used to set a range of valid call counts.

If this is used, then expectation is said to be satisfied if actual call count is not less than `minimal` and not greater than `maximal`.

Parameters

- **minimal** – Integer value representing minimal expected call count
- **maximal** – Integer value representing maximal expected call count

class mockify.cardinality.**Exactly**(*expected*)
Bases: object

Used to expect fixed call count to be made.

You do not have to use this class explicitly as its instances are automatically created when you call `times` method with integer value as argument.

Parameters **expected** – Integer value representing expected call count

1.4.5 mockify.matchers - Classes for wildcarding expected arguments

Module containing predefined matchers.

A matcher is every class that inherits from *Matcher* and implements following methods:

__repr__(self) Return matcher's text representation.

__eq__(self, other) Check if `self` is *equal* to `other`.

Here we use standard Python `__eq__` operator as it will be automatically executed by Python no matter where the matcher is used. But *equality* definition is completely up to the matcher implementation.

class mockify.matchers.**Any**

Bases: *mockify.matchers.Matcher*

Matcher that matches any value.

It is available also as _ (underscore) single instance that can be imported from this module.

For example, you can record expectation that mock must be called with one positional argument of any value but exactly 3 times:

```
>>> from mockify.matchers import _
>>> from mockify.mock import Function
>>> foo = Function('foo')
>>> foo.expect_call(_).times(3)
<mockify.Expectation: foo(_)>
>>> for i in range(3):
...     foo(i)
>>> foo.assert_satisfied()
```

class mockify.matchers.**Matcher**

Bases: `object`

Base class for matchers.

class mockify.matchers.**SaveArg**

Bases: *mockify.matchers.Matcher*

Matcher that matches any value and keeps ordered track of unique values.

This can be used as a replacement for *Any* in case that you need to ensure that mock was called in specified order.

For example:

```
>>> from mockify.mock import Function
>>> arg = SaveArg()
>>> foo = Function('foo')
>>> foo.expect_call(arg).times(3)
<mockify.Expectation: foo(SaveArg)>
>>> for i in range(3):
...     foo(i)
>>> foo.assert_satisfied()
>>> arg.called_with == [0, 1, 2]
True
```

called_with

List of ordered unique values that this matcher was called with.

1.4.6 mockify.exc - Library exceptions

exception mockify.exc.**OversaturatedCall**(*expectation*, *call*)
Bases: TypeError

Raised when mock is called more times than expected.

This exception will be thrown only if mock has actions defined as it does not know what to do next if all expected actions were already executed.

Parameters

- **expectation** – Instance of mockify.engine.Expectation class representing expectation that was oversaturated
- **call** – Instance of mockify.engine.Call class representing mock call that oversaturated expectation

call
Instance of mockify.engine.Call passed to *OversaturatedCall* constructor.

expectation
Instance of mockify.engine.Expectation passed to *OversaturatedCall* constructor.

exception mockify.exc.**UninterestedCall**(*call*)
Bases: TypeError

Raised when uninterested mock is called.

Mockify requires each mock call to have matching expectation recorded. If none is found during call, then this exception is raised, terminating the test.

Parameters **call** – Instance of mockify.engine.Call class representing uinterested mock call

call
Instance of mockify.engine.Call passed to *UninterestedCall* constructor.

exception mockify.exc.**UninterestedGetterCall**(*name*)
Bases: *mockify.exc.UninterestedPropertyAccess*

Raised when uninterested property getter is called.

This will be raised if some system under test gets mock property that has no expectations set.

New in version 0.3.

exception mockify.exc.**UninterestedPropertyAccess**(*name*)
Bases: TypeError

Base class for exceptions signalling uninterested property access.

This situation occurs when object property is accessed without previous matching expectation being recorded.

New in version 0.3.

Parameters **name** – Property name

name
Name of property being accessed.

exception mockify.exc.**UninterestedSetterCall**(*name*, *value*)
Bases: *mockify.exc.UninterestedPropertyAccess*

Raised when uninterested property setter is called.

This will be raised if some system under test sets mock property that has no matching expectations set.

New in version 0.3.

value
Value property was set to.

exception mockify.exc.**Unsatisfied**(*expectations*)
Bases: AssertionError

Raised by mockify.engine.Registry.assert_satisfied() method when there is at least one unsatisfied expectation.

This exception displays explanatory information to the user:

- file location where unsatisfied expectation was recorded
- expected call pattern
- expected call count
- actual call count
- next action to be executed (if any)

Parameters **expectations** – List of mockify.engine.Expectation instances representing all unsatisfied expectations

expectations
Instance of mockify.engine.Expectation passed to *Unsatisfied* constructor.

1.5 License

Mockify is released under the terms of the MIT license.

Python Module Index

m

Index

Symbols

A

B

C

E

F

I

K

M

N

O

R

S

T

U

V

W