Mockify Documentation

Release 0.7.1

Maciej Wiatrzyk

Jun 17, 2020

Contents

CHAPTER 1

About Mockify

Mockify is a highly customizable and expressive mocking library for Python inspired by Google Mock C++ framework, but adopted to Python world.

Unlike tools like `unittest.mock`, Mockify is based on **expectations** that you record on your mocks **before** they are injected to code being under test. Each expectation represents arguments the mock is expected to be called with and provides sequence of **actions** the mock will do when called with that arguments. Actions allow to set a value to be returned, exception to be raised or just function to be called. Alternatively, if no actions should take place, you can just say how many times the mock is expected to be called. And all of these is provided by simple, expressive and easy to use API.

Here's a simple example:

```
from mockify import satisfied
from mockify.mock import Mock
from mockify.actions import Return

def invoke(func):
    return func()

def test_invoke_calls_func_returning_hello_world():
    func = Mock('func')
    func.expect_call().will_once(Return('Hello, world!'))

    with satisfied(func):
        assert invoke(func) == 'Hello, world!'
```

I hope you'll find this library useful.

CHAPTER 2

User's Guide

2.1 Installation

2.1.1 From PyPI using pipenv

If your project's dependencies are managed by **pipenv**, simply proceed to your project's directory and invoke following command:

```
$ pipenv install --dev mockify
```

That will install most recent version of the library and automatically add it into your project's development dependencies.

2.1.2 From PyPI using virtualenv and pip

If you are using **virtualenv** in your project, just activate it and invoke following command:

```
$ pip install mockify
```

That will install most recent 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
```

2.1.3 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 to PyPI yet. And also you will be able to install from any branch or tag.

2.1.4 Verifying installation

After installation you can print installed version of Mockify library using following command:

```
$ python -c "import mockify; print(mockify.__version__)"
```

That command will print version of installed Mockify library. If installation was not successful, the command will fail.

Now you should be able to start using Mockify.

2.2 Quickstart

2.2.1 Introduction

In this quickstart guide we are going to use test-driven development technique to write a class for parsing messages of some textual protocol for transferring bytes of data of known size. In this guide you will:

- get familiar with basic concepts of Mockify,
- learn how to create **mocks** and inject them to code being under test,
- learn how to record **expectations** and **actions** on that mocks,
- learn how to set expected call count on mocks,
- learn how to check if mocks were **satisfied** once test is ended,
- learn how to read some of Mockify assertions.

After going through this quickstart guide you will be able to use Mockify in your projects, but to learn even more you should also read *Tutorial* chapter, which covers some more advanced features. I hope you will enjoy Mockify.

Let's start then!

2.2.2 The XYZ protocol

Imagine you work in a team which was given a task to design and write a library for transferring binary data over a wire between two peers. There are no any special requirements for chunking data, re-sending chunks, handling connection failures etc. The protocol must be very simple, easy to implement in different languages, which other teams will start implementing once design is done, and easy to extend in the future if needed. Moreover, there is a preference for this protocol to be textual, like HTTP. So your team starts with a brainstorming session and came out with following protocol design proposal:

```
MAGIC_BYTES | \n | Version | \n | len(PAYLOAD) | \n | PAYLOAD
```

The protocol was named **XYZ** and single message of the protocol is composed of following parts, separated with single newline character:

MAGIC_BYTES The string `"XYZ"` with protocol name.

Used to identify beginning of **XYZ** messages in byte stream coming from remote peer.

Version Protocol version in string format.

Currently always `"1.0"`, but your team wants the protocol to be extensible in case when more features would have to be incorporated.

len(PAYLOAD) Length of `PAYLOAD` part in bytes, represented in string format.

PAYLOAD Message payload.

These are actual bytes that are transferred.

You and your team have presented that design to other teams, the design was accepted, and now every team starts implementing the protocol. Your team is responsible for Python part.

2.2.3 The XYZReader class

Your team has decided to divide work into following independent flows:

- Implementing higher level **StreamReader** and **StreamWriter** classes for reading/writing bytes from/to underlying socket, but with few additional features like reading/writing **exactly** given amount of bytes (socket on its own does not guarantee that),
- Implementing protocol **logic** in form of **XYZReader** and **XYZWriter** classes that depend on stream readers and writers, accordingly.

The only common point between these two categories of classes is the interface between protocol logic and streams. After internal discussion you and your team agreed for following interfaces:

```
class StreamReader:

    def read(self, count) -> bytes
        """Read exactly *count* data from underlying stream."""

    def readline(self) -> bytes
        """Read data from underlying stream and stop once newline is
        found.

        Newline is also returned, as a last byte.
        """

class StreamWriter:

    def write(self, buffer):
        """Write entire *buffer* to underlying stream."""
```

Now half of the team can work on implementation of those interfaces, while the other half - on implementation of protocol's logic. You will be writing **XYZReader** class.

2.2.4 Writing first test

Step 0: Mocking StreamReader interface

You know that **XYZReader** class logic must somehow use **StreamReader**. So you've started with following draft:

```
class XYZReader:

    def __init__(self, stream_reader):
        self._stream_reader = stream_reader

    def read(self):
        return b'Hello world!'
```

To instantiate that class you need to pass something as a *stream_reader* parameter. You know how this interface looks like, but don't have a **real** implementation, because it is under development by rest of your team. But you cannot wait until they're done - you have to **mock** it. And here Mockify comes in to help you.

First you need to import *mockify.mock.Mock* class:

```
from mockify.mock import Mock
```

This class can be used to mock things like functions, methods, calls via module, getters, setters and more. This is the only one class to create mocks. And now you can instantiate it into **StreamReader** mock by creating instance of **Mock** class giving it a name:

```
stream_reader = Mock('stream_reader')
```

As you can see, there is no interface defined yet. It will be defined soon. Now you can instantiate **XYZReader** class with the mock we've created earlier:

```
xyz_reader = XYZReader(stream_reader)
assert xyz_reader.read() == b'Hello world!'
```

And here's complete test at this step:

```
from mockify.mock import Mock

def test_read_xyz_message():
    stream_reader = Mock('stream_reader')
    xyz_reader = XYZReader(stream_reader)
    assert xyz_reader.read() == b'Hello world!'
```

Step 1: Reading magic bytes

Okay, you have first iteration ready, but in fact there is nothing really interesting happening yet. Let's now add some business logic. You know, that first part of **XYZ** message is `MAGIC_BYTES` string that should always be `"XYZ"`. To get first part of message from incoming payload you need to read it from underlying **StreamReader**. And since we've used newline-separated parts, we'll be using **readline()** method. Here's **XYZReader** class supplied with code for reading `MAGIC_BYTES`:

```
class XYZReader:

    def __init__(self, stream_reader):
        self._stream_reader = stream_reader

    def read(self):
        self._stream_reader.readline()
        return b'Hello world!'
```

And now let's run our test again. You'll see that it fails with *mockify.exc.UninterestedCall* exception:

```
>>> test_read_xyz_message()
Traceback (most recent call last):
    ...
mockify.exc.UninterestedCall: No expectations recorded for mock:

at <doctest default[0]>:7
-------------------------
Called:
  stream_reader.readline()
```

That exception is triggered when for called mock there are no **expectations** recorded. To make the test pass again you have to record expectation for **stream_reader.readline()** method on *stream_reader* mock. Expectations are recorded by calling **expect_call()** method with arguments (positional and/or keyword) you **expect** your mock to be called with. And that method has to be called on *readline* attribute of *stream_reader* mock object. Here's complete solution:

```
from mockify.mock import Mock

def test_read_xyz_message():
    stream_reader = Mock('stream_reader')
    stream_reader.readline.expect_call()
    xyz_reader = XYZReader(stream_reader)
    assert xyz_reader.read() == b'Hello world!'
```

Step 2: Reading version

Now let's go back to out **XYZReader** class and add instruction for reading `Version` part of **XYZ** protocol message:

```
class XYZReader:

    def __init__(self, stream_reader):
        self._stream_reader = stream_reader

    def read(self):
        self._stream_reader.readline()  # read magic bytes
        self._stream_reader.readline()  # read version
        return b'Hello world!'
```

If you now run the test again, you'll see it passes. That's not what we were expecting: we've changed the code, so the test should fail. But it doesn't. And that is due to the fact that we are missing one additional assertion.

Step 3: Using satisfied() context manager

In Mockify not all assertion errors will be caused by invalid or unexpected mock calls. If the call to mock finds matching expectation, it runs it. And running expectation can be in some situations as trivial as just increasing call counter, with no side effects. And that is what happened in previous test.

To make your test verify all aspects of mocks provided by Mockify, you have to check if mocks you were created are **satisfied** before your test ends. A mock is said to be satisfied if all its expectations are consumed during execution of tested code. Such check can be done in few ways, but this time let's use *`mockify.satisfied()`* context manager:

```
from mockify import satisfied
from mockify.mock import Mock

def test_read_xyz_message():
```

(continues on next page)

(continued from previous page)

```
    stream_reader = Mock('stream_reader')
    stream_reader.readline.expect_call()
    xyz_reader = XYZReader(stream_reader)
    with satisfied(stream_reader):
        assert xyz_reader.read() == b'Hello world!'
```

This context manager is created with mock object(-s) as argument(-s) and should wrap part of the test function where tested code is executed. If at least one of given mocks have at least one expectation **unsatisfied** (i.e. called less or more times than expected), then context manager fails with *mockify.exc.Unsatisfied* assertion. And that happens when our updated test is run:

```
>>> test_read_xyz_message()
Traceback (most recent call last):
    ...
mockify.exc.Unsatisfied: Following expectation is not satisfied:

at <doctest default[0]>:6
-------------------------
Pattern:
  stream_reader.readline()
Expected:
  to be called once
Actual:
  called twice
```

The error was caused by second call to **stream_reader.readline()** method (to read protocol version), but we have only one expectation recorded in our test. This time we know that test should be adjusted, but of course that could also mean (f.e. when test was passing before making changes) that tested code needs to be fixed.

Step 4: Using Expectation.times() method

Okay, we know that our expectation needs to be somehow extended to fix error from previous step. We can either double the expectation (i.e. copy and paste just below) or change expected call count, which is one by default. Let's go with a second approach.

When you call **expect_call()**, special *mockify.Expectation* object is created and returned. That object has few methods that can be used to refine the expectation. And one of these methods is *mockify.Expectation.times()*. Here's our fixed test with **stream_reader.readline()** expected to be called twice:

```
from mockify import satisfied
from mockify.mock import Mock

def test_read_xyz_message():
    stream_reader = Mock('stream_reader')
    stream_reader.readline.expect_call().times(2)
    xyz_reader = XYZReader(stream_reader)
    with satisfied(stream_reader):
        assert xyz_reader.read() == b'Hello world!'
```

As you can see, the expectation clearly says that it is expected to be called twice. And now our test is running fine, so let's go back to **XYZReader** class, because there are still two parts missing.

Step 5: Reading payload

So far we've read magic bytes and version of our **XYZ** protocol frame. In this section let's speed up a bit and read two remaining parts at once: payload size and payload. Here's updated **XYZReader** class:

```
class XYZReader:

    def __init__(self, stream_reader):
        self._stream_reader = stream_reader

    def read(self):
        self._stream_reader.readline()  # read magic bytes
        self._stream_reader.readline()  # read version
        payload_size = self._stream_reader.readline()  # read payload size (1)
        payload_size = payload_size.rstrip()  # trim ending newline (which is␣
↪included) (2)
        payload_size = int(payload_size)  # conversion to int (3)
        return self._stream_reader.read(payload_size)  # read payload (4)
```

Here we are once again calling **readline()** to get payload size as string ending with newline (1). Then ending newline is stripped (2) and payload size is converted to integer (3). Finally **read()** method is called, with calculated *payload_size* as an argument (4).

Now let's try to run our test we fixed before. The test will fail with following error:

```
>>> test_read_xyz_message()
Traceback (most recent call last):
    ...
AttributeError: 'NoneType' object has no attribute 'rstrip'
```

It fails on (2), during test code execution, not during checking if expectations are satisfied. This is caused by default value returned by mocked call, which is `None` - like for Python functions that does not return any values. To make our test move forward we need to change that default behavior.

Step 6: Introducing actions

Mockify provides so called **actions**, available via *mockify.actions* module. Actions are simply special classes that are used to override default behaviour of returning `None` when mock is called. You record actions directly on expectation object using one of two methods:

- *mockify.Expectation.will_once()* for recording chains of unique actions,
- or *mockify.Expectation.will_repeatedly()* for recording so called **repeated actions**.

In this example we'll cover use of first of that methods and also we'll use *mockify.actions.Return* action for setting return value. Here's a fixed test:

```
from mockify import satisfied
from mockify.mock import Mock
from mockify.actions import Return

def test_read_xyz_message():
    stream_reader = Mock('stream_reader')
    stream_reader.readline.expect_call().times(2)
    stream_reader.readline.expect_call().will_once(Return(b'12\n'))  # (1)
    xyz_reader = XYZReader(stream_reader)
    with satisfied(stream_reader):
        assert xyz_reader.read() == b'Hello world!'
```

We've added one more expectation on **readline()** (1) and recorded single action to return `b'12\n'` as mock's return value. So when **readline()** is called for the third time, recorded action is invoked, forcing it to return given bytes. Of course the test will move forward now, but it will fail again, but few lines later:

```
>>> test_read_xyz_message()
Traceback (most recent call last):
    ...
mockify.exc.UninterestedCall: No expectations recorded for mock:

at <doctest default[0]>:12
--------------------------
Called:
  stream_reader.read(12)
```

Yes, that's right - we did not record any expectations for **read()** method, and *mockify.exc.UninterestedCall* tells that. We need to fix that by recording adequate expectation.

Step 7: Completing the test

Here's our final complete and passing test with one last missing expectation recorded:

```
from mockify import satisfied
from mockify.mock import Mock
from mockify.actions import Return

def test_read_xyz_message():
    stream_reader = Mock('stream_reader')
    stream_reader.readline.expect_call().times(2)
    stream_reader.readline.expect_call().will_once(Return(b'12\n')) # (1)
    stream_reader.read.expect_call(12).will_once(Return(b'Hello world!'))  # (2)
    xyz_reader = XYZReader(stream_reader)
    with satisfied(stream_reader):
        assert xyz_reader.read() == b'Hello world!'
```

We've added **read()** expectation at (2). Note that this time it is expected to be called with an argument, which is the same as we've injected in (1), but converted to integer (as our tested code does).

2.2.5 Verifying magic bytes

So far we've written one test covering successful scenario of reading message from underlying stream. Let's take a look at our **XYZReader** class we've developed:

```
class XYZReader:

    def __init__(self, stream_reader):
        self._stream_reader = stream_reader

    def read(self):
        self._stream_reader.readline()  # read magic bytes (1)
        self._stream_reader.readline()  # read version (2)
        payload_size = self._stream_reader.readline()
        payload_size = payload_size.rstrip()
        payload_size = int(payload_size)
        return self._stream_reader.read(payload_size)
```

There are two NOK (not OK) scenarios missing:

- magic bytes verification (1),
- and version verification (2).

Let's start by writing test that handles checking if magic bytes received are equal to `b"XYZ"`. We've decided to raise **XYZError** exception (not yet declared) in case when magic bytes are different than expected. Here's the test:

```
import pytest

from mockify.mock import Mock
from mockify.actions import Return

def test_when_invalid_magic_bytes_received__then_xyz_error_is_raised():
    stream_reader = Mock('stream_reader')
    stream_reader.readline.expect_call().will_once(Return(b'ABC\n'))
    xyz_reader = XYZReader(stream_reader)
    with pytest.raises(XYZError) as excinfo:
        xyz_reader.read()
    assert str(excinfo.value) == "Invalid magic bytes: b'ABC'"
```

But that test will fail now, because we do not have **XYZError** defined:

```
>>> test_when_invalid_magic_bytes_received__then_xyz_error_is_raised()
Traceback (most recent call last):
    ...
NameError: name 'XYZError' is not defined
```

So let's define it:

```
class XYZError(Exception):
    pass
```

And if now run the test again, it will fail with *mockify.exc.OversaturatedCall* error, because we do not have that functionality implemented yet:

```
>>> test_when_invalid_magic_bytes_received__then_xyz_error_is_raised()
Traceback (most recent call last):
    ...
mockify.exc.OversaturatedCall: Following expectation was oversaturated:

at <doctest default[0]>:8
-------------------------
Pattern:
  stream_reader.readline()
Expected:
  to be called once
Actual:
  oversaturated by stream_reader.readline() at <doctest default[0]>:8 (no more␣
↪actions)
```

Now we need to go back to our **XYZReader** class and fix it by implementing exception raising when invalid magic bytes are received:

```
class XYZError(Exception):
    pass

class XYZReader:
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, stream_reader):
        self._stream_reader = stream_reader

    def read(self):
        magic_bytes = self._stream_reader.readline()
        magic_bytes = magic_bytes.rstrip()
        if magic_bytes != b'XYZ':
            raise XYZError("Invalid magic bytes: {!r}".format(magic_bytes))
        self._stream_reader.readline()
        payload_size = self._stream_reader.readline()
        payload_size = payload_size.rstrip()
        payload_size = int(payload_size)
        return self._stream_reader.read(payload_size)
```

And now our second test will run fine, but first one will fail:

```
>>> test_read_xyz_message()
Traceback (most recent call last):
    ...
AttributeError: 'NoneType' object has no attribute 'rstrip'
```

Let's have a look at our first test again:

```
from mockify import satisfied
from mockify.mock import Mock
from mockify.actions import Return

def test_read_xyz_message():
    stream_reader = Mock('stream_reader')
    stream_reader.readline.expect_call().times(2)  # (1)
    stream_reader.readline.expect_call().will_once(Return(b'12\n'))
    stream_reader.read.expect_call(12).will_once(Return(b'Hello world!'))
    xyz_reader = XYZReader(stream_reader)
    with satisfied(stream_reader):
        assert xyz_reader.read() == b'Hello world!'
```

As you can see, at (1) we are expecting **readline()** to be called twice, but we did not provided any value to be returned. And that was fine when we were implementing OK case, but since we have changed **XYZReader** class, we need to inject proper magic bytes here. Here's fixed OK case test:

```
from mockify import satisfied
from mockify.mock import Mock
from mockify.actions import Return

def test_read_xyz_message():
    stream_reader = Mock('stream_reader')
    stream_reader.readline.expect_call().will_once(Return(b'XYZ\n'))
    stream_reader.readline.expect_call()
    stream_reader.readline.expect_call().will_once(Return(b'12\n'))
    stream_reader.read.expect_call(12).will_once(Return(b'Hello world!'))
    xyz_reader = XYZReader(stream_reader)
    with satisfied(stream_reader):
        assert xyz_reader.read() == b'Hello world!'
```

2.2.6 Verifying version

Since third of our tests will be basically written in the same way as second one, let me just present final solution.

Here's **XYZReader** class with code that verifies version:

```
class XYZError(Exception):
    pass

class XYZReader:

    def __init__(self, stream_reader):
        self._stream_reader = stream_reader

    def read(self):
        magic_bytes = self._stream_reader.readline()
        magic_bytes = magic_bytes.rstrip()
        if magic_bytes != b'XYZ':
            raise XYZError("Invalid magic bytes: {!r}".format(magic_bytes))
        version = self._stream_reader.readline()
        version = version.rstrip()
        if version != b'1.0':
            raise XYZError("Unsupported version: {!r}".format(version))
        payload_size = self._stream_reader.readline()
        payload_size = payload_size.rstrip()
        payload_size = int(payload_size)
        return self._stream_reader.read(payload_size)
```

And here's our third test - the one that checks if exception is raised when invalid version is provided:

```
import pytest

from mockify.mock import Mock
from mockify.actions import Return

def test_when_invalid_version_received__then_xyz_error_is_raised():
    stream_reader = Mock('stream_reader')
    stream_reader.readline.expect_call().will_once(Return(b'XYZ\n')) # (1)
    stream_reader.readline.expect_call().will_once(Return(b'2.0\n')) # (2)
    xyz_reader = XYZReader(stream_reader)
    with pytest.raises(XYZError) as excinfo:
        xyz_reader.read()
    assert str(excinfo.value) == "Unsupported version: b'2.0'"  # (3)
```

Here we have two **readline()** expectations recorded. At (1) we've set valid magic bytes (we are not interested in exception raised at that point), and then at (2) we've set an unsupported version, causing **XYZError** to be raised. Finally, at (3) we are checking if valid exception was raised.

Of course we also had to fix our first test again, returning valid version instead of `None`:

```
from mockify import satisfied
from mockify.mock import Mock
from mockify.actions import Return

def test_read_xyz_message():
    stream_reader = Mock('stream_reader')
    stream_reader.readline.expect_call().will_once(Return(b'XYZ\n'))
    stream_reader.readline.expect_call().will_once(Return(b'1.0\n'))
```

(continues on next page)

(continued from previous page)

```
    stream_reader.readline.expect_call().will_once(Return(b'12\n'))
    stream_reader.read.expect_call(12).will_once(Return(b'Hello world!'))
    xyz_reader = XYZReader(stream_reader)
    with satisfied(stream_reader):
        assert xyz_reader.read() == b'Hello world!'
```

2.2.7 Refactoring tests

If you take a look at all three tests at once you'll see a some parts are basically copied and pasted. Creating *stream_reader* mock, instantiating **XYZReader** class and checking if mocks are satisfied can be done better if we use organize our tests with a class:

```
import pytest

from mockify import assert_satisfied
from mockify.mock import Mock
from mockify.actions import Return

class TestXYZReader:

    def setup_method(self):
        self.stream_reader = Mock('stream_reader')  # (1)
        self.uut = XYZReader(self.stream_reader)  # (2)

    def teardown_method(self):
        assert_satisfied(self.stream_reader)  # (3)

    def test_read_xyz_message(self):
        self.stream_reader.readline.expect_call().will_once(Return(b'XYZ\n'))
        self.stream_reader.readline.expect_call().will_once(Return(b'1.0\n'))
        self.stream_reader.readline.expect_call().will_once(Return(b'12\n'))
        self.stream_reader.read.expect_call(12).will_once(Return(b'Hello world!'))
        assert self.uut.read() == b'Hello world!'

    def test_when_invalid_magic_bytes_received__then_xyz_error_is_raised(self):
        self.stream_reader.readline.expect_call().will_once(Return(b'ABC\n'))
        with pytest.raises(XYZError) as excinfo:
            self.uut.read()
        assert str(excinfo.value) == "Invalid magic bytes: b'ABC'"

    def test_when_invalid_version_received__then_xyz_error_is_raised(self):
        self.stream_reader.readline.expect_call().will_once(Return(b'XYZ\n'))
        self.stream_reader.readline.expect_call().will_once(Return(b'2.0\n'))
        with pytest.raises(XYZError) as excinfo:
            self.uut.read()
        assert str(excinfo.value) == "Unsupported version: b'2.0'"
```

Tip: Alternatively you can use **fixtures** instead of **setup_method**() and **teardown_method**(). Fixtures are way more powerful. For more details please visit https://docs.pytest.org/en/latest/fixture.html.

We've moved mock (1) and unit under test (2) construction into **setup_method**() method and used *mockify.assert_satisfied()* function (3) in **teardown_method**(). That function works the same as *mockify.satisfied()*, but is not a context manager. Notice that we've also removed context manager from OK test, as

it is no longer needed.

Now, once tests are refactored, you can just add another tests without even remembering to check the mock before test is done - it all happens automatically. And the tests look much cleaner than before refactoring. There is even more: you can easily extract recording expectations to separate methods if needed.

2.2.8 Putting it all together

Here's once again complete **XYZReader** class:

```
class XYZError(Exception):
    pass

class XYZReader:

    def __init__(self, stream_reader):
        self._stream_reader = stream_reader

    def read(self):
        magic_bytes = self._stream_reader.readline()
        magic_bytes = magic_bytes.rstrip()
        if magic_bytes != b'XYZ':
            raise XYZError("Invalid magic bytes: {!r}".format(magic_bytes))
        version = self._stream_reader.readline()
        version = version.rstrip()
        if version != b'1.0':
            raise XYZError("Unsupported version: {!r}".format(version))
        payload_size = self._stream_reader.readline()
        payload_size = payload_size.rstrip()
        payload_size = int(payload_size)
        return self._stream_reader.read(payload_size)
```

And tests:

```
import pytest

from mockify import assert_satisfied
from mockify.mock import Mock
from mockify.actions import Return

class TestXYZReader:

    def setup_method(self):
        self.stream_reader = Mock('stream_reader')  # (1)
        self.uut = XYZReader(self.stream_reader)  # (2)

    def teardown_method(self):
        assert_satisfied(self.stream_reader)  # (3)

    def test_read_xyz_message(self):
        self.stream_reader.readline.expect_call().will_once(Return(b'XYZ\n'))
        self.stream_reader.readline.expect_call().will_once(Return(b'1.0\n'))
        self.stream_reader.readline.expect_call().will_once(Return(b'12\n'))
        self.stream_reader.read.expect_call(12).will_once(Return(b'Hello world!'))
        assert self.uut.read() == b'Hello world!'

    def test_when_invalid_magic_bytes_received__then_xyz_error_is_raised(self):
```

(continues on next page)

(continued from previous page)

```
        self.stream_reader.readline.expect_call().will_once(Return(b'ABC\n'))
        with pytest.raises(XYZError) as excinfo:
            self.uut.read()
        assert str(excinfo.value) == "Invalid magic bytes: b'ABC'"

    def test_when_invalid_version_received__then_xyz_error_is_raised(self):
        self.stream_reader.readline.expect_call().will_once(Return(b'XYZ\n'))
        self.stream_reader.readline.expect_call().will_once(Return(b'2.0\n'))
        with pytest.raises(XYZError) as excinfo:
            self.uut.read()
        assert str(excinfo.value) == "Unsupported version: b'2.0'"
```

And that's the end of quickstart guide :-)

Now you can proceed to *Tutorial* section, covering some more advanced features, or just try it out in your projects. Thanks for reaching that far. I hope you will find Mockify useful.

2.3 Tutorial

2.3.1 Creating mocks and recording expectations

Introduction

Since version 0.6 Mockify provides single *mockify.mock.Mock* class for mocking things. With that class you will be able to mock:

- functions,
- objects with methods,
- modules with functions,
- setters and getters.

That new class can create attributes when you first access them and then you can record **expectations** on that attributes. Furthermore, that attributes are **callable**. When you call one, it consumes previously recorded expectations.

To create a mock, you need to import *mockify.mock.Mock* class and instantiate it with a name of choice:

```
from mockify.mock import Mock

foo = Mock('foo')
```

That name should reflect what is being mocked and this should be function, object or module name. You can only use names that are valid Python identifiers or valid Python module names, with submodules separated with a period sign.

Now let's take a brief introduction to what can be done with just created *foo* object.

Mocking functions

Previously created *foo* mock can be used to mock a function or any other callable. Consider this example code:

```
def async_sum(a, b, callback):
    result = a + b
    callback(result)
```

We have "asynchronous" function that calculates sum of *a* and *b* and triggers given *callback* with a sum of those two. Now, let's call that function with *foo* mock object as a *callback*. This will happen:

```
>>> async_sum(2, 3, foo)
Traceback (most recent call last):
    ...
mockify.exc.UninterestedCall: No expectations recorded for mock:

at <doctest default[0]>:3
-------------------------
Called:
  foo(5)
```

Now you should notice two things:

- Mock object *foo* is **callable** and was called with 5 (2 + 3 = 5),
- Exception *mockify.exc.UninterestedCall* was raised, caused by lack of **expectations** on mock *foo*.

Raising that exception is a default behavior of Mockify. You can change this default behavior (see *mockify.Session.config* for more details), but it can be very useful, because it will make your tests fail early and you will see what expectation needs to be recorded to move forward. In our case we need to record **foo(5)** call expectation.

To do this you will need to call **expect_call()** method on *foo* object:

```
foo.expect_call(5)
```

Calling **expect_call()** records call expectation on rightmost mock attribute, which is *foo* in this case. Given arguments must match the arguments the mock will later be called with.

And if you call *async_sum* again, it will now pass:

```
from mockify import satisfied

with satisfied(foo):
    async_sum(2, 3, foo)
```

Note that we've additionally used *mockify.satisfied()*. It's a context manager for wrapping portions of test code that **satisfies** one or more given mocks. And mock is satisfied if all expectations recorded for it are satisfied, meaning that they were called **exactly** expected number of times. Alternatively, you could also use *mockify.assert_satisfied()* function:

```
from mockify import assert_satisfied

foo.expect_call(3)
async_sum(1, 2, foo)
assert_satisfied(foo)
```

That actually work in the same way as context manager version, but can be used out of any context, for example in some kind of teardown function.

Mocking objects with methods

Now let's take a look at following code:

```
class APIGateway:

    def __init__(self, connection):
```

(continues on next page)

(continued from previous page)

```
        self._connection = connection

    def list_users(self):
        return self._connection.get('/api/users')
```

This class implements a facade on some lower level *connection* object. Let's now create instance of *APIGateway* class. Oh, it cannot be created without a *connection* argument... That's not a problem - let's use a **mock** for that:

```
connection = Mock('connection')
gateway = APIGateway(connection)
```

If you now call *APIGateway.list_users()* method, you will see similar error to the one we had earlier:

```
>>> gateway.list_users()
Traceback (most recent call last):
    ...
mockify.exc.UninterestedCall: No expectations recorded for mock:

at <doctest default[0]>:7
-------------------------
Called:
  connection.get('/api/users')
```

And again, we need to record matching expectation to move test forward. To record method call expectation you basically need to do the same as for functions, but with additional attribute - a method object:

```
connection.get.expect_call('/api/users')
with satisfied(connection):
    gateway.list_users()
```

And now it works fine.

Mocking functions behind a namespace or module

This kind of mocking is extended version of previous one.

Now consider this example:

```
class APIGateway:

    def __init__(self, connection):
        self._connection = connection

    def list_users(self):
        return self._connection.http.get('/api/users')
```

We have basically the same example, but this time our *connection* interface was divided between various protocols. You can assume that *connection* object handles entire communication with external world by providing a facade to lower level libs. And *http* part is one of them.

To mock that kind of stuff you basically only need to add another attribute to *connection* mock, and call **expect_call()** on that attribute. Here's a complete example:

```
connection = Mock('connection')
gateway = APIGateway(connection)
```

(continues on next page)

(continued from previous page)

```
connection.http.get.expect_call('/api/users')
with satisfied(connection):
    gateway.list_users()
```

Creating ad-hoc data objects

Class *mockify.mock.Mock* can also be used to create ad-hoc data objects to be used as a response for example. To create one, you just need to instantiate it, and assign values to automatically created properties. Like in this example:

```
mock = Mock('mock')
mock.foo = 1
mock.bar = 2
mock.baz.spam.more_spam = 'more spam' # (1)
```

The most cool feature about data objects created this way is (1) - you can assign values to any nested attributes. And now let's get those values:

```
>>> mock.foo
1
>>> mock.bar
2
>>> mock.baz.spam.more_spam
'more spam'
```

Mocking getters

Let's take a look at following function:

```
def unpack(obj, *names):
    for name in names:
        yield getattr(obj, name)
```

That function yields attributes extracted from given *obj* in order specified by *names*. Of course it is a trivial example, but we'll use a mock in place of *obj* and will record expectations on property getting. And here's the solution:

```
from mockify.actions import Return

obj = Mock('obj')
obj.__getattr__.expect_call('a').will_once(Return(1))   # (1)
obj.__getattr__.expect_call('b').will_once(Return(2))   # (2)
with satisfied(obj):
    assert list(unpack(obj, 'a', 'b')) == [1, 2]   # (3)
```

As you can see, recording expectation of getting property on (1) and (2) is that you record **call expectation** on a magic method *__getattr__*. And similar to data objects, you can record getting attribute expectation at any nesting level - just prefix **expect_call()** with **__getattr__** attribute and you're done.

Mocking setters

Just like getters, setters can also be mocked with Mockify. The difference is that you will have to use **__setattr__.expect_call()** this time and obligatory give two arguments:

- attribute name,

- and value you expect it to be set with.

Here's a complete solution with a *pack* function - a reverse of the one used in previous example:

```
def pack(obj, **kwargs):
    for name, value in kwargs.items():
        setattr(obj, name, value)

obj = Mock('obj')
obj.__setattr__.expect_call('a', 1)
obj.__setattr__.expect_call('b', 2)
with satisfied(obj):
    pack(obj, a=1, b=2)
```

And that also work on nested attributes.

Correlated setters and getters

Setters and getters mocks are not correlated by default; if you set both **__setattr__** and **__getattr__** expectations on one property, those two will be treated as two separate things. Two correlate them, you will have to do record a bit more complex expectations and use *mockify.actions.Invoke* to store value in between. Here's an example:

```
from mockify.actions import Invoke
from mockify.matchers import Type

store = Mock('store') # (1)

obj = Mock('obj')
obj.__setattr__.expect_call('value', Type(int)).will_once(Invoke(setattr, store)) #␣
↪(2)
obj.__getattr__.expect_call('value').will_repeatedly(Invoke(getattr, store))  # (3)
```

In example above we are intercepting property setting and getting with **setattr()** and **getattr()** built-in functions invoked by *mockify.actions.Invoke* action. Those functions are bound with mock (1) acting as a data store and it will be used as first argument. Moreover, we've recorded setting expectation with a help of *mockify.matchers.Type* matcher (2), that will only match integer values. Finally, we've used **will_repeatedly()** at (3), so we are expecting any number of value reads after it was set. This is how it works in practice:

```
>>> obj.value = 123
>>> obj.value
123
>>> obj.value
123
```

2.3.2 Most common assertions

Uninterested mock calls

Just after mock object is created, it does not have any expectations recorded. Calling a mock with no expectations is by default not possible and results in *mockify.exc.UninterestedCall* assertion and test termination:

```
>>> from mockify.mock import Mock
>>> mock = Mock('mock')
>>> mock()
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
    ...
mockify.exc.UninterestedCall: No expectations recorded for mock:

at <doctest default[2]>:1
-------------------------
Called:
  mock()
```

That error will be raised for any attribute you would call on such mock with no expectations. That default behavior can be changed (see *Using sessions* for more details), but it can get really useful. For example, if you are writing tests for existing code, you could write tests in step-by-step mode, recording expectations one-by-one.

Unexpected mock calls

This new behavior was introduced in version 0.6.

It is meant to differentiate mocks that has **no** expectations from mocks that have **at least one**, but not matching **actual** call. This is illustrated by following example:

```
from mockify.mock import Mock

mock = Mock('mock')
mock.expect_call(1, 2)
```

We have mock *mock* that is expected to be called with 1 and 2 as two positional arguments. And now, if that mock is called with unexpected parameters, for instance 1 and **3**, *mockify.exc.UnexpectedCall* assertion will be raised:

```
>>> mock(1, 3)
Traceback (most recent call last):
    ...
mockify.exc.UnexpectedCall: No matching expectations found for call:

at <doctest default[0]>:1
-------------------------
Called:
  mock(1, 3)
Expected (any of):
  mock(1, 2)
```

That error is basically extended version of previous **uninterested call** error, with additional list of existing expectations. That will make it easier to decide if expectation has a typo, or if there is a bug in tested code.

Unsatisfied and satisfied mocks

All previously presented assertion errors can only be raised during mock call. But even if mock is called with expected parameters and for each call matching expectation is found, we still need a way to verify if expectations we've recorded are **satisfied**, which means that all are called expected number of times.

To check if mock is satisfied you can use *mockify.assert_satisfied()* function. This function can be used more than once, but usually the best place to check if mock is satisfied is at the end of test function.

Each newly created mock is already satisfied:

```
from mockify import assert_satisfied
from mockify.mock import Mock

foo = Mock('foo')

assert_satisfied(foo)
```

Let's now record some expectation:

```
foo.bar.expect_call('spam')
```

When expectation is recorded, then mock becomes **unsatisfied**, which means that it is not yet or not fully consumed. That will be reported with *mockify.exc.Unsatisfied* assertion:

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

at <doctest default[0]>:1
-------------------------
Pattern:
  foo.bar('spam')
Expected:
  to be called once
Actual:
  never called
```

The exception will print out all unsatisfied expectations with their:

- location in test code,
- call pattern that describes function or method with its parameters,
- expected call count of the pattern,
- and actual call count.

By reading exception we see that our method is expected to be called once and was never called. That's true, because we've only recorded an expectation so far. To make *foo* satisfied again we need to call the method with params that will match the expectation:

```
from mockify import satisfied

with satisfied(foo):
    foo.bar('spam')
```

In example above we've used *mockify.satisfied()* context manager instead of *mockify.assert_satisfied()* presented above. Those two work in exactly the same way, raising exactly the same exceptions, but context manager version is better suited for simple tests or when you want to mark part of test code that satisfies all given mocks.

If you now call our expected method again, the call will not raise any exceptions:

```
foo.bar('spam')
```

And even if you run it 5 more times, it will still just work:

```
for _ in range(5):
    foo.bar('spam')
```

But the mock will no longer be satisfied even after first of that additional calls:

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

at <doctest default[0]>:1
-------------------------
Pattern:
  foo.bar('spam')
Expected:
  to be called once
Actual:
  called 7 times
```

So once again, we have *mockify.exc.Unsatisfied* raised. But as you can see, the mock was called 7 times so far, while it still is expected to be called exactly once.

Why there was no exception raised on second call?

Well, this was made like this actually to make life easier. Mockify allows you to record very sophisticated expectations, including expected call count ranges etc. And when mock is called it does not know how many times it will be called during the test, so we must explicitly tell it that testing is done. And that's why *mockify.assert_satisfied()* is needed. Moreover, it is the only single assertion function you will find in Mockify (not counting its context manager counterpart).

2.3.3 Setting expected call count

Expecting mock to be called given number of times

When you create expectation, you **implicitly** expect your mock to be called **exactly once** with given given params:

```
from mockify import satisfied
from mockify.mock import Mock

foo = Mock('foo')
foo.expect_call(1, 2)
with satisfied(foo):
    foo(1, 2)
```

But what if we need our mock to be called exactly N-times?

First solution is simply to **repeat expectation exactly N-times**. And here's example test that follows this approach, expecting `foo` mock to be called exactly three times:

```
from mockify import satisfied
from mockify.mock import Mock

def func_caller(func, a, b, count):
    for _ in range(count):
        func(a, b)
```

(continues on next page)

(continued from previous page)

```
def test_func_caller():
    foo = Mock('foo')
    for _ in range(3):
        foo.expect_call(1, 2)
    with satisfied(foo):
        func_caller(foo, 1, 2, 3)
```

Although that will certainly work, it is not the best option, as it unnecessary complicates test code. So here's another example, presenting recommended solution for setting expected call count to fixed value:

```
def test_func_caller():
    foo = Mock('foo')
    foo.expect_call(1, 2).times(3)  # (1)
    with satisfied(foo):
        func_caller(foo, 1, 2, 3)
```

We've removed loop from test function and instead used *mockify.Expectation.times()* method (1), giving it expected number of calls to `foo(1, 2)`. Thanks to this, our expectation is self-explanatory and in case of unsatisfied assertion you will see that expected call count in error message:

```
>>> from mockify import assert_satisfied
>>> from mockify.mock import Mock
>>> foo = Mock('foo')
>>> foo.expect_call().times(3)
<mockify.Expectation: foo()>
>>> assert_satisfied(foo)
Traceback (most recent call last):
    ...
mockify.exc.Unsatisfied: Following expectation is not satisfied:

at <doctest default[3]>:1
-------------------------
Pattern:
  foo()
Expected:
  to be called 3 times
Actual:
  never called
```

Expecting mock to be never called

Although expecting something to never happen is a bit tricky, here we can use it to overcome *mockify.exc.UninterestedCall* and *mockify.exc.UnexpectedCall* assertions. Normally, if mock is called with parameters for which there are no matching expectations, the call will fail with one of mentioned exceptions. But you can change that to *mockify.exc.Unsatisfied* assertion with following simple trick:

```
from mockify import assert_satisfied
from mockify.mock import Mock

foo = Mock('foo')
foo.expect_call(-1).times(0)  # (1)

assert_satisfied(foo)
```

As you can see, the mock is satisfied despite the fact it **does have** an expectation recorded at (1). But that expectation

has expected call count set to zero with `times(0)` call. And that's the trick - you are explicitly **expecting** *foo* to be **never** called (or called zero times) with -1 as *an argument.

And now if you make a matching call, the mock will instantly become unsatisfied:

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

at <doctest default[0]>:5
-------------------------
Pattern:
  foo(-1)
Expected:
  to be never called
Actual:
  called once
```

And that's the whole trick.

Setting expected call count using cardinality objects

Previously presented *mockify.Expectation.times()* can also be used in conjunction with so called **cardinality objects** available via *mockify.cardinality* module.

Here's an example of setting **minimal** expected call count:

```
from mockify.mock import Mock
from mockify.cardinality import AtLeast

foo = Mock('foo')
foo.expect_call().times(AtLeast(1))  # (1)
```

In example above we've recorded expectation that `foo()` will be called **at least once** by passing *mockify.cardinality.AtLeast* instance to `times()` method. So currently it will not be satisfied, because it is not called yet:

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

at <doctest default[0]>:5
-------------------------
Pattern:
  foo()
Expected:
  to be called at least once
Actual:
  never called
```

But after it is called and made satisfied:

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

It will be satisfied forever - no matter how many times `foo()` will be called afterwards:

```
>>> for _ in range(10):
...     foo()
>>> assert_satisfied(foo)
```

Using the same approach you can also set:

- **maximal** call count (*mockify.cardinality.AtMost*),
- or **ranged** call count (*mockify.cardinality.Between*).

2.3.4 Recording actions

What are actions used for?

In Mockify, each mocked function by default returns `None` when called with parameters for which expectation was recorded:

```
from mockify.mock import Mock

foo = Mock('foo')
foo.expect_call(1)
foo.expect_call(2)

assert foo(1) is None
assert foo(2) is None
```

That behavior is a normal thing for mocking **command**-like functions, i.e. functions that **do something** by modifying internal state of an object, signalling only failures with various exceptions. That functions does not need or even must not return any values, and in Python function that does not return anything implicitly returns `None`.

But there are also **query**-like methods and that kind of methods **must** return a value of some kind, as we use them to obtain current state of some object.

So we basically have two problems to solve:

- How to force mocks of command-like functions to raise exceptions?
- How to force mocks of query-like functions to return various values, but different than `None`?

That's where **actions** come in. In Mockify you have various actions available via *mockify.actions* module. With actions you can, in addition to recording expectations, set what the mock will do when called with matching set of parameters. For example, you can:

- set value to be returned by mock (*mockify.actions.Return*),
- set exception to be raised by mock (*mockify.actions.Raise*),
- set function to be called by mock (*mockify.actions.Invoke*),
- or run your custom action (defined by subclassing *mockify.actions.Action* class).

Single actions

To record single action, you have to use *mockify.Expectation.will_once()* method and give it instance of action you want your mock to perform. For example:

```
from mockify.mock import Mock
from mockify.actions import Return

foo = Mock('foo')
foo.expect_call(1).will_once(Return('one'))   # (1)
foo.expect_call(2).will_once(Return('two'))   # (2)
```

We've recorded two expectations, and set a return value for each. Actions are tied together with expectations, so in our example we've recorded that `foo(1)` will return `'one'` (1), and that `foo(2)` will return `'two'`.

Mocks with actions set are not satisfied if there are actions left to be consumed. If we now check if *foo* is satisfied, *mockify.exc.Unsatisfied* will be raised with two unsatisfied expectations defined in (1) and (2) present:

```
>>> from mockify import assert_satisfied
>>> assert_satisfied(foo)
Traceback (most recent call last):
    ...
mockify.exc.Unsatisfied: Following 2 expectations are not satisfied:

at <doctest default[0]>:5
-------------------------
Pattern:
  foo(1)
Action:
  Return('one')
Expected:
  to be called once
Actual:
  never called

at <doctest default[0]>:6
-------------------------
Pattern:
  foo(2)
Action:
  Return('two')
Expected:
  to be called once
Actual:
  never called
```

Notice that the exception also shows an action to be performed next. That information is not present if you have no custom actions recorded. Let's now call *foo* with params matching previously recorded expectations:

```
>>> foo(1)
'one'
>>> foo(2)
'two'
>>> assert_satisfied(foo)
```

As you can see, the mock returned values we've recorded. And it is also satisfied now.

Action chains

It is also possible to record **multiple** actions on single expectation, simply by adding more *mockify.Expectation.will_once()* method calls:

```
from mockify.mock import Mock
from mockify.actions import Return

count = Mock('count')
count.expect_call().\
    will_once(Return(1)).\
    will_once(Return(2)).\
    will_once(Return(3))
```

In example above we've created a mock named *count*, and it will consume and invoke subsequent action on each call:

```
>>> count()
1
>>> count()
2
>>> count()
3
```

That's how action chains work. Of course each chain is tied to a particular expectation, so you are able to create different chains for different expectations. And you can have different actions in your chains, and even mix them.

When multiple single actions are recorded, then mock is implicitly expected to be called N-times, where N is a number of actions in a chain. But if you have actions recorded and mock gets called more times than expected, it will fail on mock call with *mockify.exc.OversaturatedCall*:

```
>>> count()
Traceback (most recent call last):
    ...
mockify.exc.OversaturatedCall: Following expectation was oversaturated:

at <doctest default[0]>:5
-------------------------
Pattern:
  count()
Expected:
  to be called 3 times
Actual:
  oversaturated by count() at <doctest default[0]>:1 (no more actions)
```

That error will only be raised if you are using actions. Normally, the mock would simply be unsatisfied. But it was added for a reason; if there are no more custom actions recorded and mock is called again, then it would most likely fail few lines later (f.e. due to invalid value type), but with stacktrace pointing to tested code, not to call of mocked function. And that would potentially be harder to debug.

Repeated actions

You also can record so called **repeated** actions with *mockify.Expectation.will_repeatedly()* method instead of previously used `will_once()`:

```
from mockify.mock import Mock
from mockify.actions import Return

foo = Mock('foo')
foo.expect_call().will_repeatedly(Return(123))  # (1)
```

Repeated actions defined like in (1) can be executed any number of times, including zero, so currently mock *foo* is already satisfied:

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

Repeated actions are useful when you need same thing to be done every single time the mock is called. So if `foo()` is now called, it will always return `123`, as we've declared in (1):

```
>>> [foo() for _ in range(4)]
[123, 123, 123, 123]
```

And *foo* will always be satisfied:

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

Repeated actions with cardinality

You can also declare repeated actions that can only be executed given number of times by simply adding call to *`mockify.Expectation.times()`* method just after `will_repeatedly()`:

```
from mockify.mock import Mock
from mockify.actions import Return

foo = Mock('foo')
foo.expect_call().will_repeatedly(Return(123)).times(1)   # (1)
```

Such declared expectation will have to be executed exactly once. But of course you can use any cardinality object from *`mockify.cardinality`* to record even more complex behaviors. The difference between such constrained repeated actions and actions recorded using `will_once()` is that repeated actions cannot be oversaturated - the mock will simply keep returning value we've set, but of course will no longer be satisfied:

```
>>> foo()
123
>>> assert_satisfied(foo)
>>> foo()
123
>>> assert_satisfied(foo)
Traceback (most recent call last):
    ...
mockify.exc.Unsatisfied: Following expectation is not satisfied:

at <doctest default[0]>:5
-------------------------
Pattern:
  foo()
Action:
  Return(123)
Expected:
  to be called once
Actual:
  called twice
```

Using chained and repeated actions together

It is also possible to use both single and repeated actions together, like in this example:

```
from mockify.mock import Mock
from mockify.actions import Return

foo = Mock('foo')
foo.expect_call().\
    will_once(Return(1)).\
    will_once(Return(2)).\
    will_repeatedly(Return(3))
```

Such declared expectations have implicitly set **minimal** expected call count that is equal to number of actions recorded using `will_once()`. So currently the mock is not satisfied:

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

at <doctest default[0]>:5
-------------------------
Pattern:
  foo()
Action:
  Return(1)
Expected:
  to be called at least twice
Actual:
  never called
```

But the mock becomes satisfied after it is called twice:

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

And at this point it will continue to be satisfied - no matter how many times it is called after. And for every call it will execute previously set repeated action:

```
>>> foo()
3
>>> foo()
3
>>> assert_satisfied(foo)
```

Using chained and repeated actions with cardinality

You can also record expectations like this one:

```
from mockify.mock import Mock
from mockify.actions import Return

foo = Mock('foo')
foo.expect_call().\
    will_once(Return(1)).\
    will_once(Return(2)).\
```

(continues on next page)

(continued from previous page)

```
    will_repeatedly(Return(3)).\
    times(2)  # (1)
```

Basically, this is a constrained version of previous example in which repeated action is expected to be called only twice. But total expected call count is 4, as we have two single actions recorded:

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

at <doctest default[0]>:5
-------------------------
Pattern:
  foo()
Action:
  Return(1)
Expected:
  to be called 4 times
Actual:
  never called
```

Now let's satisfy the expectation by calling a mock:

```
>>> [foo() for _ in range(4)]
[1, 2, 3, 3]
>>> assert_satisfied(foo)
```

Since last of your actions is a repeated action, you can keep calling the mock more times:

```
>>> foo()
3
```

But the mock will no longer be satisfied, as we've recorded at (1) that repeated action will be called exactly twice:

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

at <doctest default[0]>:5
-------------------------
Pattern:
  foo()
Action:
  Return(3)
Expected:
  to be called 4 times
Actual:
  called 5 times
```

2.3.5 Managing multiple mocks

Introduction

So far we've discussed situations where single mock object is suitable and fits well. But those are very rare situations, as usually you will need more than just one mock. Let's now take a look at following Python code:

```
import hashlib
import base64


class AlreadyRegistered(Exception):
    pass


class RegisterUserAction:

    def __init__(self, database, crypto, mailer):
        self._database = database
        self._crypto = crypto
        self._mailer = mailer

    def invoke(self, email, password):
        session = self._database.session()
        if session.users.exists(email):
            raise AlreadyRegistered("E-mail {!r} is already registered".format(email))
        password = self._crypto.hash_password(password)
        session.users.add(email, password)
        self._mailer.send_confirm_registration_to(email)
        session.commit()
```

That classes implements business logic of user registration process:

- User begins registration by entering his/her e-mail and password,
- System verifies whether given e-mail is already registered,
- System adds new user to users database and marks as "confirmation in progress",
- System sends confirmation email to the User with confirmation link.

That use case has dependencies to database, e-mail sending service and service that provides some sophisticated way of generating random numbers suitable for cryptographic use. Now let's write one test for that class:

```
from mockify import satisfied
from mockify.mock import Mock
from mockify.actions import Return

def test_register_user_action():
    session = Mock('session')  # (1)
    database = Mock('database')
    crypto = Mock('crypto')
    mailer = Mock('mailer')

    database.session.\
        expect_call().will_once(Return(session))
    session.users.exists.\
        expect_call('foo@bar.com').will_once(Return(False))
    crypto.hash_password.\
        expect_call('p@55w0rd').will_once(Return('***'))
    session.users.add.\
```

(continues on next page)

(continued from previous page)

```
        expect_call('foo@bar.com', '***')
    mailer.send_confirm_registration_to.\
        expect_call('foo@bar.com')
    session.commit.\
        expect_call()

    action = RegisterUserAction(database, crypto, mailer)

    with satisfied(database, session, crypto, mailer):  # (2)
        action.invoke('foo@bar.com', 'p@55w0rd')
```

We had to create 3 mocks + one additional at (1) for mocking database session. And since we have 4 mock objects, we also need to remember to verify them all at (2). And remembering things may lead to bugs in the test code. But Mockify is supplied with tools that will help you deal with that.

Using mock factories

First solution is to use *mockify.mock.MockFactory* class. With that class you will be able to create mocks without need to use *mockify.mock.Mock* directly. Morever, mock factories will not allow you to duplicate mock names and will automatically track all created mocks for you. Besides, all mocks created by one factory will share same **session** object and that is important for some of Mockify's features.

Here's our previous test rewritten to use mock factory instead of several mock objects:

```
from mockify import satisfied
from mockify.mock import MockFactory
from mockify.actions import Return

def test_register_user_action():
    factory = MockFactory()  # (1)
    session = factory.mock('session')
    database = factory.mock('database')
    crypto = factory.mock('crypto')
    mailer = factory.mock('mailer')

    database.session.\
        expect_call().will_once(Return(session))
    session.users.exists.\
        expect_call('foo@bar.com').will_once(Return(False))
    crypto.hash_password.\
        expect_call('p@55w0rd').will_once(Return('***'))
    session.users.add.\
        expect_call('foo@bar.com', '***')
    mailer.send_confirm_registration_to.\
        expect_call('foo@bar.com')
    session.commit.\
        expect_call()

    action = RegisterUserAction(database, crypto, mailer)

    with satisfied(factory):  # (2)
        action.invoke('foo@bar.com', 'p@55w0rd')
```

Although the code did not change a lot in comparison to previous version, we've introduced a major improvement. At (1) we've created a **mock factory** instance, which is used to create all needed mocks. Also notice, that right now we only check factory object at (2), so we don't have to remember all the mocks we've created. That saves a lot

of problems later, when test is modified; each new mock will most likely be created using *factory* object and it will automatically check that new mock.

Using mock factories with test suites

Mock factories work the best with test suites containing **setup** and **teardown** customizable steps executed before and after every single test. Here's once again our test, but this time in form of test suite (written as an example, without use of any specific framework):

```
from mockify import assert_satisfied
from mockify.mock import MockFactory
from mockify.actions import Return

class TestRegisterUserAction:

    def setup(self):
        self.factory = MockFactory()  # (1)

        self.session = self.factory.mock('session')    # (2)
        self.database = self.factory.mock('database')
        self.crypto = self.factory.mock('crypto')
        self.mailer = self.factory.mock('mailer')

        self.database.session.\
            expect_call().will_repeatedly(Return(self.session))  # (3)

        self.uut = RegisterUserAction(self.database, self.crypto, self.mailer)  # (4)

    def teardown(self):
        assert_satisfied(self.factory)  # (5)

    def test_register_user_action(self):
        self.session.users.exists.\
            expect_call('foo@bar.com').will_once(Return(False))
        self.crypto.hash_password.\
            expect_call('p@55w0rd').will_once(Return('***'))
        self.session.users.add.\
            expect_call('foo@bar.com', '***')
        self.mailer.send_confirm_registration_to.\
            expect_call('foo@bar.com')
        self.session.commit.\
            expect_call()

        self.uut.invoke('foo@bar.com', 'p@55w0rd')
```

Notice, that we've moved factory to `setup()` method (1), and created all mocks inside it (2) along with unit under test instance (4). Also notice that obtaining database session (3) was also moved to setup step and made optional with `will_repeatedly()`. Finally, our factory (and every single mock created by it) is verified at (5), during teardown phase of test execution. Thanks to that we have only use case specific expectations in test method, and a common setup code, so it is now much easier to add more tests to that class.

Note: If you are using **pytest**, you can take advantage of **fixtures** and use those instead of setup/teardown methods:

```
import pytest
```

(continues on next page)

(continued from previous page)

```
from mockify import satisfied
from mockify.mock import MockFactory


@pytest.fixture
def mock_factory():
    factory = MockFactory()
    with satisfied(factory):
        yield factory


def test_something(mock_factory):
    mock = mock_factory.mock('mock')
    # ...
```

Using sessions

A core part of Mockify library is a **session**. Sessions are instances of *mockify.Session* class and their role is to provide mechanism for storing recorded expectations, and matching them with calls being made. Normally sessions are created automatically by each mock or mock factory, but you can also give it explicitly via *session* argument:

```
from mockify import Session
from mockify.mock import Mock, MockFactory

session = Session()  # (1)

first = Mock('first', session=session)  # (2)
second = MockFactory(session=session)  # (3)
```

In example above, we've explicity created session object (1) and gave it to mock *first* (2) and mock factory *second* (3), which now share the session. This means that all expectations registered for mock *first* or any of mocks created by factory *second* will be passed to a common session object. Some of Mockify features, like **ordered expectations** (see *Recording ordered expectations*) will require that to work. Although you don't have to create one common session for all your mocks, creating it explicitly may be needed if you want to:

- override some of Mockify's default behaviors (see *mockify.Session.config* for more info),
- write a common part for your tests.

For the sake of this example let's stick to the last point. And now, let's write a base class for our test suite defined before:

```
from mockify import Session

class TestCase:

    def setup(self):
        self.mock_session = Session()  # (1)

    def teardown(self):
        self.mock_session.assert_satisfied()  # (2)
```

As you can see, nothing really interesting is happening here. We are creating session (1) in **setup** section and checking it it is satisfied (2) in **teardown** section. And here comes our test from previous example:

```
class TestRegisterUserAction(TestCase):

    def setup(self):
        super().setup()

        self.factory = MockFactory(session=self.mock_session)  # (1)

        self.session = self.factory.mock('session')
        self.database = self.factory.mock('database')
        self.crypto = self.factory.mock('crypto')
        self.mailer = self.factory.mock('mailer')

        self.database.session.\
            expect_call().will_repeatedly(Return(self.session))

        self.uut = RegisterUserAction(self.database, self.crypto, self.mailer)

    def test_register_user_action(self):
        self.session.users.exists.\
            expect_call('foo@bar.com').will_once(Return(False))
        self.crypto.hash_password.\
            expect_call('p@55w0rd').will_once(Return('***'))
        self.session.users.add.\
            expect_call('foo@bar.com', '***')
        self.mailer.send_confirm_registration_to.\
            expect_call('foo@bar.com')
        self.session.commit.\
            expect_call()

        self.uut.invoke('foo@bar.com', 'p@55w0rd')
```

As you can see, `teardown()` method was completely removed because it was no longer needed - all mocks are checked by one single call to *`mockify.Session.assert_satisfied()`* method in base class. The part that changed is a `setup()` function that triggers base class setup method, and a mock factory (1) that is given a session. With this approach you only implement mock checking once - in a base class for your tests. The only thing you have to remember is to give a session instance to either factory, or each of your mocks for that to work.

2.3.6 Using matchers

Introduction

So far we've been recording expectations with fixed argument values. But Mockify provides to you a very powerful mechanism of **matchers**, available via *`mockify.matchers`* module. Thanks to the matchers you can record expectations that will match more than just a single value. Let's take a brief tour of what you can do with matchers!

Recording expectations with matchers

Let's take a look at following code that we want to test:

```
import uuid


class ProductAlreadyExists(Exception):
    pass
```

(continues on next page)

(continued from previous page)

```
class AddProductAction:

    def __init__(self, database):
        self._database = database

    def invoke(self, category_id, name, data):
        if self._database.products.exists(category_id, name):
            raise ProductAlreadyExists()
        product_id = str(uuid.uuid4())  # (1)
        self._database.products.add(product_id, category_id, name, data)  # (2)
```

That code represents a business logic of adding some kind of product into database. The product is identified by a **name** and **category**, and there cannot be more than one product of given name inside given category. But tricky part is at (1), where we calculate UUID for our new product. That value is random, and we are passing it into `products.add()` method, which will be mocked. How to mock that, when we don't know what will the value be? And here comes **the matchers**:

```
from mockify import satisfied
from mockify.mock import Mock
from mockify.actions import Return
from mockify.matchers import _  # (1)


def test_add_product_action():
    database = Mock('database')

    database.products.exists.\
        expect_call('dummy-category', 'dummy-name').\
        will_once(Return(False))
    database.products.add.\
        expect_call(_, 'dummy-category', 'dummy-name', {'description': 'A dummy␣
↪product'})  # (2)

    action = AddProductAction(database)
    with satisfied(database):
        action.invoke('dummy-category', 'dummy-name', {'description': 'A dummy product
↪'})
```

We've used a **wildcard** matcher imported in (1), and placed it as first argument of our expectation (2). That underscore object is in fact instance of *mockify.matchers.Any* and is basically **equal** to every possible Python object, therefore it will match any possible UUID value, so our test will pass.

Of course you can also use another matcher if you need a more strict check. For example, we can use *mockify.matchers.Regex* to check if this **is** a real UUID value:

```
from mockify import satisfied
from mockify.mock import Mock
from mockify.actions import Return
from mockify.matchers import Regex

any_uuid = Regex(r'^[a-z0-9]{8}-[a-z0-9]{4}-[a-z0-9]{4}-[a-z0-9]{4}-[a-z0-9]{12}$')


def test_add_product_action():
```

(continues on next page)

(continued from previous page)

```
    database = Mock('database')

    database.products.exists.\
        expect_call('dummy-category', 'dummy-name').\
        will_once(Return(False))
    database.products.add.\
        expect_call(any_uuid, 'dummy-category', 'dummy-name', {'description': 'A␣
↪dummy product'})

    action = AddProductAction(database)
    with satisfied(database):
        action.invoke('dummy-category', 'dummy-name', {'description': 'A dummy product
↪'})
```

Combining matchers

You can also combine matchers using `|` and `&` binary operators.

For example, if you want to expect values that can only be integer numbers or lower case ASCII strings, you can combine *mockify.matchers.Type* and *mockify.matchers.Regex* matchers like in this example:

```
from mockify.mock import Mock
from mockify.actions import Return
from mockify.matchers import Type, Regex

mock = Mock('mock')
mock.\
    expect_call(Type(int) | Regex(r'^[a-z]+$', 'LOWER_ASCII')).\
    will_repeatedly(Return(True))
```

And now let's try it:

```
>>> mock(1)
True
>>> mock('abc')
True
>>> mock(3.14)
Traceback (most recent call last):
    ...
mockify.exc.UnexpectedCall: No matching expectations found for call:

at <doctest default[2]>:1
--------------------------
Called:
  mock(3.14)
Expected (any of):
  mock(Type(int) | Regex(LOWER_ASCII))
```

In the last line we've called our mock with float number which is neither integer, nor lower ASCII string. And since it did not matched our expectation, *mockify.exc.UnexpectedCall* was raised - the same that would be raised if we had used fixed values in expectation.

And now let's try with one more example.

This time we are expecting only positive integer numbers. To expect that we can combine previously introduced `Type` matcher with *mockify.matchers.Func* matcher. The latter is very powerful, as it accepts any custom function. Here's our expectation:

```
from mockify.mock import Mock
from mockify.actions import Return
from mockify.matchers import Type, Func

mock = Mock('mock')
mock.\
    expect_call(Type(int) & Func(lambda x: x > 0, 'POSITIVE_ONLY')).\
    will_repeatedly(Return(True))
```

And now let's do some checks:

```
>>> mock(1)
True
>>> mock(10)
True
>>> mock(3.14)
Traceback (most recent call last):
    ...
mockify.exc.UnexpectedCall: No matching expectations found for call:

at <doctest default[2]>:1
-------------------------
Called:
  mock(3.14)
Expected (any of):
  mock(Type(int) & Func(POSITIVE_ONLY))
```

Using matchers in structured data

You are not only limited to use matchers in `expect_call()` arguments and keyword arguments. You can also use it inside larger structures, like dicts. That is a side effect of the fact that matchers are implemented by customizing standard Python's `__eq__()` operator, which is called every time you compare one object with another. Here's an example:

```
from mockify.mock import Mock
from mockify.actions import Return
from mockify.matchers import Type, List

mock = Mock('mock')
mock.expect_call({
    'action': Type(str),
    'params': List(Type(int), min_length=2),
}).will_repeatedly(Return(True))
```

We've recorded expectation that `mock()` will be called with dict containing *action* key that is a string, and *params* key that is a list of integers containing at least 2 elements. Here's how it works:

```
>>> mock({'action': 'sum', 'params': [2, 3]})
True
>>> mock({'action': 'sum', 'params': [2, 3, 4]})
True
>>> mock({'action': 'sum', 'params': [2]})
Traceback (most recent call last):
    ...
mockify.exc.UnexpectedCall: No matching expectations found for call:
```

(continues on next page)

(continued from previous page)

```
at <doctest default[2]>:1
-------------------------
Called:
  mock({'action': 'sum', 'params': [2]})
Expected (any of):
  mock({'action': Type(str), 'params': List(Type(int), min_length=2)})
```

In the last example we got *mockify.exc.UnexpectedCall* exception because our *params* key got only one argument, while it was expected at least 2 to be given. There is no limit of how deep you can go with your structures.

Using matchers in custom objects

You can also use matchers with your objects. Like in this example:

```
from collections import namedtuple

from mockify.mock import Mock
from mockify.matchers import Type

Vec2 = namedtuple('Vec2', 'x, y')  # (1)

Float = Type(float)  # (2)

canvas = Mock('canvas')
canvas.draw_line.expect_call(
    Vec2(Float, Float), Vec2(Float, Float)).\
    will_repeatedly(Return(True))  # (3)
```

We've created a vector object (1), then an alias to `Type(float)` (2) for a more readable expectation composing (an _ alias for *mockify.matchers.Any* is created in same way). Finally, we've created *canvas* mock and mocked `draw_line()` method, taking start and end point arguments in form of 2-dimensional vectors. And here's how it works:

```
>>> canvas.draw_line(Vec2(0.0, 0.0), Vec2(5.0, 5.0))
True
>>> canvas.draw_line(Vec2(0, 0), Vec2(5, 5))
Traceback (most recent call last):
    ...
mockify.exc.UnexpectedCall: No matching expectations found for call:

at <doctest default[1]>:1
-------------------------
Called:
  canvas.draw_line(Vec2(x=0, y=0), Vec2(x=5, y=5))
Expected (any of):
  canvas.draw_line(Vec2(x=Type(float), y=Type(float)), Vec2(x=Type(float),␣
↪y=Type(float)))
```

Using matchers out of Mockify library

Matchers are pretty generic tool that you can also use outside of Mockify - just for assertion checking. For example, if you have a code that creates some records with auto increment ID you can use a matcher from Mockify to check if that ID matches some expected criteria - especially when exact value is hard to guess:

Here's an example code:

```
import itertools

_next_id = itertools.count(1)  # This is private

def make_product(name, description):
    return {
        'id': next(_next_id),
        'name': name,
        'description': description
    }
```

And here's an example test:

```
from mockify.matchers import Type, Func

def test_make_product():
    product = make_product('foo', 'foo desc')
    assert product == {
        'id': Type(int) & Func(lambda x: x > 0, 'GREATER_THAN_ZERO'),
        'name' : 'foo',
        'description': 'foo desc',
    }
```

2.3.7 Patching imported modules

New in version 0.6.

With Mockify you can easily substitute imported module with a mocked version. Consider following code:

```
import os

def iter_dirs(path):
    for name in os.listdir(path):
        fullname = os.path.join(path, name)
        if os.path.isdir(fullname):
            yield fullname
```

That function generates full paths to all direct children directories of given *path*. And it uses `os` to make some file system operations. To test that function without refactoring it you will have to **patch** some methods of `os` module. And here's how this can be done in Mockify:

```
from mockify import satisfied, patched
from mockify.mock import Mock
from mockify.actions import Return

def test_iter_dirs():
    os = Mock('os')  # (1)
    os.listdir.expect_call('/tmp').will_once(Return(['foo', 'bar', 'baz.txt']))  # (2)
    os.path.isdir.expect_call('/tmp/foo').will_once(Return(True))  # (3)
    os.path.isdir.expect_call('/tmp/bar').will_once(Return(True))
    os.path.isdir.expect_call('/tmp/baz.txt').will_once(Return(False))

    with patched(os):  # (4)
        with satisfied(os):  # (5)
            assert list(iter_dirs('/tmp')) == ['/tmp/foo', '/tmp/bar']  # (6)
```

And here's what's going on in presented test:

- We've created *os* mock (1) for mocking **os.listdir()** (2) and **os.path.isdir()** (3) methods,
- Then we've used *mockify.patched()* context manager (4) that does the whole magic of substituting modules matching full names of mocks with expectations recorded (which are `'os.listdir'` and `'os.path.isdir'` in our case) with corresponding mock objects
- Finally, we've used *mockify.satisfied()* context manager (5) to ensure that all expectations are satisfied, and ran tested function (6) checking it's result.

Note that we did not mock `os.path.join()` - that will be used from `os` module.

2.3.8 Recording ordered expectations

New in version 0.6.

Mockify provides a mechanism for recording **ordered expectation**, i.e. expectations that can only be resolved in their declaration order. That may be crucial if you need to provide additional level of testing for parts of code that **must not** call given interfaces in any order. Consider this:

```
class InterfaceCaller:

    def __init__(self, first, second):
        self._first = first
        self._second = second

    def run(self):
        # A lot of complex processing goes in here...
        self._first.inform()  # (1)
        # ...and some in here.
        self._second.inform()  # (2)
```

We have a class that depends on two interfaces: *first* and *second*. That class has a `run()` method in which some complex processing takes place. The result of processing is a **call** to both of these two interfaces, but **the order does matter**; calling *second* before *first* is considered a **bug** which should be discovered by tests. And here is our test:

```
from mockify import satisfied
from mockify.mock import Mock

def test_interface_caller():
    first = Mock('first')
    second = Mock('second')

    first.inform.expect_call()
    second.inform.expect_call()

    caller = InterfaceCaller(first, second)
    with satisfied(first, second):
        caller.run()
```

And of course, the test passes. But will it pass if we change the order of calls in class we are testing? Of course it **will**, because by default the order of declared expectations is irrelevant (for as long as return values does not come into play). And here comes **ordered expectations**:

```
from mockify import satisfied, ordered
from mockify.mock import MockFactory

```

(continues on next page)

(continued from previous page)

```
def test_interface_caller():
    factory = MockFactory()  # (1)
    first = factory.mock('first')
    second = factory.mock('second')

    first.inform.expect_call()
    second.inform.expect_call()

    caller = InterfaceCaller(first, second)
    with satisfied(factory):
        with ordered(factory):  # (2)
            caller.run()
```

In the test above we've used mock factory (1), because ordered expectations require all checked mocks to operate on a common session. The main difference however is use of *mockify.ordered()* context manager (2) which ensures that given mocks (mocks created by *factory* in this case) will be called **in their declaration order**. And since we've changed the order in tested code, the test will no longer pass and *mockify.exc.UnexpectedCallOrder* assertion will be raised:

```
>>> test_interface_caller()
Traceback (most recent call last):
    ...
mockify.exc.UnexpectedCallOrder: Another mock is expected to be called:

at <doctest default[0]>:9
-------------------------
Called:
  second.inform()
Expected:
  first.inform()
```

And that exception tells us that we've called `second.inform()`, while it was expected to call `first.inform()` earlier.

2.4 Tips & tricks

2.4.1 Mocking functions with output parameters

Sometimes you will need to mock calls to interfaces that will force you to create some kind of object which later is used as an argument to that call. Value of that object is not constant, it is different on every run, so you will not be able to record adequate expectation using fixed value. Moreover, that object is changed by call to mocked interface method. How to mock that out?

This kind of problem exists in following simple class:

```
import io


class RemoteFileStorage:

    def __init__(self, bucket):
        self._bucket = bucket
```

(continues on next page)

(continued from previous page)

```
    def read(self, name):
        buffer = io.BytesIO()  # (1)
        self._bucket.download('bucket-name', "uploads/{}".format(name), buffer)  # (2)
        return buffer.getvalue()
```

That class is kind of a facade on top of some cloud service for accessing files that were previously uploaded by another part of the application. To download the file we need to create a buffer (1) which is later passed to `bucket.download()` method (2). In production, that method downloads a file into a buffer, but how to actually mock that in test?

Here's a solution:

```
from mockify import satisfied
from mockify.mock import Mock
from mockify.actions import Invoke
from mockify.matchers import _


def download(payload, bucket, key, fd):  # (1)
    fd.write(payload)


def test_reading_file_using_remote_storage():
    bucket = Mock('bucket')

    bucket.download.\
        expect_call('bucket-name', 'uploads/foo.txt', _).\
        will_once(Invoke(download, b'spam'))  # (2)

    storage = RemoteFileStorage(bucket)

    with satisfied(bucket):
        assert storage.read('foo.txt') == b'spam'  # (3)
```

And here's an explanation:

- We've implemented `download()` function - a minimal and dummy implementation of `bucket.download()` method. Our function simply writes given payload to file descriptor created in tested class and passed as a last argument.
- We've recorded an expectation that `bucket.download()` will be called once with three args, having last argument wildcarded using *`mockify.matchers.Any`* matcher. Therefore, buffer object passed to a mock will match that expectation.
- We've recorded single *`mockify.actions.Invoke`* action to execute function created in (1), with `b'spam'` bytes object bound as first argument (that's why `download()` function accepts one more argument). With this approach we can force `download()` to write different bytes - depending on out test.
- Finally, we've used assertion at (3) to check if tested method returns "downloaded" bytes.

2.5 API Reference

2.5.1 mockify - Library core

Library core module.

class mockify.**Call**(*_name_*, **args*, ***kwargs*)
An object representing mock call.

Instances of this class are created when expectations are recorded or when mock is called. Call objects are comparable. Two call objects are equal if and only if:

- mock names are the same,
- args are the same,
- and keyword args are the same.

Parameters **_name_** – The name of a mock

name
The name of a mock.

args
Positional args mock was called with or is expected to be called with.

kwargs
Keyword args mock was called with or is expected to be called with.

location
Information of place in test or tested code where this call object was created.

New in version 0.6.

Return type *LocationInfo*

class mockify.**LocationInfo**(*filename*, *lineno*)
A placeholder for file name and line number obtained from the stack.

Used by *Call* objects to get their location in the code. That information is later used in assertion messages.

New in version 0.6.

Parameters

- **filename** – Name of the file
- **lineno** – Line number in given file

filename
File name from stack.

lineno
Line number form stack.

classmethod get_external()
Factory method for creating instances of this class.

It extracts stack and finds (in reversed order) first frame that is **outside** of the Mockify library. Thanks to this all mock calls or expectation recordings will point to test function or tested code that uses Mockify, not to Mockify's internals.

Return type *LocationInfo*

class mockify.**Session**
A class providing core logic of connecting mock calls with recorded expectations.

Sessions are created for each mock automatically, or can be created explicitly and then shared across multiple mocks. While mock classes can be seen as som kind of frontends that mimic behavior of various Python constructs, session instances are some kind of backends that receive *mockify.Call* instances created by mocks during either mock call, or expectation recording.

Changed in version 0.6: Previously this was named **Registry**.

config
A dictionary-like object for configuring sessions.

Following options are currently available:

'expectation_class' Can be used to override expectation class used when expectations are recorded.

By default, this is *mockify.Expectation*, and there is a requirement that custom class must inherit from original one.

'uninterested_call_strategy' Used to set a way of processing so called **unexpected calls**, i.e. calls to mocks that has no expectations recorded. Following values are supported:

'fail' This is default option.

When mock is called unexpectedly, *mockify.exc.UninterestedCall* exception is raised and test is terminated.

'warn' Instead of raising exception, *mockify.exc.UninterestedCallWarning* warning is issued, and test continues.

'ignore' Unexpected calls are silently ignored.

__call__(*actual_call*)
Trigger expectation matching *actual_call* received from mock being called.

This method is called on every mock call and basically all actual call processing takes place here. Values returned or exceptions raised by this method are also returned or raised by mock.

Parameters **actual_call** – Instance of *mockify.Call* class created by calling mock.

expectations()
An iterator over all expectations recorded in this session.

Yields *mockify.Expectation* instances.

expect_call(*expected_call*)
Called by mock when expectation is recorded on it.

This method creates expectation object, adds it to the list of expectations, and returns.

Return type *mockify.Expectation*

Parameters **expected_call** – Instance of *mockify.Call* created by mock when **expect_call()** was called on it.

Represents parameters the mock is expected to be called with.

assert_satisfied()
Check if all registered expectations are satisfied.

This works exactly the same as *mockify.assert_satisfied()*, but for given session only. Can be used as a replacement for any other checks if one global session object is used.

enable_ordered(*names*)
Mark expectations matching given mock *names* as **ordered**, so they will have to be resolved in their declaration order.

This is used internally by *mockify.ordered()*.

disable_ordered()
Called by *mockify.ordered()* when processing of ordered expectations is done.

Moves any remaining expectations back to the **unordered** storage, so they will be later displayed as unsatisfied.

class mockify.**Expectation**(*expected_call*)

An class representing single expectation.

Instances of this class are created and returned by factory **expect_call()** method you will use to record expectations on your mocks:

```
>>> from mockify.mock import Mock
>>> mock = Mock('mock')
>>> mock.expect_call(1, 2)
<mockify.Expectation: mock(1, 2)>
```

Parameters **expected_call** – Instance of *Call* class containing parameters passed to **expect_call()** factory method that created this expectation object.

__call__(*actual_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

is_satisfied()

Check if this expectation is satisfied.

Expectation object is satisfied if and only if:

- total number of calls is not exceeding expected number of calls,
- all actions (if any were recorded) are **consumed**.

Return type bool

times(*cardinality*)

Set expected number or range of call counts.

Following values are possible:

- integer number (for setting expected call count to fixed value),
- instance of *mockify.cardinality.ExpectedCallCount* (for setting expected call count to **ranged** value).

See *Setting expected call count* tutorial section for more details.

will_once(*action*)

Append next action to be executed when this expectation object receives a call.

Once this method is called, it returns special proxy object that you can use to mutate this expectation even further by calling one of given methods on that proxy:

- **will_once()** (this one again),
- **will_repeatedly()** (see *will_repeatedly()*).

Thanks to that you can record so called **action chains** (see *Recording actions* for more details).

This method can be called with any action object from `mockify.actions` as an argument.

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

Unlike single actions, recorded with `will_once()`, repeated actions are by default executed any number of times, including zero (see *Repeated actions* for more details).

Once this method is called, it returns a proxy object you can use to adjust repeated action even more by calling one of following methods:

- **times**(), used to record repeated action call count limits (see `times()`).

This method accepts actions defined in `mockify.actions` module.

expected_call
Returns *expected_call* parameter passed during construction.

This is used when this expectation is compared with `Call` object representing **actual call**, to find expectations matching that call.

Return type *Call*

actual_call_count
Number of matching calls this expectation object received so far.

This is relative value; if one action expires and another one is started to be executed, then the counter starts counting from 0 again. Thanks to this you'll receive information about actual action execution count. If your expectation does not use `will_once()` or `will_repeatedly()`, then this counter will return total number of calls.

New in version 0.6.

expected_call_count
Return object representing expected number of mock calls.

Like `actual_call_count`, this varies depending on internal expectation object state.

Return type *mockify.cardinality.ExpectedCallCount*

action
Return action to be executed when this expectation receives another call or `None` if there are no (more) actions.

Return type *mockify.actions.Action*

mockify.**assert_satisfied**(**mocks*)
Check if all given mocks are **satisfied**.

This function collects all expectations from given mock for which `Expectation.is_satisfied()` evaluates to `False`. Finally, if at least one **unsatisfied** expectation is found, this method raises `mockify.exc.Unsatisfied` exception.

See recording-and-validating-expectations tutorial for more details.

mockify.**ordered**(**mocks*)
Context manager that checks if expectations in wrapped scope are consumed in same order as they were defined.

This context manager will raise `mockify.exc.UnexpectedCallOrder` assertion on first found mock that is executed out of specified order.

See *Recording ordered expectations* for more details.

mockify.**satisfied**(**mocks*)
Context manager wrapper for *assert_satisfied()*.

mockify.**patched**(**mocks*)
Context manager that replaces imported objects and functions with their mocks using mock name as a name of patched module.

It will patch only functions or objects that have expectations recorded, so all needed expectations will have to be recorded before this context manager is used.

See *Patching imported modules* for more details.

2.5.2 mockify.mock - Classes for creating and inspecting mocks

class mockify.mock.**MockInfo**(*mock*)
An object used to inspect given target mock.

This class provides a sort of public interface on top of underlying *Mock* instance, that due to its specific features has no methods or properties publicly available.

Parameters **mock** – Instance of *Mock* object to be inspected

mock
Reference to target mock object.

name
Name of target mock.

session
Instance of *mockify.Session* assigned to given target mock.

expectations()
An iterator over all *mockify.Expectation* objects recorded for target mock.

children()
An iterator over target mock's direct children.

It yields *MockInfo* object for each target mock's children.

walk()
Recursively iterates in depth-first order over all target mock's children and yields *MockInfo* object for each found child.

It always yields *self* as first element.

class mockify.mock.**MockFactory**(*name=None*, *session=None*, *mock_class=None*)
A factory class used to create groups of related mocks.

This class allows to create mocks using class given by *mock_class* ensuring that:

- names of created mocks are **unique**,
- all mocks share one common session object.

Instances of this class keep track of created mocks. Moreover, functions that would accept *Mock* instances will also accept *MockFactory* instances, so you can later f.e. check if all created mocks are satisfied using just a factory object. That makes it easy to manage multiple mocks in large test suites.

See *Managing multiple mocks* for more details.

New in version 0.6.

Parameters

- **name** – This is optional.

  Name of this factory to be used as a common prefix for all created mocks and nested factories.

- **session** – Instance of *mockify.Session* to be used.

  If not given, a default session will be created and shared across all mocks created by this factory.

- **mock_class** – The class that will be used by this factory to create mocks.

  By default it will use *Mock* class.

mock(*name*)
Create and return mock of given *name*.

This method will raise `TypeError` if *name* is already used by either mock or child factory.

factory(*name*)
Create and return child factory.

Child factory will use session from its parent, and will prefix all mocks and grandchild factories with given *name*.

This method will raise `TypeError` if *name* is already used by either mock or child factory.

Return type *MockFactory*

class mockify.mock.**Mock**(*name*, *session=None*)
All-in-one mocking utility.

This class is used to:

- create mocks of functions,
- create mocks of objects with methods, setters and getters,
- create mocks of modules,
- create ad-hoc data objects.

No matter what you will be mocking, for all cases creating mock objects is always the same - by giving it a *name* and optionally *session*. Mock objects automatically create attributes on demand, and that attributes form some kind of **nested** or **child** mocks.

To record expectations, you have to call **expect_call()** method on one of that attributes, or on mock object itself (for function mocks). Then you pass mock object to unit under test. Finally, you will need *mockify.assert_satisfied()* function or *mockify.satisfied()* context manager to check if the mock is satisfied.

Here's an example:

```
from mockify import satisfied
from mockify.mock import Mock

def caller(func, a, b):
    func(a + b)

def test_caller():
    func = Mock('func')
    func.expect_call(5)
    with satisfied(func):
        caller(func, 2, 3)
```

See *Creating mocks and recording expectations* for more details.

New in version 0.6.

2.5.3 mockify.actions - Classes for recording side effects

class mockify.actions.**Action**
Bases: abc.ABC

Abstract base class for actions.

This is common base class for all actions defined in this module. Custom actions should also inherit from this one.

New in version 0.6.

__call__(*actual_call*)
Action body.

It receives actual call object and returns action result based on that call object. This method may also raise exceptions if that is functionality of the action being implemented.

Parameters **actual_call** – Instance of *mockify.Call* containing params of actual call being made

format_params(**args*, ***kwargs*)
Used to calculate **str()** and **repr()** for this action.

This method should be overloaded in subclass as a no argument method, and then call **super().format_params(...)** with args and kwargs you want to be included in **str()** and **repr()** methods.

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

Forces mock to return *value* when called.

For example:

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

__call__(*actual_call*)
Action body.

It receives actual call object and returns action result based on that call object. This method may also raise exceptions if that is functionality of the action being implemented.

Parameters **actual_call** – Instance of *mockify.Call* containing params of actual call being made

format_params()
Used to calculate **str()** and **repr()** for this action.

This method should be overloaded in subclass as a no argument method, and then call **super().format_params(...)** with args and kwargs you want to be included in **str()** and **repr()** methods.

class mockify.actions.**Iterate**(*iterable*)

Bases: *mockify.actions.Action*

Similar to *Return*, but returns an iterator to given *iterable*.

For example:

```
>>> from mockify.mock import Mock
>>> from mockify.actions import Iterate
>>> mock = Mock('mock')
>>> mock.expect_call().will_once(Iterate('foo'))
<mockify.Expectation: mock()>
>>> next(mock())
'f'
```

New in version 0.6.

__call__(*actual_call*)

Action body.

It receives actual call object and returns action result based on that call object. This method may also raise exceptions if that is functionality of the action being implemented.

Parameters **actual_call** – Instance of *mockify.Call* containing params of actual call being made

format_params()

Used to calculate **str()** and **repr()** for this action.

This method should be overloaded in subclass as a no argument method, and then call **super().format_params(...)** with args and kwargs you want to be included in **str()** and **repr()** methods.

class mockify.actions.**Raise**(*exc*)

Bases: *mockify.actions.Action*

Forces mock to raise *exc* when called.

For example:

```
>>> from mockify.mock import Mock
>>> from mockify.actions import Raise
>>> mock = Mock('mock')
>>> mock.expect_call().will_once(Raise(ValueError('invalid value')))
<mockify.Expectation: mock()>
>>> mock()
Traceback (most recent call last):
    ...
ValueError: invalid value
```

__call__(*actual_call*)

Action body.

It receives actual call object and returns action result based on that call object. This method may also raise exceptions if that is functionality of the action being implemented.

Parameters **actual_call** – Instance of *mockify.Call* containing params of actual call being made

format_params()

Used to calculate **str()** and **repr()** for this action.

This method should be overloaded in subclass as a no argument method, and then call **super().format_params(...)** with args and kwargs you want to be included in **str()** and **repr()** methods.

class mockify.actions.**Invoke**(*func*, **args*, ***kwargs*)

Bases: *mockify.actions.Action*

Forces mock to invoke *func* when called.

When *func* is called, it is called with all bound arguments plus all arguments mock was called with. Value that mock returns is the one *func* returns. Use this action when more sophisticated checks have to be done when mock gets called or when your mock must operate on some **output parameter**.

See *Mocking functions with output parameters* for more details.

Here's an example using one of built-in functions as a *func*:

```
>>> from mockify.mock import Mock
>>> from mockify.actions import Invoke
>>> mock = Mock('mock')
>>> mock.expect_call([1, 2, 3]).will_once(Invoke(sum))
<mockify.Expectation: mock([1, 2, 3])>
>>> mock([1, 2, 3])
6
```

Changed in version 0.6: Now this action allows binding args to function being called.

Parameters

- **func** – Function to be executed
- **args** – Additional positional args to be bound to `func`.
- **kwargs** – Additional named args to be bound to `func`.

__call__(*actual_call*)

Action body.

It receives actual call object and returns action result based on that call object. This method may also raise exceptions if that is functionality of the action being implemented.

Parameters **actual_call** – Instance of *mockify.Call* containing params of actual call being made

format_params()

Used to calculate **str()** and **repr()** for this action.

This method should be overloaded in subclass as a no argument method, and then call **super().format_params(...)** with args and kwargs you want to be included in **str()** and **repr()** methods.

2.5.4 mockify.cardinality - Classes for setting expected call cardinality

class mockify.cardinality.**ActualCallCount**(*initial_value*)

Bases: object

Proxy class that is used to calculate actual mock calls.

Provides all needed arithmetic operators and a logic of rendering actual call message that is used in assertion messages.

Here's an example:

```
>>> from mockify.cardinality import ActualCallCount
>>> str(ActualCallCount(0))
'never called'
```

(continues on next page)

(continued from previous page)

```
>>> str(ActualCallCount(1))
'called once'
```

New in version 0.6.

__str__()
Return str(self).

class mockify.cardinality.**ExpectedCallCount**
Bases: abc.ABC

Abstract base class for classes used to set expected call count on mock objects.

New in version 0.6.

__str__()
Format message to be used in assertion reports.

This message must state how many times the mock was expected to be called and will only be evaluated if expectation is not satisfied.

match(*actual_call_count*)
Check if *actual_call_count* matches expected call count.

adjust_minimal(*minimal*)
Make a new cardinality object based on its current state and given *minimal*.

format_params(**args*, ***kwargs*)
Format params to be used in **repr()**.

This method must be overloaded without params, and call **super().format_params(...)** with args and kwargs you want to include in **repr()**.

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

Used to set expected call count to fixed *expected* value.

Expectations marked with this cardinality object will have to be called **exactly** *expected* number of times to be satisfied.

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

__str__()
Format message to be used in assertion reports.

This message must state how many times the mock was expected to be called and will only be evaluated if expectation is not satisfied.

match(*actual_call_count*)
Check if *actual_call_count* matches expected call count.

adjust_minimal(*minimal*)
Make a new cardinality object based on its current state and given *minimal*.

format_params()
Format params to be used in **repr()**.

This method must be overloaded without params, and call **super().format_params(...)** with args and kwargs you want to include in **repr()**.

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

Used to set expected call count to given *minimal* value.

Expectation will be satisfied if called not less times that given *minimal*.

__str__()
Format message to be used in assertion reports.

This message must state how many times the mock was expected to be called and will only be evaluated if expectation is not satisfied.

match(*actual_call_count*)
Check if *actual_call_count* matches expected call count.

adjust_minimal(*minimal*)
Make a new cardinality object based on its current state and given *minimal*.

format_params()
Format params to be used in **repr()**.

This method must be overloaded without params, and call **super().format_params(...)** with args and kwargs you want to include in **repr()**.

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

Used to set expected call count to given *maximal* value.

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

__str__()
Format message to be used in assertion reports.

This message must state how many times the mock was expected to be called and will only be evaluated if expectation is not satisfied.

match(*actual_call_count*)
Check if *actual_call_count* matches expected call count.

adjust_minimal(*minimal*)
Make a new cardinality object based on its current state and given *minimal*.

format_params()
Format params to be used in **repr()**.

This method must be overloaded without params, and call **super().format_params(...)** with args and kwargs you want to include in **repr()**.

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

Used to set a range of expected call counts between *minimal* and *maximal*, both included.

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*.

__str__()
Format message to be used in assertion reports.

This message must state how many times the mock was expected to be called and will only be evaluated if expectation is not satisfied.

match(*actual_call_count*)
Check if *actual_call_count* matches expected call count.

adjust_minimal(*minimal*)
Make a new cardinality object based on its current state and given *minimal*.

format_params()
Format params to be used in **repr()**.

This method must be overloaded without params, and call **super().format_params(...)** with args and kwargs you want to include in **repr()**.

2.5.5 mockify.matchers - Classes for wildcarding expected arguments

class mockify.matchers.**Matcher**
Bases: abc.ABC

Abstract base class for matchers.

Changed in version 0.6: Now this inherits from abc.ABC

__eq__(*other*)
Check if *other* can be accepted by this matcher.

format_repr(**args*, ***kwargs*)
Return matcher's textual representation.

Typical use case of this class is to override it in child class without parameters and then call super giving it args you want to include in repr. Like in this example:

```
def format_repr(self):
    return super().format_repr(self._first_arg, self._second_arg, kwd=self._
↪kwd_arg)
```

__repr__()
Return repr(self).

class mockify.matchers.**AnyOf**(**values*)
Bases: *mockify.matchers.Matcher*

Matches any value from given list of *values*.

You can also use matchers in *values*.

New in version 0.6.

__eq__(*other*)
Check if *other* can be accepted by this matcher.

format_repr()
Return matcher's textual representation.

Typical use case of this class is to override it in child class without parameters and then call super giving it args you want to include in repr. Like in this example:

```
def format_repr(self):
    return super().format_repr(self._first_arg, self._second_arg, kwd=self._
↪kwd_arg)
```

class mockify.matchers.**AllOf**(**values*)
Bases: *mockify.matchers.Matcher*

Matches if and only if received value is equal to all given *values*.

You can also use matchers in *values*.

New in version 0.6.

__eq__(*other*)
Check if *other* can be accepted by this matcher.

format_repr()
Return matcher's textual representation.

Typical use case of this class is to override it in child class without parameters and then call super giving it args you want to include in repr. Like in this example:

```
def format_repr(self):
    return super().format_repr(self._first_arg, self._second_arg, kwd=self._
↪kwd_arg)
```

class mockify.matchers.**Any**
Bases: *mockify.matchers.Matcher*

Matches any value.

This can be used as a wildcard, when you care about number of arguments in your expectation, not their values or types. This can also be imported as underscore:

```
from mockify.matchers import _
```

__eq__(*other*)
Check if *other* can be accepted by this matcher.

format_repr()
Return matcher's textual representation.

Typical use case of this class is to override it in child class without parameters and then call super giving it args you want to include in repr. Like in this example:

```
def format_repr(self):
    return super().format_repr(self._first_arg, self._second_arg, kwd=self._
↪kwd_arg)
```

class mockify.matchers.**Type**(**types*)
Bases: *mockify.matchers.Matcher*

Matches any value that is instance of one of given *types*.

This is useful to record expectations where we do not care about expected value, but we do care about expected value type.

New in version 0.6.

__eq__(*other*)
Check if *other* can be accepted by this matcher.

format_repr()
Return matcher's textual representation.

Typical use case of this class is to override it in child class without parameters and then call super giving it args you want to include in repr. Like in this example:

```
def format_repr(self):
    return super().format_repr(self._first_arg, self._second_arg, kwd=self._
↪kwd_arg)
```

class mockify.matchers.**Regex**(*pattern*, *name=None*)

Bases: *mockify.matchers.Matcher*

Matches value if it is a string that matches given regular expression *pattern*.

Parameters

- **pattern** – Regular expression pattern
- **name** – Optional name for given pattern.

  If given, then name will be used in text representation of this matcher. This can be very handy, especially when regular expression is complex and hard to read. Example:

```
>>> r = Regex(r'^[a-z]+$', 'LOWER_ASCII')
>>> repr(r)
'Regex(LOWER_ASCII)'
```

New in version 0.6.

__eq__(*other*)

Check if *other* can be accepted by this matcher.

format_repr()

Return matcher's textual representation.

Typical use case of this class is to override it in child class without parameters and then call super giving it args you want to include in repr. Like in this example:

```
def format_repr(self):
    return super().format_repr(self._first_arg, self._second_arg, kwd=self._
↪kwd_arg)
```

class mockify.matchers.**List**(*matcher*, *min_length=None*, *max_length=None*)

Bases: *mockify.matchers.Matcher*

Matches value if it is a list of values matching *matcher*.

Parameters

- **matcher** – A matcher that every value in the list is expected to match.

  Use *Any* matcher if you want to match list containing any values.
- **min_length** – Minimal accepted list length
- **max_length** – Maximal accepted list length

New in version 0.6.

__eq__(*other*)

Check if *other* can be accepted by this matcher.

format_repr()

Return matcher's textual representation.

Typical use case of this class is to override it in child class without parameters and then call super giving it args you want to include in repr. Like in this example:

```
def format_repr(self):
    return super().format_repr(self._first_arg, self._second_arg, kwd=self._
↪kwd_arg)
```

class mockify.matchers.**Object**(***kwargs*)

Bases: *mockify.matchers.Matcher*

Matches value if it is an object with attributes equal to names and values given via keyword args.

This matcher creates ad-hoc object using provided keyword args. These args are then used to compare with value's attributes of same name. All attributes must match for this matcher to accept value.

Here's an example:

```
from collections import namedtuple

from mockify import satisfied
from mockify.mock import Mock
from mockify.matchers import Object

CallArg = namedtuple('CallArg', 'foo, bar')

mock = Mock('mock')
mock.expect_call(Object(foo=1, bar=2))

with satisfied(mock):
    mock(CallArg(1, 2))
```

New in version 0.6.5.

Parameters ****kwargs** – Arguments to compare value with

__eq__(*other*)

Check if *other* can be accepted by this matcher.

format_repr()

Return matcher's textual representation.

Typical use case of this class is to override it in child class without parameters and then call super giving it args you want to include in repr. Like in this example:

```
def format_repr(self):
    return super().format_repr(self._first_arg, self._second_arg, kwd=self._
↪kwd_arg)
```

class mockify.matchers.**Func**(*func*, *name=None*)

Bases: *mockify.matchers.Matcher*

Matches value if *func* returns `True` for that value.

This is the most generic matcher as you can use your own match function if needed.

Parameters

- **func** – Function to be used to calculate match.
- **name** – Optional name for this matcher.

  This can be used to set a name used to format matcher's text representation for assertion errors. Here's a simple example:

```
>>> f = Func(lambda x: x > 0, 'POSITIVE_ONLY')
>>> repr(f)
'Func(POSITIVE_ONLY)'
```

New in version 0.6.

__eq__(*other*)
Check if *other* can be accepted by this matcher.

format_repr()
Return matcher's textual representation.

Typical use case of this class is to override it in child class without parameters and then call super giving it args you want to include in repr. Like in this example:

```
def format_repr(self):
    return super().format_repr(self._first_arg, self._second_arg, kwd=self._
↪kwd_arg)
```

2.5.6 mockify.exc - Library exceptions

exception mockify.exc.**MockifyWarning**
Bases: Warning

Common base class for Mockify warnings.

New in version 0.6.

exception mockify.exc.**UninterestedCallWarning**
Bases: *mockify.exc.MockifyWarning*

This warning is used to inform about uninterested call being made.

It is only used when uninterested call strategy is changed in mocking session. See *mockify.Session* for more details.

New in version 0.6.

exception mockify.exc.**MockifyError**(***kwargs*)
Bases: Exception

Common base class for all Mockify exceptions.

New in version 0.6.

exception mockify.exc.**MockifyAssertion**(***kwargs*)
Bases: *mockify.exc.MockifyError*, AssertionError

Common base class for all Mockify assertion errors.

With this exception it will be easy to re-raise Mockify-specific assertion exceptions for example during debugging.

New in version 0.6.

exception mockify.exc.**UnexpectedCall**(*actual_call*, *expected_calls*)
Bases: *mockify.exc.MockifyAssertion*

Raised when mock was called with parameters that couldn't been matched to any of existing expectations.

This exception was added for easier debugging of failing tests; unlike *UninterestedCall* exception, this one signals that there are expectations set for mock that was called.

For example, we have expectation defined like this:

```
from mockify.mock import Mock

mock = Mock('mock')
mock.expect_call(1, 2)
```

And if the mock is now called f.e. without params, this exception will be raised:

```
>>> mock()
Traceback (most recent call last):
    ...
mockify.exc.UnexpectedCall: No matching expectations found for call:

at <doctest default[0]>:1
-------------------------
Called:
  mock()
Expected (any of):
  mock(1, 2)
```

New in version 0.6.

Parameters

- **actual_call** – Instance of *mockify.Call* representing parameters of call that was made
- **expected_calls** – List of *mockify.Call* instances, each representing expected parameters of single expectation

exception mockify.exc.**UnexpectedCallOrder**(*actual_call*, *expected_call*)

Bases: *mockify.exc.MockifyAssertion*

Raised when mock was called but another one is expected to be called before.

This can only be raised if you use ordered expectations with *mockify.ordered()* context manager.

See *Recording ordered expectations* for more details.

New in version 0.6.

Parameters

- **actual_call** – The call that was made
- **expected_call** – The call that is expected to be made

exception mockify.exc.**UninterestedCall**(*actual_call*)

Bases: *mockify.exc.MockifyAssertion*

Raised when call is made to a mock that has no expectations set.

This exception can be disabled by changing unexpected call strategy using *mockify.Session.config* attribute (however, you will have to manually create and share session object to change that).

Parameters **actual_call** – The call that was made

exception mockify.exc.**OversaturatedCall**(*actual_call*, *oversaturated_expectation*)

Bases: *mockify.exc.MockifyAssertion*

Raised when mock with actions recorded using *mockify.Expectation.will_once()* was called more times than expected and has all recorded actions already consumed.

This exception can be avoided if you record repeated action to the end of expected action chain (using *mockify.Expectation.will_repeatedly()*). However, it was added for a reason. For example, if your mock returns value of incorrect type (the default one), you'll result in production code errors instead of mock errors. And that can possibly be harder to debug.

Parameters

- **actual_call** – The call that was made
- **oversaturated_expectation** – The expectation that was oversaturated

exception mockify.exc.**Unsatisfied**(*unsatisfied_expectations*)

Bases: *mockify.exc.MockifyAssertion*

Raised when unsatisfied expectations are present.

This can only be raised by either *mockify.satisfied()* *mockify.assert_satisfied()* or mockify.Session.done(). You'll not get this exception when mock is called.

Parameters **unsatisfied_expectations** – List of all unsatisfied expectations found

unsatisfied_expectations

List of unsatisfied expectations.

New in version 0.6: Previously it was called expectations.

2.6 Changelog

2.6.1 0.7.1

- Fix *mockify.matchers.Object* matcher to be inequal to reference object if reference object does not have one or more properties listed in matcher

2.6.2 0.7.0

- An alias to 0.6.5 to fix versioning (new feature was introduced, and wrong version part was increased by mistake)

2.6.3 0.6.5

- Added *mockify.matchers.Object* matcher

2.6.4 0.6.4

- Deprecated code was removed
- Improved documentation
- Documentation is now tested by Sphinx
- Class **Registry** was renamed to *mockify.Session*
- All classes for making mocks were replaced by single generic *mockify.mock.Mock* class, supported by *mockify.mock.MockFactory* class
- New actions introduced (see *mockify.actions*)
- New matchers introduced (see *mockify.matchers*)

- New assertion errors introduced and improved exception hierarchy (see *mockify.exc*)
- Better reporting of expectation location in assertion messages
- Can now define ordered expectations with *mockify.ordered()* context manager
- Can now patch imports using *mockify.patched()* context manager
- CI workflow updated + added testing against various Python versions (3.x for now)
- Many other improvements in the code and the tests

2.6.5 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*

2.6.6 0.4.0

- Added strategies for dealing with unexpected calls

2.6.7 0.3.1

- Added frontend for mocking Python objects

2.6.8 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

2.6.9 0.1.12

- First release published to PyPI

2.7 License

Python Module Index

m

Index

Symbols

A

B

C

D

E

W