flexmock Documentation

Release 0.10.2

Slavek Kabrda, Herman Sheremetyev

November 08, 2016

Contents

Authors Slavek Kabrda, Herman Sheremetyev

Version 0.10.2

Homepage flexmock Homepage

Contribute flexmock on Github

Download http://pypi.python.org/pypi/flexmock

License FreeBSD-style License

Issue tracker Issue Tracker

flexmock is a testing library for Python.

Its API is inspired by a Ruby library of the same name. However, it is not a goal of Python flexmock to be a clone of the Ruby version. Instead, the focus is on providing full support for testing Python programs and making the creation of fake objects as unobtrusive as possible.

As a result, Python flexmock removes a number of redundancies in the Ruby flexmock API, alters some defaults, and introduces a number of Python-only features.

flexmock's design focuses on simplicity and intuitivenes. This means that the API is as lean as possible, though a few convenient short-hand methods are provided to aid brevity and readability.

flexmock declarations are structured to read more like English sentences than API calls, and it is possible to chain them together in any order to achieve high degree of expressiveness in a single line of code.

CHAPTER 1

Installation

```
$ sudo easy_install flexmock
```

Or download the tarball, unpack it and run:

```
$ sudo python setup.py install
```

CHAPTER 2

Compatibility

Tested to work with:

- python 2.6
- python 2.7
- python 3.3
- python 3.4
- python 3.5
- pypy
- pypy3
- jython

Automatically integrates with all major test runners, including:

- unittest
- unittest2
- nose
- py.test
- django
- twisted / trial
- doctest
- zope.testrunner
- subunit
- testtools

CHAPTER 3

Start Here

3.1 Start Here

So what does flexmock actually help you do?

3.1.1 Creating fake objects

Making a new object in Python requires defining a new class with all the fake methods and attributes you're interested in emulating and then instantiating it. For example, to create a FakePlane object to use in a test in place of a real Plane object we would need to do something like:

```
class FakePlane(object):
    operational = True
    model = "MIG-21"
    def fly(self): pass

plane = FakePlane()  # this is tedious!
```

In other words, we must first create a class, make sure it contains all required attributes and methods, and finally instantiate it to create the object.

flexmock provides an easier way to generate a fake object on the fly using the flexmock() function:

```
plane = flexmock(
                operational=True,
                model="MIG-21")
```

It is also possible to add methods to this object using the same notation and Python's handy lambda keyword to turn an attribute into a method:

```
plane = flexmock(
                operational=True,
                model="MIG-21",
                fly=lambda: None)
```

3.1.2 Replacing parts of existing objects and classes (stubs)

While creating fake objects from scratch is often sufficient, many times it is easier to take an existing object and simply stub out certain methods or replace them with fake ones. flexmock makes this easy as well:

```
flexmock(
         Train,  # this can be an instance, a class, or a module
         get_destination="Tokyo",
         get_speed=200)
```

By passing a real object (or class or module) into the flexmock() function as the first argument it is possible to modify that object in place and provide default return values for any of its existing methods.

In addition to simply stubbing out return values, it can be useful to be able to call an entirely different method and substitute return values based on test-specific conditions:

```
(flexmock(Train)
    .should_receive("get_route")
    .replace_with(lambda x: custom_get_route()))
```

3.1.3 Creating and checking expectations

flexmock features smooth integration with pretty much every popular test runner, so no special setup is necessary. Simply importing flexmock into your test module is sufficient to get started with any of the following examples.

Mocks

Expectations take many flavors, and flexmock has many different facilities and modes to generate them. The first and simplest is ensuring that a certain method is called:

```
flexmock(Train).should_receive("get_destination").once()
```

The .once() modifier ensures that Train.get_destination() is called at some point during the test and will raise an exception if this does not happen.

Of course, it is also possible to provide a default return value:

```
flexmock(Train).should_receive("get_destination").once().and_return("Tokyo")
```

Or check that a method is called with specific arguments:

```
flexmock(Train).should_receive("set_destination").with_args("Tokyo").at_least().times(1)
```

In this example we used .times(1) instead of .once() and added the .at_least() modifier to demonstate that it is easy to match any number of calls, including 0 calls or a variable amount of calls. As you've probably guessed there is also an at_most() modifier.

Spies

While replacing method calls with canned return values or checking that they are called with specific arguments is quite useful, there are also times when you want to execute the actual method and simply find out how many times it was called. flexmock uses should_call() to generate this sort of expectations instead of should_receive():

```
flexmock(Train).should_call("get_destination").once()
```

In the above case the real get_destination() method will be executed, but flexmock will raise an exception unless it is executed exactly once. All the modifiers allowed with should_receive() can also be used with should_call() so it is possible to tweak the allowed arguments, return values and call times.

```
(flexmock(Train)
    .should_call("set_destination")
    .once()
    .with_args(object, str, int)
    .and_raise(Exception, re.compile("^No such dest.*")))
```

The above example introduces a handful of new capabilities – raising exceptions, matching argument types (object naturally matches any argument type) and regex matching on string return values and arguments.

3.1.4 Summary

flexmock has many other features and capabilities, but hopefully the above overview has given you enough of the flavor for the kind of things that it makes possible. For more details see the User Guide.

CHAPTER 4

User Guide

4.1 Usage Documentation

4.1.1 Definitions

In order to discuss flexmock usage it's important to define the following terms.

Stub fake object that returns a canned value

Mock fake object that returns a canned value and has an expectation, i.e. it includes a built-in assertion

Spy watches method calls and records/verifies if the method is called with required parameters and/or returns expected values/exceptions

4.1.2 Overview

flexmock declarations follow a consistent style of the following 3 forms:

```
flexmock ( OBJECT ).COMMAND( ATTRIBUTE ).MODIFIER[.MODIFIER, ...]

- or -

flexmock ( OBJECT [, ATTRIBUTE=VALUE, ...] )

- or -

flexmock ( ATTRIBUTE=VALUE [, ATTRIBUTE=VALUE,...] )


OBJECT:

  Either a module, a class, or an instance of a class

COMMAND:

  One of should_receive, should_call, or new_instances. These
  create the initial expectation object.

ATTRIBUTE:

  String name of an attribute
```

```
MODIFIER:

  One of several Expectation modifiers, such as with_args,
  and_return, should_raise, times, etc.

VALUE:

  Anything
```

4.2 Example Usage

4.2.1 Setup

```
from flexmock import flexmock
```

This will include flexmock in your test and make the necessary runner modifications so no further setup or cleanup is necessary.

Since version 0.10.0, it's also possible to call flexmock module directly, so you can just do:

```
import flexmock
```

4.2.2 Fake objects

```
fake = flexmock()  # creates an object with no attributes
```

Specify attribute/return value pairs

```
fake_plane = flexmock(
    model="MIG-16",
    condition="used")
```

Specify methods/return value pairs

```
fake_plane = flexmock(
    fly=lambda: "voooosh!",
    land=lambda: "landed!")
```

You can mix method and non-method attributes by making the return value a lambda for callable attributes.

flexmock fake objects support the full range of flexmock commands but differ from partial mocks (described below) in that should_receive() can assign them new methods rather than being limited to acting on methods they already possess.

```
fake_plane = flexmock(fly=lambda: "vooosh!")
fake_plane.should_receive("land").and_return("landed!")
```

4.2.3 Partial mocks

flexmock provides three syntactic ways to hook into an existing object and override its methods.

Mark the object as partially mocked, allowing it to be used to create new expectations

```
flexmock(plane)
plane.should_receive('fly').and_return('vooosh!').once()
plane.should_receive('land').and_return('landed!').once()
```

Equivalent syntax assigns the partially mocked object to a variable

```
plane = flexmock(plane)
plane.should_receive('fly').and_return('vooosh!').once()
plane.should_receive('land').and_return('landed!').once()
```

Or you can combine everything into one line if there is only one method to override

```
flexmock(plane).should_receive('fly').and_return('vooosh!').once()
```

You can also return the mock object after setting the expectations

```
plane = flexmock(plane).should_receive('fly').and_return('vooosh!').mock()
```

Note the "mock" modifier above – the expectation chain returns an expectation otherwise

```
plane.should_receive('land').with_args().and_return('foo', 'bar')
```

NOTE If you do not provide a with_args() modifier then any set of arguments, including none, will be matched. However, if you specify with_args() the expectation will only match exactly zero arguments.

NOTE If you do not provide a return value then None is returned by default. Thus, and_return() is equivalent to and_return(None) is equivalent to simply leaving off and_return.

4.2.4 Attributes and properties

Just as you're able to stub return values for functions and methods, flexmock also allows to stub out non-callable attributes and even (getter) properties. Syntax for this is exactly the same as for methods and functions.

4.2.5 Shorthand

Instead of writing out the lengthy should_receive/and_return statements, you can also use the handy shorthand approach of passing them in as key=value pairs to the flexmock() function. For example, we can stub out two methods of the plane object in the same call:

```
flexmock(plane, fly='voooosh!', land=('foo', 'bar'))
```

This approach is handy and quick but only limited to stubs, i.e. it is not possible to further modify these kind of calls with any of the usual modifiers described below.

4.2.6 Class level mocks

If the object your partially mock is a class, flexmock effectively replaces the method for all instances of that class.

```
>>> class User:
>>>     def get_name(self):
>>>         return 'George Bush'
>>>
>>> flexmock(User)
>>> User.should_receive('get_name').and_return('Bill Clinton')
>>> bubba = User()
```

```
>>> bubba.get_name()
'Bill Clinton'
```

4.2.7 Automatically checked expectations

Using the times(N) modifier, or its aliases – once, twice, never – allows you to create expectations that will be automatically checked by the test runner.

Ensure fly('forward') gets called exactly three times

```
(flexmock(plane)
    .should_receive('fly')
    .with_args('forward')
    .times(3))
```

Ensure turn('east') gets called at least twice

```
(flexmock(plane)
    .should_receive('turn')
    .with_args('east')
    .at_least().twice())
```

Ensure land('airfield') gets called at most once

```
(flexmock(plane)
    .should_receive('land')
    .with_args('airfield')
    .at_most().once())
```

Ensure that crash('boom!') is never called

```
(flexmock(plane)
    .should_receive('crash')
    .with_args('boom!')
    .never())
```

4.2.8 Exceptions

You can make the mocked method raise an exception instead of returning a value.

```
(flexmock(plane)
    .should_receive('fly')
    .and_raise(BadWeatherException))
```

Or you can add a message to the exception being raised

```
(flexmock(plane)
    .should_receive('fly')
    .and_raise(BadWeatherException, 'Oh noes, rain!'))
```

4.2.9 Spies (proxies)

In addition to stubbing out a given method and returning fake values, flexmock also allows you to call the original method and make expectations based on its return values/exceptions and the number of times the method is called with the given arguments.

Matching specific arguments

```
(flexmock(plane)
    .should_call('repair')
    .with_args(wing, cockpit)
    .once())
```

Matching any arguments

```
(flexmock(plane)
    .should_call('turn')
    .twice())
```

Matching specific return values

```
(flexmock(plane)
    .should_call('land')
    .and_return('landed!'))
```

Matching a regular expression

```
(flexmock(plane)
    .should_call('land')
    .and_return(re.compile('^la')))
```

Match return values by class/type

```
(flexmock(plane)
    .should_call('fly')
    .and_return(str, object, None))
```

Ensure that an appropriate exception is raised

```
(flexmock(plane)
    .should_call('fly')
    .and_raise(BadWeatherException))
```

Check that the exception message matches your expectations

```
(flexmock(plane)
    .should_call('fly')
    .and_raise(BadWeatherException, 'Oh noes, rain!'))
```

Check that the exception message matches a regular expression

```
(flexmock(plane)
    .should_call('fly')
    .and_raise(BadWeatherException, re.compile('rain')))
```

If either and_return() or and_raise() is provided, flexmock will verify that the return value matches the expected return value or exception.

> **NOTE** should_call() changes the behavior of and_return() and and_raise() to specify expectations rather than generate given values or exceptions.

4.2.10 Multiple return values

It's possible for the mocked method to return different values on successive calls.

```
>>> flexmock(group).should_receive('get_member').and_return('user1').and_return('user2').and_return(
>>> group.get_member()
'user1'
>>> group.get_member()
'user2'
>>> group.get_member()
'user3'
```

Or use the short-hand form

```
(flexmock(group)
    .should_receive('get_member')
    .and_return('user1', 'user2', 'user3')
    .one_by_one())
```

You can also mix return values with exception raises

```
(flexmock(group)
    .should_receive('get_member')
    .and_return('user1')
    .and_raise(Exception)
    .and_return('user2'))
```

4.2.11 Fake new instances

Occasionally you will want a class to create fake objects when it's being instantiated. flexmock makes it easy and painless.

Your first option is to simply replace the class with a function.

```
(flexmock(some_module)
    .should_receive('NameOfClass')
    .and_return(fake_instance))
# fake_instance can be created with flexmock as well
```

The upside of this approach is that it works for both new-style and old-style classes. The downside is that you may run into subtle issues since the class has now been replaced by a function.

If you're dealing with new-style classes, flexmock offers another alternative using the *.new_instances()* method.

```
>>> class Group(object): pass
>>> fake_group = flexmock(name='fake')
>>> flexmock(Group).new_instances(fake_group)
>>> Group().name == 'fake'
True
```

It is also possible to return different fake objects in a sequence.

```
>>> class Group(object): pass
>>> fake_group1 = flexmock(name='fake')
>>> fake_group2 = flexmock(name='real')
>>> flexmock(Group).new_instances(fake_group1, fake_group2)
>>> Group().name == 'fake'
True
>>> Group().name == 'real'
True
```

Another approach, if you're familiar with how instance instatiation is done in Python, is to stub the *__new__* method directly.

```
>>> flexmock(Group).should_receive('__new__').and_return(fake_group)
>>> # or, if you want to be even slicker
>>> flexmock(Group, __new__=fake_group)
```

In fact, the new_instances command is simply shorthand for *should_receive('__new__').and_return()* under the hood.

Note, that Python issue 25731 causes a problem with restoring the original *__new__* method. It has been already fixed upstream, but all versions of Python 3 lower than 3.5.2 are affected and will probably never receieve a bug fix for this. If you're using some of the affected versions and are getting *TypeError: object() takes no parameters*, you're hitting this issue (original bug report is at flexmock issue 13.

4.2.12 Generators

In addition to returning values and raising exceptions, flexmock can also turn the mocked method into a generator that yields successive values.

```
>>> flexmock(plane).should_receive('flight_log').and_yield('take off', 'flight', 'landing')
>>> for i in plane.flight_log():
>>>   print i
'take off'
'flight'
'landing'
```

You can also use Python's builtin iter() function to generate an iterable return value.

```
flexmock(plane, flight_log=iter(['take off', 'flight', 'landing']))
```

In fact, the and_yield() modifier is just shorthand for should_receive().and_return(iter) under the hood.

4.2.13 Private methods

One of the small pains of writing unit tests is that it can be difficult to get at the private methods since Python "conveniently" renames them when you try to access them from outside the object. With flexmock there is nothing special you need to do to – mocking private methods is exactly the same as any other methods.

4.2.14 Call order

flexmock does not enforce call order by default, but it's easy to do if you need to.

```
(flexmock(plane)
    .should_receive('fly')
    .with_args('forward')
    .and_return('ok')
    .ordered())
(flexmock(plane)
    .should_receive('fly')
    .with_args('up')
    .and_return('ok')
    .ordered())
```

The order of the flexmock calls is the order in which these methods will need to be called by the code under test.

If method fly() above is called with the right arguments in the declared order things will be fine and both will return 'ok'. But trying to call fly('up') before fly('forward') will result in an exception.

4.2.15 State Support

flexmock supports conditional method execution based on external state. Consider the rather contrived Radio class with the following methods:

```
>>> class Radio:
...   is_on = False
...   def switch_on(self): self.is_on = True
...   def switch_off(self): self.is_on = False
...   def select_channel(self): return None
...   def adjust_volume(self, num): self.volume = num
>>> radio = Radio()
```

Now we can define some method call expectations dependent on the state of the radio:

```
>>> flexmock(radio)
>>> radio.should_receive('select_channel').once().when(lambda: radio.is_on)
>>> radio.should_call('adjust_volume').once().with_args(5).when(lambda: radio.is_on)
```

Calling these while the radio is off will result in an error:

```
>>> radio.select_channel()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "flexmock.py", line 674, in mock_method
    (method, expectation._get_runnable()))
flexmock.StateError: select_channel expected to be called when condition is True

>>> radio.adjust_volume(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "flexmock.py", line 674, in mock_method
    (method, expectation._get_runnable()))
flexmock.StateError: adjust_volume expected to be called when condition is True
Traceback (most recent call last):
```

Turning the radio on will make things work as expected:

```
>>> radio.is_on = True
>>> radio.select_channel()
>>> radio.adjust_volume(5)
```

4.2.16 Chained methods

Let's say you have some code that looks something like the following:

```
http = HTTP()
results = (http.get_url('http://www.google.com')
               .parse_html()
               .display_results())
```

You could use flexmock to mock each of these method calls individually:

```
mock = flexmock(get_url=lambda: flexmock(parse_html=lambda: flexmock(display_results='ok')))
flexmock(HTTP).new_instances(mock)
```

But that looks really error prone and quite difficult to parse when reading. Here's a better way:

```
mock = flexmock()
flexmock(HTTP).new_instances(mock)
mock.should_receive('get_url.parse_html.display_results').and_return('ok')
```

When using this short-hand, flexmock will create intermediate objects and expectations, returning the final one in the chain. As a result, any further modifications, such as with_args() or times() modifiers, will only be applied to the final method in the chain. If you need finer grained control, such as specifying specific arguments to an intermediate method, you can always fall back to the above long version.

Word of caution: because flexmock generates temporary intermediate mock objects for each step along the chain, trying to mock two method call chains with the same prefix will not work. That is, doing the following will fail to set up the stub for display_results() because the one for save_results() overrides it:

```
flexmock(HTTP).should_receive('get_url.parse_html.display_results').and_return('ok')
flexmock(HTTP).should_receive('get_url.parse_html.save_results').and_return('ok')
```

In this situation, you should identify the point where the chain starts to diverge and return a flexmock() object that handles all the "tail" methods using the same object:

```
(flexmock(HTTP)
    .should_receive('get_url.parse_html')
    .and_return(flexmock(display_results='ok', save_results='ok')))
```

4.2.17 Replacing methods

There are times when it is useful to replace a method with a custom lambda or function, rather than simply stubbing it out, in order to return custom values based on provided arguments or a global value that changes between method calls.

```
(flexmock(plane)
    .should_receive('set_speed')
    .replace_with(lambda x: x == 5))
```

There is also shorthand for this, similar to the shorthand for should_receive/and_return:

```
flexmock(plane, set_speed=lambda x: x == 5)
```

NOTE Whenever the return value provided to the key=value shorthand is a callable (such as lambda), flexmock expands it to should_receive().replace_with() rather than should_receive().and_return().

4.2.18 Builtin functions

Mocking or stubbing out builtin functions, such as open(), can be slightly tricky. The "builtins" module is accessed differently in interactive Python sessions versus running applications and named differently in Python 3.0 and above.

It is also not always obvious when the builtin function you are trying to mock might be internally called by the test runner and cause unexpected behavior in the test. As a result, the recommended way to mock out builtin functions is to always specify a fall-through with should_call() first and use with_args() to limit the scope of your mock or stub to just the specific invocation you are trying to replace:

```
# python 2.4+
mock = flexmock(sys.modules['__builtin__'])
mock.should_call('open')  # set the fall-through
(mock.should_receive('open')
    .with_args('/your/file')
    .and_return( flexmock(read=lambda: 'file contents') ))
```

```
# python 3.0+
mock = flexmock(sys.modules['builtins'])
mock.should_call('open')  # set the fall-through
(mock.should_receive('open')
    .with_args('/your/file')
    .and_return( flexmock(read=lambda: 'file contents') ))
```

4.3 Expectation Matching

Creating an expectation with no arguments will by default match all arguments, including no arguments.

```
>>> flexmock(plane).should_receive('fly').and_return('ok')
```

Will be matched by any of the following:

```
>>> plane.fly()
'ok'
>>> plane.fly('up')
'ok'
>>> plane.fly('up', 'down')
'ok'
```

You can also match exactly no arguments

```
(flexmock(plane)
    .should_receive('fly')
    .with_args())
```

Or match any single argument

```
(flexmock(plane)
    .should_receive('fly')
    .with_args(object))
```

NOTE In addition to exact values, you can match against the type or class of the argument.

Match any single string argument

```
(flexmock(plane)
    .should_receive('fly')
    .with_args(str))
```

Match the empty string using a compiled regular expression

```
regex = re.compile('^(up|down)$')
(flexmock(plane)
    .should_receive('fly')
    .with_args(regex))
```

Match any set of three arguments where the first one is an integer, second one is anything, and third is string 'notes' (matching against user defined classes is also supported in the same fashion)

```
(flexmock(plane)
    .should_receive('repair')
    .with_args(int, object, 'notes'))
```

And if the default argument matching based on types is not flexible enough, flexmock will respect matcher objects that provide a custom __eq__ method.

For example, when trying to match against contents of numpy arrays, equality is undefined by the library so comparing two of them directly is meaningless unless you use all() or any() on the return value of the comparison.

What you can do in this case is create a custom matcher object and flexmock will use its __eq__ method when comparing the arguments at runtime.

```
class NumpyArrayMatcher(object):
    def __init__(self, array): self.array = array
    def __eq__(self, other): return all(other == self.array)

(flexmock(obj)
    .should_receive('function')
    .with_args(NumpyArrayMatcher(array1)))
```

The above approach will work for any objects that choose not to return proper boolean comparison values, or if you simply find the default equality and type-based matching not sufficiently specific.

It is, of course, also possible to create multiple expectations for the same method differentiated by arguments.

```
>>> flexmock(plane).should_receive('fly').and_return('ok')
>>> flexmock(plane).should_receive('fly').with_args('up').and_return('bad')
```

Try to excecute plane.fly() with any, or no, arguments as defined by the first flexmock call will return the first value.

```
>>> plane.fly()
'ok'
>>> plane.fly('forward', 'down')
'ok'
```

But! If argument values match the more specific flexmock call the function will return the other return value.

```
>>> plane.fly('up')
'bad'
```

The order of the expectations being defined is significant, with later expectations having higher precedence than previous ones. Which means that if you reversed the order of the example expectations above the more specific expectation would never be matched.

4.4 Style

While the order of modifiers is unimportant to flexmock, there is a preferred convention that will make your tests more readable.

If using with_args(), place it before should_return(), and_raise() and and_yield() modifiers:

```
(flexmock(plane)
    .should_receive('fly')
    .with_args('up', 'down')
    .and_return('ok'))
```

If using the times() modifier (or its aliases: once, twice, never), place them at the end of the flexmock statement:

```
(flexmock(plane)
    .should_receive('fly')
    .and_return('ok')
    .once())
```

CHAPTER 5

API

5.1 flexmock API

CHAPTER 6

Changelog

6.1 Changelog

6.1.1 Release 0.10.2

- fix recognizing whether mocked object is a method or not on Python 3

6.1.2 Release 0.10.1

- fix decode problem in setup.py on Python 3

6.1.3 Release 0.10.0

- new official upstream repository: https://github.com/bkabrda/flexmock/
- new official homepage: https://flexmock.readthedocs.org
- adopted the official BSD 2-clause license https://en.wikipedia.org/wiki/BSD_licenses#2-clause_license_.28.22Simplified_BSD_License.22_or_.22FreeBSD_License.22.29
- add support for calling flexmock module directly
- add support for mocking keyword-only args
- add support for Python 3.4 and 3.5
- drop support for Python 2.4, 2.5, 3.1 and 3.2
- add `__version__` attribute to flexmock module
- add various metadata to the package archive
- fix properly find out whether function is method or not and thanks to that don't strip first args of functions
- fix should_call to work when function returns `None` or `False`
- fix various py.test issues
- fix `CallOrderError` with same subsequent mocking calls
- fix PyPy support issues
- various code style issues were fixed, 4-spaces indent is now used

6.1.4 Release 0.9.7

- small update to add support for TeamCity / PyCharm test runner.

6.1.5 Release 0.9.6

- fix staticmethod mocking on instances
- fix comparison of kwargs ordering issues
- fix `ReturnValue.__str__`

6.1.6 Release 0.9.5

- bugfix: stop enforcing argument signatures on flexmock objects

6.1.7 Release 0.9.4

- add support for stubbing return values on getter properties
- add custom matcher object support to `with_args`
- add support for striter function signature checks
- add support for non-callable attributes
- add support chained attributes (thanks Bryce Covert!)
- add iter support to `Mock` objects
- add PyPy support
- add Jython support
- fix `should_call` to work with class mocks
- fix `and_return` to return `None` by default
- fix MRO issues on builtin methods on 2.7+/3.2+
- imporove defaults: partial mocks created using the `func=return_value` style now default to `replace_with` instead of `should_receive` for callables

6.1.8 Release 0.9.3

- add python 3.3 test target
- add proper handling of `ordered()` expectation across different methods
- add property support on fake objects
- fix compatibility with pytest 2.2 (thanks jpvanhal!)
- fix insidious bug with mocking subclasses of `str` class
- fix `tuple` handling when formatting arguments
- fix reseting subclass methods

6.1.9 Release 0.9.2

- fix mocking builtins by reseting expectation when raising exceptions
- fix mocking private methods on classes with leading underscores
- limit the damage of `from flexmock import *` by limiting to just `flexmock()`
- ensure `_pre_flexmock_success` is cleaned up after each test

6.1.10 Release 0.9.1

- adding support for more test runners:
  - unittest2
  - django
  - twisted/trial
  - zope.testrunner
  - subunit
  - testtools

6.1.11 Release 0.9.0

- adding state machine support using `when()`
- make expectation fail as soon as number of expected calls is exceeded
- `flexmock_teardown` no longer returns a function
- allow `should_call` on class and static methods
- disallow `should_call` on class mocks
- fixing `unicode` args handling
- fixing issues with `@property` methods misbehaving in the debugger
- fixing pytest integration and instance teardown
- fixing private method handling

6.1.12 Release 0.8.1

- fixing pytest and doctest integration to always call `flexmock_teardown`
- fixing `flexmock_teardown` to return a function as before so it can be used as a decorator

6.1.13 Release 0.8.0

- big changes in runner integration support (no more stack examination or sketchy teardown replacement)
- doctest integration
- fixing ordering verification when the method has a default stub
- fixing calling `with_args()` without arguments to match exactly no arguments (thanks jerico-dev!)

- 20% performance improvement
- make sure to return object itself when partial mocking instances unless the object already has some of the methods
- ensure consecutive calls return same mock object

6.1.14 Release 0.7.4.2

- adding regex support for arg matching and spy return values
- enabling `replace_with` for class mocks
- disabling expectation checking if an exception has already been raised
- massive refactoring of the way flexmock does monkey patching

6.1.15 Release 0.7.4.1

- Fixing replace_with to work properly like `and_execute`
- (`and_execute` will be deprecated in next release!)

6.1.16 Release 0.7.4

- Fixed exception type check when no message specified
- Make properties work optionally with parentheses
- Make sure `should_receive` does not replace flexmock methods
- Removed `new_instances=` param in favor of `new_instances()` method
- Refactoring to move all state to `FlexmockContainer` class

6.1.17 Release 0.7.3

- Added `new_instances` method (`new_instances` param will be deprecated in next release!)
- Added `replace_with` to enable returning results of custom functions
- Added `with` support for `FlexMock` objects
- Moved tests to their own directory
- Lots of documentation cleanup and updates

6.1.18 Release 0.7.2

- Added support for chained methods
- Moved `flexmock_teardown` to module level to expose it for other test runners
- Added py.test support (thanks to derdon)
- Lots of test refactoring and improvements for multiple test runner support
- Fix loop in teardown

- Fix `should_call` for same method with different args

6.1.19 Release 0.7.1

- Fix bug with "never" not working when the expectation is not met
- Fix bug in duplicate calls to original method in `pass_thru` mode (thanks sagara-!)
- Fix bug in handling unicode characters in `ReturnValue`

6.1.20 Release 0.7.0

- Better error handling for trying to mock builtins
- Added simple test harness for running on multiple versions / test runners
- Fixed `unicode` arg formatting (thanks to sagara-!)
- Made it impossible to mock non-existent methods
- Ensure flexmock teardown takes varargs (for better runner integration)

6.1.21 Release 0.6.9

- Initial nose integration (still no support for generated tests)
- Fixing private class methods
- Some test refactoring to support different test runners

6.1.22 Release 0.6.8

- Add `should_call()` alias for `should_receive().and_execute`
- Ensure `new_instances` can't be used with expectation modifiers
- Make `and_execute` match return value by class in addition to value
- Support for mocking out static methods
- Bit of test fixage (thanks to derdon)

6.1.23 Release 0.6.7

- Fixing clobbering of original method by multiple flexmock calls
- Making `and_raise` work properly with exception classes and args
- Proper exception matching with `and_execute`
- Fix mocking same class twice

6.1.24 Release 0.6.6

- Removing extra args from `should_receive`
- Making `and_execute` check return/raise value of original method
- Refactoring FlexMock constructor into factory method
- Fixing `new_instances` to accept multiple args instead of just none
- Raising an exception when `and_execute` is set on class mock

6.1.25 Release 0.6.5

- Adding support for multiple `flexmock()` calls on same object
- Adding error detection on `and_execute` for missing or unbound methods
- Make sure empty args don't include `None`

6.1.26 Release 0.6.4

- Fixing up teardown cleanup code after an exception is raised in tests
- Fixing `and_yield` to return proper generator
- Adding `and_yield` returning a predefined generator
- Replacing `and_passthru` with `and_execute`
- Make it easier to mock private methods

6.1.27 Release 0.6.3

- Adding keyword argument expectation matching

6.1.28 Release 0.6.2

- Changing `and_return(multiple=True)` to `one_by_one`
- Making it possible to supply multiple args to `and_return` instead of a tuple
- Changing default mock behavior to create attributes instead of methods
- FIX teardown for python3

6.1.29 Release 0.6.1

- Make it even easier to integrate with new test runners
- Adding support for mixing returns and raises in return values

6.1.30 Release 0.6

- Adding support for multiple arg type matches
- Pulling out the entry point code from constructor into its own method.

6.1.31 Release 0.5

- FIX: ensuring that mocks are cleaned up properly between tests
- BROKEN: part1 on ensuring mocking multiple objects works correctly
- Make sure `pass_thru` doesn't try to call a non-existent method
- Fixing up copyright notice
- Adding some missing pydocs

6.1.32 Release 0.4

- Fixing tests and ensuring mock methods really get created properly
- Making sure shortcuts create methods rather than attributes
- Fixing doc strings
- Removing the new-style/old-style convert code, it's stupid

6.1.33 Release 0.3

- Making `Expectation.mock` into a property so that it shows up in pydoc
- Adding proxying/spying and `at_least`/`at_most` expectation modifiers

CHAPTER 7

Comparison

7.1 Mock Library Comparison

7.1.1 (flexmock for Mox or Mock users.)

This document shows a side-by-side comparison of how to accomplish some basic tasks with flexmock as well as other popular Python mocking libraries.

Simple fake object (attributes only)

```
# flexmock
mock = flexmock(some_attribute="value", some_other_attribute="value2")
assertEquals("value", mock.some_attribute)
assertEquals("value2", mock.some_other_attribute)

# Mox
mock = mox.MockAnything()
mock.some_attribute = "value"
mock.some_other_attribute = "value2"
assertEquals("value", mock.some_attribute)
assertEquals("value2", mock.some_other_attribute)

# Mock
my_mock = mock.Mock()
my_mock.some_attribute = "value"
my_mock.some_other_attribute = "value2"
assertEquals("value", my_mock.some_attribute)
assertEquals("value2", my_mock.some_other_attribute)
```

Simple fake object (with methods)

```
# flexmock
mock = flexmock(some_method=lambda: "calculated value")
assertEquals("calculated value", mock.some_method())

# Mox
mock = mox.MockAnything()
mock.some_method().AndReturn("calculated value")
mox.Replay(mock)
```

```
assertEquals("calculated value", mock.some_method())

# Mock
my_mock = mock.Mock()
my_mock.some_method.return_value = "calculated value"
assertEquals("calculated value", my_mock.some_method())
```

Simple mock

```
# flexmock
mock = flexmock()
mock.should_receive("some_method").and_return("value").once()
assertEquals("value", mock.some_method())

# Mox
mock = mox.MockAnything()
mock.some_method().AndReturn("value")
mox.Replay(mock)
assertEquals("value", mock.some_method())
mox.Verify(mock)

# Mock
my_mock = mock.Mock()
my_mock.some_method.return_value = "value"
assertEquals("value", mock.some_method())
my_mock.some_method.assert_called_once_with()
```

Creating partial mocks

```
# flexmock
flexmock(SomeObject).should_receive("some_method").and_return('value')
assertEquals("value", mock.some_method())

# Mox
mock = mox.MockObject(SomeObject)
mock.some_method().AndReturn("value")
mox.Replay(mock)
assertEquals("value", mock.some_method())
mox.Verify(mock)

# Mock
with mock.patch("SomeObject") as my_mock:
  my_mock.some_method.return_value = "value"
  assertEquals("value", mock.some_method())
```

Ensure calls are made in specific order

```
# flexmock
mock = flexmock(SomeObject)
mock.should_receive('method1').once().ordered().and_return('first thing')
mock.should_receive('method2').once().ordered().and_return('second thing')
# exercise the code

# Mox
```

```
mock = mox.MockObject(SomeObject)
mock.method1().AndReturn('first thing')
mock.method2().AndReturn('second thing')
mox.Replay(mock)
# exercise the code
mox.Verify(mock)

# Mock
mock = mock.Mock(spec=SomeObject)
mock.method1.return_value = 'first thing'
mock.method2.return_value = 'second thing'
# exercise the code
assert mock.method_calls == [('method1',) ('method2',)]
```

Raising exceptions

```
# flexmock
mock = flexmock()
mock.should_receive("some_method").and_raise(SomeException("message"))
assertRaises(SomeException, mock.some_method)

# Mox
mock = mox.MockAnything()
mock.some_method().AndRaise(SomeException("message"))
mox.Replay(mock)
assertRaises(SomeException, mock.some_method)
mox.Verify(mock)

# Mock
my_mock = mock.Mock()
my_mock.some_method.side_effect = SomeException("message")
assertRaises(SomeException, my_mock.some_method)
```

Override new instances of a class

```
# flexmock
flexmock(some_module.SomeClass).new_instances(some_other_object)
assertEqual(some_other_object, some_module.SomeClass())

# Mox
# (you will probably have mox.Mox() available as self.mox in a real test)
mox.Mox().StubOutWithMock(some_module, 'SomeClass', use_mock_anything=True)
some_module.SomeClass().AndReturn(some_other_object)
mox.ReplayAll()
assertEqual(some_other_object, some_module.SomeClass())

# Mock
with mock.patch('somemodule.Someclass') as MockClass:
  MockClass.return_value = some_other_object
  assert some_other_object == some_module.SomeClass()
```

Verify a method was called multiple times

```
# flexmock (verifies that the method gets called at least twice)
flexmock(some_object).should_receive('some_method').at_least().twice()
# exercise the code

# Mox
# (does not support variable number of calls, so you need to create a new entry for each explicit ca
mock = mox.MockObject(some_object)
mock.some_method(mox.IgnoreArg(), mox.IgnoreArg())
mock.some_method(mox.IgnoreArg(), mox.IgnoreArg())
mox.Replay(mock)
# exercise the code
mox.Verify(mock)

# Mock
my_mock = mock.Mock(spec=SomeObject)
# exercise the code
assert my_mock.some_method.call_count >= 2
```

Mock chained methods

```
# flexmock
# (intermediate method calls are automatically assigned to temporary fake objects
# and can be called with any arguments)
(flexmock(some_object)
    .should_receive('method1.method2.method3')
    .with_args(arg1, arg2)
    .and_return('some value'))
assertEqual('some_value', some_object.method1().method2().method3(arg1, arg2))

# Mox
mock = mox.MockObject(some_object)
mock2 = mox.MockAnything()
mock3 = mox.MockAnything()
mock.method1().AndReturn(mock1)
mock2.method2().AndReturn(mock2)
mock3.method3(arg1, arg2).AndReturn('some_value')
self.mox.ReplayAll()
assertEqual("some_value", some_object.method1().method2().method3(arg1, arg2))
self.mox.VerifyAll()

# Mock
my_mock = mock.Mock()
my_mock.method1.return_value.method2.return_value.method3.return_value = 'some value'
method3 = my_mock.method1.return_value.method2.return_value.method3
method3.assert_called_once_with(arg1, arg2)
assertEqual('some_value', my_mock.method1().method2().method3(arg1, arg2))
```

Mock context manager

```
# flexmock
my_mock = flexmock()
with my_mock:
    pass
```

```
# Mock
my_mock = mock.MagicMock()
with my_mock:
    pass

# Mox
my_mock = mox.MockAnything()
with my_mock:
    pass
```

Mocking the builtin open used as a context manager

The following examples work in an interactive Python session but may not work quite the same way in a script, or with Python 3.0+. See examples in the *Builtin functions* section for more specific flexmock instructions on mocking builtins.

```
# flexmock
(flexmock(__builtins__)
    .should_receive('open')
    .once()
    .with_args('file_name')
    .and_return(flexmock(read=lambda: 'some data')))
with open('file_name') as f:
    assertEqual('some data', f.read())

# Mox
self_mox = mox.Mox()
mock_file = mox.MockAnything()
mock_file.read().AndReturn('some data')
self_mox.StubOutWithMock(__builtins__, 'open')
__builtins__.open('file_name').AndReturn(mock_file)
self_mox.ReplayAll()
with mock_file:
    assertEqual('some data', mock_file.read())
self_mox.VerifyAll()

# Mock
with mock.patch('__builtin__.open') as my_mock:
    my_mock.return_value.__enter__ = lambda s: s
    my_mock.return_value.__exit__ = mock.Mock()
    my_mock.return_value.read.return_value = 'some data'
    with open('file_name') as h:
        assertEqual('some data', h.read())
my_mock.assert_called_once_with('foo')
```

A possibly more up-to-date version of this document, featuring more mocking libraries, is availale at:

http://garybernhardt.github.com/python-mock-comparison/

f

F