sinon Documentation

Release 0.1.1

Kir Chou

February 18, 2017

Contents

Note: This document is partially referenced from Sinon.JS. However, most usages and API are redesigned.

Sinon.PY is inspired by Sinon.JS. Standalone test spies, stubs and mocks for Python. No dependencies, works with any unit testing framework.

CHAPTER 1

Overview

This document contains the entire Sinon.PY API documentation along with brief introductions and examples to the concepts Sinon implements.

CHAPTER 2

Contents

Setup

Installation

1. Get the stable Sinon.PY from pypi:

```
$ pip install sinon
```

2. Install latest version manually:

```
$ git clone https://github.com/note35/sinon.git
$ cd sinon/
$ python setup.py install
```

3. With venv, since Sinon.PY is a library for unittest, It would be good if using virtualenv in development environment by following command below:

```
$ pip install virtualenv
$ virtualenv env
$ . env/bin/activate
$ pip install sinon
```

Getting Started

Spies

```
>>> import sinon
>>> import os
>>> spy = sinon.spy(os, "system")
>>> os.system("pwd")
to/your/current/path
>>> spy.called
True
>>> spy.calledWith("pwd")
True
>>> spy.calledWith(sinon.match(str))
True
>>> spy.calledOnce
True
>>> spy.restore()
```

Stubs

```
>>> import sinon
>>> import os
>>> stub = sinon.stub(os, "system").returns("stub result")
>>> os.system("pwd")
'stub result'
>>> stub.restore()
>>>
>>> stub = sinon.stub(os, "system").onCall(2).returns("stub result")
>>> os.system("pwd")
>>> os.system("pwd")
'stub result'
>>> os.system("pwd")
>>> stub.restore()
>>>
>>> sinon.stub(os, "system").throws()
>>> stub = sinon.stub(os, "system").throws()
>>> try:
...     system("pwd")
... except:
...     pass
...
>>> stub.restore()
```

Spies

What is a test spy?

A test spy is a function that records arguments, return value, the value of this and exception thrown (if any) for all its calls. A test spy can be an anonymous function or it can wrap an existing function.

When to use spies?

Test spies are useful to test both callbacks and how certain functions/methods are used throughout the system under test.

Example1: use spies to test how a function handles a callback

```
import sinon

def test_handle_empty_spy():
    callback = sinon.spy()

    def __inner_func(func):
        func("expected_args")

    __inner_func(spy)

    assert spy.called
    assert spy.calledWith("expected_args")

if __name__ == "__main__":
    test_handle_empty_spy()
```

Example2: Spying on existing methods

```
import sinon
import os

def test_inspect_os_system():
    spy = sinon.spy(os, "system")
    os.system("pwd")
    assert spy.called
    assert spy.calledWith("pwd")

if __name__ == "__main__":
    test_inspect_os_system()
```

Creating spies: sinon.spy()

spy = sinon.spy()

Creates an anonymous function that records arguments, exceptions and return values for all calls.

spy = sinon.spy(myFunc)

Spies on the provided function

Note: If there is a function declare in the same scope of unittest file, you should use *Scope API*. You can read the reason from *Why should use scope?*

spy = sinon.spy(class|instance|module, "method")

Creates a spy for object.method and wraps the original method. The spy acts exactly like the original method in all cases. The original method can be restored by calling *.restore()*. The returned spy is the function object which replaced the original method.

Because in python2, if im_self is empty, the unbound function will not have fixed id, thus **class is only supported by python3**.

Spy API

spy.callCount

The number of recorded calls.

```
spy = sinon.spy(os, "system")
assert spy.callCount == 0
os.system("pwd")
assert spy.callCount == 1
os.system("pwd")
assert spy.callCount == 2
```

spy.called

true if the spy was called at least once

```
spy = sinon.spy(os, "system")
assert not spy.called
os.system("pwd")
assert spy.called
```

spy.calledOnce

true if spy was called exactly once

```
spy = sinon.spy(os, "system")
assert not spy.calledOnce
os.system("pwd")
assert spy.calledOnce
os.system("pwd")
assert not spy.calledOnce
```

spy.calledTwice

true if the spy was called exactly twice

```
spy = sinon.spy(os, "system")
assert not spy.calledTwice
os.system("pwd")
assert not spy.calledTwice
os.system("pwd")
assert spy.calledTwice
```

spy.calledThrice

true if the spy was called exactly thrice

```
spy = sinon.spy(os, "system")
assert not spy.calledThrice
os.system("pwd")
assert not spy.calledThrice
os.system("pwd")
assert not spy.calledThrice
os.system("pwd")
assert spy.calledThrice
```

spy.firstCall

The first call

```
spy = sinon.spy(os, "system")
spy2 = sinon.spy(os, "getcwd")
os.system("pwd")
os.getcwd()
assert spy.firstCall
```

spy.secondCall

The second call

```
spy = sinon.spy(os, "system")
spy2 = sinon.spy(os, "getcwd")
os.system("pwd")
os.getcwd()
assert spy2.secondCall
```

spy.thirdCall

The third call

```
spy = sinon.spy(os, "system")
spy2 = sinon.spy(os, "getcwd")
os.system("pwd")
```

```
os.getcwd()
os.system("pwd")
assert spy.thirdCall
```

spy.thirdCall

The third call

```
spy = sinon.spy(os, "system")
spy2 = sinon.spy(os, "getcwd")
os.system("pwd")
os.getcwd()
os.system("pwd")
assert spy.thirdCall
```

spy.lastCall

The last call

```
spy = sinon.spy(os, "system")
spy2 = sinon.spy(os, "getcwd")
os.system("pwd")
assert spy.lastCall
os.getcwd()
assert not spy.lastCall
assert spy2.lastCall
```

spy.calledBefore(anotherSpy)

Returns true if the spy was called before anotherSpy

```
spy = sinon.spy(os, "system")
spy2 = sinon.spy(os, "getcwd")
os.system("pwd")
os.getcwd()
assert spy.calledBefore(spy2)
```

spy.calledAfter(anotherSpy)

Returns true if the spy was called after anotherSpy

```
spy = sinon.spy(os, "system")
spy2 = sinon.spy(os, "getcwd")
os.system("pwd")
os.getcwd()
assert spy2.calledAfter(spy)
```

spy.calledWith(*args, **kwargs)

Returns true if spy was called at least once with the provided arguments. Can be used for partial matching, Sinon only checks the provided arguments against actual arguments, so a call that received the provided arguments (in the same spots) and possibly others as well will return true.

```
spy = sinon.spy(os, "system")
os.system("pwd")
assert spy.calledWith("pwd")
```

spy.alwaysCalledWith(*args, **kwargs)

Returns true if spy was always called with the provided arguments (and possibly others).

```
spy = sinon.spy(os, "system")
os.system("pwd")
assert spy.alwaysCalledWith("pwd")
os.system("ls")
assert not spy.alwaysCalledWith("pwd")
```

spy.calledWithExactly(*args, **kwargs)

Returns true if spy was called at least once with the provided arguments and no others.

```
spy = sinon.spy(os, "getenv")
os.getenv("NOT_EXIST_ENV_VAR", "DEFAULT_VALUE")
assert spy.calledWithExactly("NOT_EXIST_ENV_VAR", "DEFAULT_VALUE")
assert not spy.calledWithExactly("NOT_EXIST_ENV_VAR")
assert spy.calledWith("NOT_EXIST_ENV_VAR")
```

spy.alwaysCalledWithExactly(*args, **kwargs)

Returns true if spy was always called with the exact provided arguments.

```
spy = sinon.spy(os, "getenv")
os.getenv("NOT_EXIST_ENV_VAR", "DEFAULT_VALUE")
assert spy.alwaysCalledWithExactly("NOT_EXIST_ENV_VAR", "DEFAULT_VALUE")
os.getenv("NOT_EXIST_ENV_VAR", "ANOTHER_VALUE")
assert not spy.alwaysCalledWithExactly("NOT_EXIST_ENV_VAR", "DEFAULT_VALUE")
```

spy.calledWithMatch(*args, **kwargs)

Returns true if spy was called with matching arguments (and possibly others). This behaves the same as spy.calledWith(sinon.match(arg1), sinon.match(arg2), ...).

```
spy = sinon.spy(os, "system")
os.system("pwd")
assert spy.calledWithMatch(str)
assert spy.calledWith(sinon.match(str))
```

spy.alwaysCalledWithMatch(*args, **kwargs)

Returns true if spy was always called with matching arguments (and possibly others). This behaves the same as spy.alwaysCalledWith(sinon.match(arg1), sinon.match(arg2), ...).

```
spy = sinon.spy(os, "system")
os.system("pwd")
assert spy.alwaysCalledWithMatch(str)
os.system("ls")
assert spy.alwaysCalledWithMatch(str)
```

spy.neverCalledWith(*args, **kwargs)

Returns true if the spy/stub was never called with the provided arguments.

```
spy = sinon.spy(os, "system")
assert spy.neverCalledWith(None)
os.system("pwd")
assert spy.neverCalledWith("ls")
```

spy.neverCalledWithMatch(*args, **kwargs)

Returns true if the spy/stub was never called with matching arguments. This behaves the same as spy.neverCalledWith(sinon.match(arg1), sinon.match(arg2), ...).

```
spy = sinon.spy(os, "system")
os.system("pwd")
assert spy.neverCalledWithMatch(int)
```

spy.threw(Exception=None)

Returns true if spy threw an provided exception at least once. By default, all exception is included.

```
spy = sinon.spy(os, "getenv")
try:
    os.getenv(1000000000)
except:
    assert spy.threw()
    assert spy.threw(TypeError)
```

spy.alwaysThrew(Exception=None)

Returns true if spy always threw an provided exception.

spy.returned(obj)

Returns true if spy returned the provided value at least once.

```
spy = sinon.spy(os, "system")
os.system("ls")
assert spy.returned(0)
```

spy.alwaysReturned(obj)

Returns true if spy returned the provided value at least once.

```
spy = sinon.spy(os, "system")
os.system("ls")
os.system("not exist command") # return non-zero value
assert not spy.alwaysReturned(0)
```

var spyCall = spy.getCall(n)

Returns the nth [call](#spycall).

```
sinon.spy(os, "getcwd")
os.getcwd()
spy = sinon.spy.getCall(0)
spy.calledWith("getcwd")
```

spy.args

Array of arguments received, spy.args is a list of arguments(tuple).

```
spy = sinon.spy(os, "getenv")
os.getenv("NOT_EXIST_ENV_VAR", "DEFAULT_VALUE")
os.getenv("NOT_EXIST_ENV_VAR2")
assert spy.args == [('NOT_EXIST_ENV_VAR', 'DEFAULT_VALUE'), ('NOT_EXIST_ENV_VAR2',)]
```

spy.kwargs

Array of arguments received, spy.args is a list of arguments(dict).

spy.exceptions

Array of exception objects thrown, spy.exceptions is a list of exceptions thrown by the spy. If the spy did not throw an error, the value will be empty.

```
spy = sinon.spy(os, "getenv")
try:
    os.getenv(1000000000)
except:
    assert spy.exceptions == [TypeError]
```

spy.returnValues

Array of return values, spy.returnValues is a list of values returned by the spy. If the spy did not return a value, the value will be None.

```
spy = sinon.spy(os, "system")
os.system("ls")
assert spy.returnValues == [0]
```

spy.reset()

Resets the state of a spy.

```
spy = sinon.spy(os, "system")
os.system("pwd")
assert spy.callCount == 1
os.reset()
assert spy.callCount == 0
```

Stubs

What are stubs?

Test stubs are functions (spies) with pre-programmed behavior. They support the full test *Spy API* in addition to methods which can be used to alter the stub's behavior.

As spies, stubs can be either anonymous, or wrap existing functions. When wrapping an existing function with a stub, the original function is not called.

When to use stubs?

Use a stub when you want to:

1. Control a method's behavior from a test to force the code down a specific path. Examples include forcing a method to throw an error in order to test error handling.
2. When you want to prevent a specific method from being called directly (possibly because it triggers undesired behavior).

Creating stubs: sinon.stub()

stub = sinon.stub()

Creates an anonymous stub function.

stub = sinon.stub(class/instance/module)

Stubs on the provided class/instance/module, which will be replaced into an *Empty Class*.

stub = sinon.stub(myFunc)

Stubs on the provided function

Note: If there is a function declare in the same scope of unittest file, you should use *Scope API*. You can read the reason from *Why should use scope?*

stub = sinon.stub(class/instance/module, "method")

Creates a stub for object.method and wraps the original method. The stub acts exactly an *Empty Function* in all cases. The original method can be restored by calling *.restore()*. The returned stub is the function object which replaced the original method.

stub = sinon.stub(class/instance/module, "method", func)

Creates a stub for object.method and wraps the original method. The stub acts exactly an provided func in all cases. The original method can be restored by calling *.restore()*. The returned stub is the function object which replaced the original method.

Because in python2, if im_self is empty, the unbound function will not have fixed id, thus **class is only supported by python3**.

Stub API

Defining stub behavior on consecutive calls

Calling behavior defining methods like returns or throws multiple times overrides the behavior of the stub. You can use the onCall method to make a stub respond differently on consecutive calls.

stub.withArgs(*args, **kwargs)

Stubs the method only for the provided arguments. This is useful to be more expressive in your assertions, where you can access the spy with the same call. It is also useful to create a stub that can act differently in response to different arguments.

stub.withArgs(*args, **kwargs)

Stubs the method only for the provided arguments. This is useful to be more expressive in your assertions, where you can access the spy with the same call. It is also useful to create a stub that can act differently in response to different arguments.

```
stub = sinon.stub()
stub.withArgs(42).returns(1)
stub.withArgs(1).throws(TypeError)

assert stub() == None
assert stub(42) == 1
try:
    stub(1) # Throws TypeError
except:
    pass
stub.exceptions == [TypeError]
```

stub.onCall(n)

Defines the behavior of the stub on the nth call. Useful for testing sequential interactions.

There are methods onFirstCall, onSecondCall,onThirdCall to make stub definitions read more naturally.

```
stub = sinon.stub()
stub.onCall(0).returns(1)
stub.onCall(1).returns(2)
stub.returns(3)
```

```
assert stub() == 1
assert stub() == 2
assert stub() == 3
assert stub() == 3
```

stub.onFirstCall()

Alias for stub.onCall(0);

```
stub = sinon.stub()
stub.onFirstCall().returns(1)
assert stub() == 1
assert stub() == None
```

stub.onSecondCall()

Alias for stub.onCall(1)

```
stub = sinon.stub()
stub.onSecondCall().returns(2)
assert stub() == None
assert stub() == 2
```

stub.onThirdCall()

Alias for stub.onCall(2)

```
stub = sinon.stub()
stub.onThirdCall().returns(3)
assert stub() == None
assert stub() == None
assert stub() == 3
```

stub.returns(obj)

Makes the stub return the provided value.

```
stub = sinon.stub()
stub.returns(["list"])
assert stub() == ["list"]
stub.returns(object)
assert stub() == object
```

stub.throws(exception=Exception)

Causes the stub to throw an exception, default exception is Exception.

```
stub = sinon.stub()
stub.throws(TypeError)
try:
    stub()
except TypeError:
    pass
assert stub.exceptions == [TypeError]
```

Empty Class

```
class EmptyClass(object):
    pass
```

Empty Function

```
def empty_function(\*args, \*\*kwargs):
    pass
```

Mocks

What are mocks?

Mocks (and mock expectations) are fake methods (like spies) with pre-programmed behavior (like stubs) as well as pre-programmed expectations. A mock will return False if it is not used as expected.

When to use mocks?

Mocks should only be used for the method under test. In every unit test, there should be one unit under test. If you want to control how your unit is being used and like stating expectations upfront (as opposed to asserting after the fact), use a mock.

When to not use mocks?

In general you should never have more than one mock (possibly with several expectations) in a single test.

Expectations implement both the *Spy API* and *Stub API*.

To see how mocks look like in Sinon.JS, here's one of the tests example:

```
mock = sinon.mock(os)

mock.expects("system").returns(0) #always execute successfully
assert os.system("not exist this command") == 0

mock.expects("getenv").withArgs("SHELL").returns("/bin/bash") #always return same string
assert os.getenv("SHELL") == "/bin/bash"
```

Mock API

mock = sinon.mock(class/instance/module)

Creates a mock for the provided class/instance/module. Does not change it, but returns a mock object to set expectations on the it's methods.

expectation = mock.expects("function")

Overrides the provided function of the mock object and returns it. See *Expectations API* below.

mock.restore()

Restores all mocked methods.

mock.verify()

Verifies all expectations on the mock. If any expectation is not satisfied, it will return false.

Expectations API

1. The constructor of expectation is as same as Spy and Stub.
2. All the expectation methods return the expectation, meaning you can chain them. Typical usage is below.

```
mock = sinon.mock(os).expects("system").atLeast(2).atMost(5)
os.system("ls")
assert not mock.verify()
os.system("ls")
assert mock.verify()
```

expectation.atLeast(number)

Specify the minimum amount of calls expected.

```
mock = sinon.mock(os).expects("system").atLeast(1)
assert not mock.verify()
os.system("ls")
assert mock.verify()
```

expectation.atMost(number)

Specify the maximum amount of calls expected.

```
mock = sinon.mock(os).expects("system").atMost(1)
assert mock.verify()
os.system("ls")
os.system("ls")
assert not mock.verify()
```

expectation.never()

Expect the method to never be called.

```
mock = sinon.mock(os).expects("system").never()
assert mock.verify()
os.system("ls")
assert not mock.verify()
```

expectation.once()

Expect the method to be called exactly once.

```
mock = sinon.mock(os).expects("system").once()
assert not mock.verify()
os.system("ls")
assert mock.verify()
os.system("ls")
assert not mock.verify()
```

expectation.twice()

Expect the method to be called exactly twice.

expectation.thrice()

Expect the method to be called exactly thrice.

expectation.exactly(number)

Expect the method to be called exactly number times.

expectation.withArgs(*args, **kwargs)

Expect the method to be called with the provided arguments and possibly others.

```
mock = sinon.mock(os).expects("getenv").withArgs("SHELL")
assert not mock.verify()
os.getenv("SHELL")
assert mock.verify()
```

expectation.withExactArgs(*args, **kwargs)

Expect the method to be called with the provided arguments and no others.

```
mock = sinon.mock(os).expects("getenv").withExactArgs("SHELL", "/bin/bash")
assert not mock.verify()
os.getenv("SHELL")
assert not mock.verify()
os.getenv("SHELL", "/bin/bash")
assert mock.verify()
```

expectation.restore()

Restores current mocked method

```
mock = sinon.mock(os)
expectation = mock.expects("system").returns("stub")
assert os.system("pwd") == "stub"
expectation.restore()
assert os.system("pwd") == 0
```

expectation.verify()

Verifies the expectation and returns false if it's not met.

```
mock = sinon.mock(os)
expectation_system = mock.expects("system").once()
expectation_getenv = mock.expects("getenv").once()
os.system("pwd")
assert not mock.verify()
assert expectation_system.verify()
```

Assertions

Sinon.PY ships with a set of assertions that mirror most behavior verification. The advantage of using the assertions is that failed expectations on stubs and spies can be expressed directly as assertion failures with detailed and helpful error messages.

To make sure assertions integrate nicely with your test framework, you should customize sinon.assert.fail.

The assertions can be used with either spies or stubs.

Assertion API

sinon.assertion.fail(message)

Setting error message when assert failed, by default the error message is empty.

```
spy = sinon.spy()
sinon.assertion.fail("expected exception message")
sinon.assertion.called(spy)
```

```
"""
Traceback (most recent call last):
  ...
AssertionError: expected exception message
"""
```

sinon.assertion.failException

The exception when assert failed, by default the exception is "AssertError".

```
spy = sinon.spy()
sinon.assertion.failException = Exception
sinon.assertion.called(spy)
"""
Traceback (most recent call last):
  ...
Exception
"""
```

sinon.assertion.notCalled(spy)

Passes if spy was never called.

```
spy = sinon.spy(os, "system")
sinon.assertion.notCalled(spy)
```

sinon.assertion.called(spy)

Passes if spy was called at least once.

```
spy = sinon.spy(os, "system")
os.system("pwd")
sinon.assertion.called(spy)
```

sinon.assertion.calledOnce(spy)

Passes if spy was called once and only once.

sinon.assertion.calledTwice()

Passes if spy was called exactly twice.

sinon.assertion.calledThrice()

Passes if spy was called exactly three times.

sinon.assertion.callCount(spy, num)

Passes if the spy was called exactly num times.

```
spy = sinon.spy()
sinon.assertion.callCount(spy, 0)
spy()
sinon.assertion.callCount(spy, 1)
```

sinon.assertion.callOrder(spy1, spy2, ...)

Passes if the provided spies where called in the specified order.

```
spy1 = sinon.spy()
spy2 = sinon.spy()
spy3 = sinon.spy()

spy1()
```

```
spy2()
spy3()
sinon.assertion.callOrder(spy1, spy2, spy3)
sinon.assertion.callOrder(spy2, spy3)
sinon.assertion.callOrder(spy1, spy3)
sinon.assertion.callOrder(spy1, spy2)

spy1()
sinon.assertion.callOrder(spy1, spy1)
sinon.assertion.callOrder(spy3, spy1)
sinon.assertion.callOrder(spy2, spy1)
sinon.assertion.callOrder(spy2, spy3, spy1)
sinon.assertion.callOrder(spy1, spy2, spy3, spy1)
```

sinon.assertionion.calledWith(spy, *args, **kwargs)

Passes if the spy was called with the provided arguments.

```
spy = sinon.spy(os, "system")
os.system("pwd")
sinon.assertion.calledWith(spy, "pwd")
```

sinon.assertion.alwaysCalledWith(spy, *args, **kwargs)

Passes if the spy was always called with the provided arguments.

```
spy = sinon.spy(os, "system")
os.system("pwd")
sinon.assertion.alwaysCalledWith(spy, "pwd") #pass
os.system("ls")
sinon.assertion.alwaysCalledWith(spy, "pwd") #fail
```

sinon.assertion.neverCalledWith(spy, *args, **kwargs)

Passes if the spy was never called with the provided arguments.

```
spy = sinon.spy(os, "system")
os.system("pwd")
sinon.assertion.neverCalledWith(spy, "ls")
```

sinon.assertion.calledWithExactly(spy, *args, **kwargs)

Passes if the spy was called with the provided arguments and no others.

```
spy = sinon.spy(os, "getenv")
os.getenv("NOT_EXIST_ENV_VAR", "DEFAULT_VALUE")
sinon.assertion.calledWithExactly(spy, "NOT_EXIST_ENV_VAR", "DEFAULT_VALUE") #pass
sinon.assertion.calledWithExactly(spy, "NOT_EXIST_ENV_VAR") #fail
```

sinon.assertion.alwaysCalledWithExactly(spy, *args, **kwargs)

Passes if the spy was always called with the provided arguments and no others.

```
spy = sinon.spy(os, "getenv")
os.getenv("NOT_EXIST_ENV_VAR", "DEFAULT_VALUE")
sinon.assertion.alwaysCalledWithExactly(spy, "NOT_EXIST_ENV_VAR", "DEFAULT_VALUE") #pass
os.getenv("NOT_EXIST_ENV_VAR", "ANOTHER_VALUE")
sinon.assertion.alwaysCalledWithExactly(spy, "NOT_EXIST_ENV_VAR", "DEFAULT_VALUE") #fail
```

sinon.assertion.calledWithMatch(spy, *args, **kwargs)

Passes if the spy was called with matching arguments. This behaves the same as sinon.assertion.calledWith(spy, sinon.match(arg1), sinon.match(arg2), ...).

```
spy = sinon.spy(os, "system")
os.system("pwd")
sinon.assertion.calledWithMatch(spy, str)
sinon.assertion.calledWith(spy, sinon.match(str))
```

sinon.assertion.alwaysCalledWithMatch(spy, *args, **kwargs)

Passes if the spy was always called with matching arguments. This behaves the same as sinon.assertion.alwaysCalledWith(spy, sinon.match(arg1), sinon.match(arg2), ...).

```
spy = sinon.spy(os, "system")
os.system("pwd")
sinon.assertion.alwaysCalledWithMatch(spy, str)
os.system("ls")
sinon.assertion.alwaysCalledWithMatch(spy, str)
```

sinon.assertion.neverCalledWithMatch(spy, *args, **kwargs)

Passes if the spy was never called with matching arguments. This behaves the same as sinon.assertion.neverCalledWith(spy, sinon.match(arg1), sinon.match(arg2), ...).

```
spy = sinon.spy(os, "system")
os.system("pwd")
sinon.assertion.neverCalledWithMatch(spy, int)
```

sinon.assertion.threw(spy, exception=None)

Passes if the spy threw the given exception. If only one argument is provided, the assertion passes if the spy ever threw any exception.

```
spy = sinon.spy(os, "getenv")
try:
    os.getenv(1000000000)
except:
    sinon.assertion.threw(spy, TypeError)
```

sinon.assertion.alwaysThrew(spy, exception=None)

Like above, only required for all calls to the spy.

Matchers

Matchers allow to be either more fuzzy or more specific about the expected value. Matchers can be passed as arguments to spy.calledWith and the corresponding sinon.assert functions. *Generally, there is no need to use Matcher directly.*

=> stub.withArgs, spy.returned is not supported

Matcher API

sinon.match(number)

Requires the value to be == to the given number.

```
match_int = sinon.match(1)
assert match_int.mtest(1)
```

sinon.match(string, strcmp="substring")

Requires the value to be a string and have the expectation as a substring,

```
match_substr = sinon.match("a long string", strcmp="substring")
assert match_substr.mtest("str")
```

sinon.match(regex, strcmp="regex")

Requires the value to be a string and match the given regular expression.

```
match_substr = sinon.match("(\d*)-(\d*)", strcmp="regex")
assert match_substr.mtest("0000-0000")
```

sinon.match(function, is_custom_func=True)

See *Custom matchers*.

sinon.match(ref)

For anything not belongs to above, the argument will be processed as a value (usually, using sinon.match.same to compare).

```
match_int = sinon.match(int)
assert match_int.mtest(1)
match_str = sinon.match(str)
assert match_str.mtest("str")
```

sinon.match.any

Matches anything.

```
match_any = sinon.match.any
assert match_any.mtest(None)
assert match_any.mtest(123)
assert match_any.mtest("123")
assert match_any.mtest(os)
assert match_any.mtest(os.system)
```

sinon.match.defined

Requires the value which is not None.

```
match_defined = sinon.match.defined
assert not match_defined.mtest(None)
assert match_defined.mtest([])
assert match_defined.mtest(1)
```

sinon.match.truthy

Requires the value to be truthy.

```
match_truthy = sinon.match.truthy
assert match_truthy.mtest(True)
assert match_truthy.mtest(1)
assert not match_truthy.mtest(False)
assert not match_truthy.mtest(0)
```

sinon.match.falsy

Requires the value to be falsy.

```
match_falsy = sinon.match.falsy
assert not match_falsy.mtest(True)
assert not match_falsy.mtest(1)
assert match_falsy.mtest(False)
assert match_falsy.mtest(0)
```

sinon.match.bool

Requires the value to be a boolean.

```
match_bool = sinon.match.bool
assert match_bool.mtest(True)
assert not match_bool.mtest(1)
assert match_bool.mtest(False)
assert not match_bool.mtest(0)
```

sinon.match.same(ref)

Requires the value to strictly equal ref.

sinon.match.typeOf(type)

Requires the value to be a type of the given type.

```
match_type = sinon.match.typeOf(int)
assert match_type.mtest(1)
assert not match_type.mtest(True)
```

sinon.match.instanceOf(instance)

Requires the value to be an instance of the given instance.

```
spy = sinon.spy()
stub = sinon.stub()
match_inst = sinon.match.instanceOf(spy)
assert match_inst.mtest(stub) #True because stub inherits spy
```

Combining matchers

All matchers implement *and* and *or*. This allows to logically combine two matchers. The result is a new matchers that requires both (and) or one of the matchers (or) to return true.

and_match(another_matcher)

```
spy = sinon.spy()
stub = sinon.stub()
expectation = sinon.mock(os).expects("system")
match_and = sinon.match.instanceOf(spy).and_match(sinon.match.instanceOf(stub))
assert match_and.mtest(expectation) #True because expectation inherits spy and stub
```

or_match(another_matcher)

```
match_or = sinon.match(int).or_match(sinon.match(str))
assert match_or.mtest(1)
assert match_or.mtest("1")
```

Custom matchers

Custom matchers are created with the *sinon.match* factory which takes a test. The test function takes a value as the only argument, returns *true* if the value matches the expectation and *false* otherwise.

```
def equal_to_square(give_value, expected_value):
    return True if give_value**2 == expected_value else False

match_custom = sinon.match(equal_to_square, is_custom_func=True)
assert not match_custom.mtest(6, 49)
assert match_custom.mtest(6, 36)
```

Sandboxes

Sandbox will make each test case isolated.

By default the properties of spy, stub and mock(expectation) of the sandbox is bound to whatever object the function is run on, so if you don't want to manually restore(), you can use decorator(@sinon.test) to wrap the test function.

.restore()

All inspectors in sinon do not allow multiple wrapping. For example:

```
>>> spy = sinon.spy(os, "system")
>>> stub = sinon.stub(os, "system")
```

This will cause an exception:

```
Exception: [system] have already been declared
```

Therefore, for making test cases work, after finishing the mission of that inspector, it should be restored mannually by .restore()

```
>>> spy = sinon.spy(os, "system")
>>> spy.restore()
>>> stub = sinon.stub(os, "system")
>>> stub.restore()
```

Sandbox API

decorator: sinon.test

Using restore in the end of each testcase makes code size huge. For solving this problem, sandbox is a good solution. Below is a fully example about using sandbox of Sinon.PY. In this example, there is no need to call .restore() anymore, sinon.test will automatically clean all inspectors in each test cases.

```
import os
import sinon

@sinon.test
def test_os_system_ls():
    spy = sinon.spy(os, "system")
    os.system("ls")
    assert spy.called
```

```
@sinon.test
def test_os_system_pwd():
    spy = sinon.spy(os, "system")
    os.system("pwd")
    assert spy.called

test_os_system_ls()
test_os_system_pwd()
```

Scopes

Why should use scope?

In a general unittest, the test function will import other class or module and test them. However, there are some exceptional possibilities for testing functions in the same module/class/function level.

For example

```
import sinon

def a_function_of_test():
    pass

def test_func():

    spy = sinon.spy(a_function_of_test)
    assert not spy.called
    a_function_of_test()
    assert spy.called

test_func()
```

In this case, a_function_of_test is not wrapped successfully. Because the scope is not able to be inspected.

```
AttributeError: 'NoneType' object has no attribute 'a_function_of_test'
```

Scope API

sinon.init(scope)

For getting a inspectable scope, passing globals()/locals() as an argument into .init()

For inspecting the function, using the return scope to call the inspected function instead of calling original function directly.

Example1: globals()

```
import sinon

def a_global_function_in_test():
    pass

def test_func():
    scope = sinon.init(globals())
    spy = sinon.spy(a_global_function_in_test)
    assert not spy.called
```

```
    scope.a_global_function_in_test()
    assert spy.called

test_func()
```

Example2: locals()

```
import sinon

def test_func():

    def a_local_function_in_test():
        pass

    scope = sinon.init(locals())
    spy = sinon.spy(a_local_function_in_test)
    assert not spy.called
    scope.a_local_function_in_test()
    assert spy.called

test_func()
```

Practices

With unittest framework

```
import unittest
import sinon

class GlobalCls(object):
    def clsFunc(self):
        return "A"

def localFunc():
    return "B"

class TestExample(unittest.TestCase):

    def setUp(self):
        global g
        g = sinon.init(globals())

    @sinon.test
    def test001(self):
        import os
        spy_system = sinon.spy(os, "system")
        os.system("ls")
        self.assertTrue(spy_system.called)

    @sinon.test
    def test002(self):
        spy_global_cls = sinon.spy(GlobalCls, "clsFunc")
        gc = GlobalCls()
        gc.clsFunc()
        self.assertTrue(spy_global_cls.called)
```

```
    @sinon.test
    def test003(self):
        stub_local_func = sinon.stub(localFunc)
        stub_local_func.returns("A")
        self.assertEqual(g.localFunc(), "A")
```

Above python2.7, it could be executed by commands below:

```
$ python -m unittest [test_file_name]
```

An example of restful_flask application

Please refer the link to see the project

CHAPTER 3

New Ideas & Issues

Please send any defects, issues, feature requests to the Github. I really appreciate people to make Sinon.PY better, easier to work with.