doublex

Release 1.8.1

David Villa Alises

November 14, 2016

Contents

Powerful test doubles framework for Python

[install | docs | changelog | sources | issues | PyPI | github clone | travis]

CHAPTER 1

a trivial example

```
import unittest
from doublex import Spy, assert_that, called

class SpyUseExample(unittest.TestCase):
    def test_spy_example(self):
        # given
        spy = Spy(SomeCollaboratorClass)
        cut = YourClassUnderTest(spy)

        # when
        cut.a_method_that_call_the_collaborator()

        # then
        assert_that(spy.some_method, called())
```

See more about doublex doubles.

CHAPTER 2

Features

- doubles have not public API framework methods. It could cause silent misspelling.
- doubles do not require collaborator instances, just classes, and it never instantiate them.
- `assert_that()` is used for ALL assertions.
- mock invocation order is relevant by default.
- supports old and new style classes.
- **supports Python versions: 2.6, 2.7, 3.3, 3.4, 3.5**

2.1 Debian

- official package (may be outdated)
- amateur repository: `deb http://pike.esi.uclm.es/arco/ sid main` (always updated)
- official ubuntu package
- debian dir: `svn://svn.debian.org/svn/python-modules/packages/doublex/trunk`

CHAPTER 3

related

- slides
- pyDoubles
- doublex-expects
- crate
- other doubles
- ludibrio
- doubles

CHAPTER 4

User's Guide

4.1 Install

The simplest install method is to use `pip`:

```
$ sudo pip install doublex
```

Or you can download the last release tarball, available in the PyPI project:

```
$ gunzip doublex-X.X.tar.gz
$ tar xvf doublex-X.X.tar
$ cd doublex-X.X/
```

To install:

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

Or use pip:

```
$ sudo pip install doubles-X.X.tar.gz
```

Note: doublex depends on PyHamcrest, but if you use `pip` it is automatically installed too.

Also it is available as Debian and Ubuntu official packages, although may be outdated:

```
$ sudo apt-get install python-doublex
```

4.2 Doubles

Some very basic examples are shown below. Remember that test doubles are created to be invoked by your SUT, and a RealWorld™ test never directly invokes doubles. Here we do it that way, but just for simplicity.

4.2.1 Stub

Hint: *Stubs tell you what you wanna hear.*

A `Stub` is a double object that may be programmed to return specified values depending on method invocations and their arguments. You must use a context (the `with` keyword) for that.

Invocations over the `Stub` must meet the collaborator interface:

```
from doublex import Stub, ANY_ARG, assert_that, is_

class Collaborator:
    def hello(self):
        return "hello"

    def add(self, a, b):
        return a + b

with Stub(Collaborator) as stub:
    stub.hello().raises(SomeException)
    stub.add(ANY_ARG).returns(4)

assert_that(stub.add(2,3), is_(4))
```

If you call an nonexistent method you will get an `AttributeError` exception.

```
>>> with Stub(Collaborator) as stub:
...     stub.foo().returns(True)
Traceback (most recent call last):
...
AttributeError: 'Collaborator' object has no attribute 'foo'
```

Wrong argument number:

```
>>> with Stub(Collaborator) as stub:
...     stub.hello(1).returns(2)  # interface mismatch exception
Traceback (most recent call last):
...
TypeError: Collaborator.hello() takes exactly 1 argument (2 given)
```

"free" Stub

This allows you to invoke any method you want because it is not restricted to an interface.

```
from doublex import Stub, assert_that, is_

# given
with Stub() as stub:
    stub.foo('hi').returns(10)

# when
result = stub.foo('hi')

# then
assert_that(result, is_(10))
```

4.2.2 Spy

Hint: *Spies remember everything that happens to them.*

Spy extends the Stub functionality allowing you to assert on the invocation it receives since its creation.

Invocations over the Spy must meet the collaborator interface.

```
from hamcrest import contains_string
from doublex import Spy, assert_that, called

class Sender:
    def say(self):
        return "hi"

    def send_mail(self, address, force=True):
        pass  # [some amazing code]

sender = Spy(Sender)

sender.send_mail("john.doe@example.net")  # right, Sender.send_mail interface support this

assert_that(sender.send_mail, called())
assert_that(sender.send_mail, called().with_args("john.doe@example.net"))
assert_that(sender.send_mail, called().with_args(contains_string("@example.net")))

sender.bar()  # interface mismatch exception
```

```
Traceback (most recent call last):
...
AttributeError: 'Sender' object has no attribute 'bar'
```

```
>>> sender = Spy(Sender)
>>> sender.send_mail()
Traceback (most recent call last):
...
TypeError: Sender.send_mail() takes at least 2 arguments (1 given)
```

```
>>> sender = Spy(Sender)
>>> sender.send_mail(wrong=1)
Traceback (most recent call last):
...
TypeError: Sender.send_mail() got an unexpected keyword argument 'wrong'
```

```
>>> sender = Spy(Sender)
>>> sender.send_mail('foo', wrong=1)
Traceback (most recent call last):
...
TypeError: Sender.send_mail() got an unexpected keyword argument 'wrong'
```

"free" Spy

As the "free" Stub, this is a spy not restricted by a collaborator interface.

```
from doublex import Stub, assert_that

# given
with Spy() as sender:
    sender.helo().returns("OK")

# when
sender.send_mail('hi')
sender.send_mail('foo@bar.net')

# then
```

```
assert_that(sender.helo(), is_("OK"))
assert_that(sender.send_mail, called())
assert_that(sender.send_mail, called().times(2))
assert_that(sender.send_mail, called().with_args('foo@bar.net'))
```

ProxySpy

Hint: *Proxy spies forward invocations to its actual instance.*

The `ProxySpy` extends the `Spy` invoking the actual instance when the corresponding spy method is called

> **Warning:** Note the `ProxySpy` breaks isolation. It is not really a double. Therefore is always the worst double and the last resource.

```
from doublex import ProxySpy, assert_that

sender = ProxySpy(Sender())  # NOTE: It takes an instance (not class)

assert_that(sender.say(), is_("hi"))
assert_that(sender.say, called())

sender.say('boo!')  # interface mismatch exception
```

```
Traceback (most recent call last):
...
TypeError: Sender.say() takes exactly 1 argument (2 given)
```

4.2.3 Mock

Hint: *Mock forces the predefined script.*

Mock objects may be programmed with a sequence of method calls. Later, the double must receive exactly the same sequence of invocations (including argument values). If the sequence does not match, an AssertionError is raised. "free" mocks are provided too:

```
from doublex import Mock, assert_that, verify

with Mock() as smtp:
    smtp.helo()
    smtp.mail(ANY_ARG)
    smtp.rcpt("bill@apple.com")
    smtp.data(ANY_ARG).returns(True).times(2)

smtp.helo()
smtp.mail("poormen@home.net")
smtp.rcpt("bill@apple.com")
smtp.data("somebody there?")
smtp.data("I am afraid..")

assert_that(smtp, verify())
```

`verify()` asserts invocation order. If your test does not require strict invocation order just use `any_order_verify()` matcher instead:

```
from doublex import Mock, assert_that, any_order_verify

with Mock() as mock:
    mock.foo()
    mock.bar()

mock.bar()
mock.foo()

assert_that(mock, any_order_verify())
```

Programmed invocation sequence also may specify stubbed return values:

```
from doublex import Mock, assert_that

with Mock() as mock:
    mock.foo().returns(10)

assert_that(mock.foo(), is_(10))
assert_that(mock, verify())
```

4.3 Reference

4.3.1 assert_that()

assert() evaluates a boolean expression, but `assert_that()` takes an arbitrary object and a matcher, that is applied over the former. This way makes possible to build up complex assertions by means of matcher composition.

CAUTION: Be ware about `hamcrest.assert_that()`

Note the hamcrest.assert_that() function has two different behavior depending of its arguments:

- `assert_that(actual, matcher, [reason])` In this form the `matcher` is applied to the `actual` object. If the matcher fails, it raises AssertionError showing the optional reason.
- `assert_that(value, [reason])` If the boolean interpretation of `value` is False, it raises AssertionError showing the optional reason.

It implies that something like:

```
from hamcrest import assert_that

assert_that(foo, bar)
```

If `bar` is not a matcher, the assertion is satisfied for any non-false `foo`, independently of the value of `bar`. A more obvious example:

```
from hamcrest import assert_that

assert_that(2 + 2, 5)  # OMG! that assertion IS satisfied!
```

For this reason, when you need compare values (equivalent to the unit `assertEquals`) you must always use a matcher, like `is_` or `equal_to`:

```
assert_that(2 + 2, is_(5))       # that assertion is NOT satisfied!
assert_that(2 + 2, equal_to(5))  # that assertion is NOT satisfied!
```

Prefer `doublex.assert_that`

New in version 1.7: - (thanks to Eduardo Ferro)

To avoid the issues described in the previous section, doublex provides an alternative `assert_that()` implementation that enforces a matcher as second argument.

```
>>> from doublex import assert_that
>>> assert_that(1, 1)
Traceback (most recent call last):
...
MatcherRequiredError: 1 should be a hamcrest Matcher
```

4.3.2 Stubbing

The stub provides all methods in the collaborator interface. When the collaborator is not given (a free stub), the stub seems to have any method you invoke on it. The default behavior for non stubbed methods is to return `None`, although it can be changed (see *Changing default stub behavior*).

```
>>> from doublex import Stub
>>> stub = Stub()
>>> stub.method()
```

This behavior may be customized in each test using the Python context manager facility:

```
from doublex import Stub

with Stub() as stub:
     stub.method(<args>).returns(<value>)
```

Hamcrest matchers may be used to define amazing stub conditions:

```
from hamcrest import all_of, has_length, greater_than, less_than
from doublex import Stub, assert_that, is_

with Stub() as stub:
    stub.foo(has_length(less_than(4))).returns('<4')
    stub.foo(has_length(4)).returns('four')
    stub.foo(has_length(
              all_of(greater_than(4),
                     less_than(8)))).returns('4<x<8')
    stub.foo(has_length(greater_than(8))).returns('>8')

assert_that(stub.foo((1, 2)), is_('<4'))
assert_that(stub.foo('abcd'), is_('four'))
assert_that(stub.foo('abcde'), is_('4<x<8'))
assert_that(stub.foo([0] * 9), is_('>8'))
```

Stubs returning input

```
from doublex import Stub, assert_that

def test_returns_input(self):
    with Stub() as stub:
        stub.foo(1).returns_input()

    assert_that(stub.foo(1), is_(1))
```

Stubs raising exceptions

```
from doublex import Stub

def test_raises(self):
    with Stub() as stub:
        stub.foo(2).raises(SomeException)

    with self.assertRaises(SomeException):
        stub.foo(2)
```

Changing default stub behavior

New in version 1.7: - (thanks to Eduardo Ferro)

Any non-stubbed method returns `None`. But this behavior can be changed by means of `set_default_behavior()` function. It can be applied to any double class: `Stub`, `Spy`, `ProxySpy` or `Mock`.

```
from doublex import Stub, assert_that
from doublex import set_default_behavior, method_returning

set_default_behavior(Stub, method_returning(20))
stub  = Stub()
assert_that(stub.unknown(), is_(20))
```

Or to a specific instance:

```
from doublex import Stub, assert_that, is_
from doublex import set_default_behavior, method_returning

stub = Stub()
set_default_behavior(stub, method_returning(20))
assert_that(stub.unknown(), is_(20))
```

Also, it is possible to raise some exception:

```
>>> from doublex import Stub, set_default_behavior, method_raising
>>> stub = Stub()
>>> set_default_behavior(stub, method_raising(SomeException))
>>> stub.unknown()
Traceback (most recent call last):
...
SomeException
```

4.3.3 Asserting method calls

To assert method invocations you need a `Spy` and the `called()` matcher.

called()

`called()` matches method invocation (argument values are not relevant):

```
from doublex import Spy, assert_that, called

spy = Spy()

spy.m1()
spy.m2(None)
spy.m3("hi", 3.0)
spy.m4([1, 2])

assert_that(spy.m1, called())
assert_that(spy.m2, called())
assert_that(spy.m3, called())
assert_that(spy.m4, called())
```

with_args(): asserting calling argument values

Match explicit argument values:

```
from hamcrest import contains_string, less_than, greater_than
from doublex import Spy, assert_that, called

spy = Spy()

spy.m1()
spy.m2(None)
spy.m3(2)
spy.m4("hi", 3.0)
spy.m5([1, 2])
spy.m6(name="john doe")

assert_that(spy.m1, called())
assert_that(spy.m2, called())

assert_that(spy.m1, called().with_args())
assert_that(spy.m2, called().with_args(None))
assert_that(spy.m3, called().with_args(2))
assert_that(spy.m4, called().with_args("hi", 3.0))
assert_that(spy.m5, called().with_args([1, 2]))
assert_that(spy.m6, called().with_args(name="john doe"))
```

Remember that hamcrest matchers matchers are fully supported:

```
assert_that(spy.m3, called().with_args(less_than(3)))
assert_that(spy.m3, called().with_args(greater_than(1)))
assert_that(spy.m6, called().with_args(name=contains_string("doe")))
```

Other example with a string argument and combining several matchers:

```
from hamcrest import has_length, greater_than, less_than
from doublex import Spy, assert_that, called, never

spy = Spy()

spy.foo("abcd")

assert_that(spy.foo, called().with_args(has_length(4)))
assert_that(spy.foo, called().with_args(has_length(greater_than(3))))
```

```
assert_that(spy.foo, called().with_args(has_length(less_than(5))))
assert_that(spy.foo, never(called().with_args(has_length(greater_than(5)))))
```

anything(): asserting wildcard values

The `anything()` hamcrest matcher may be used to match any single value. That is useful when only some arguments are relevant:

```
from hamcrest import anything

spy.foo(1, 2, 20)
spy.bar(1, key=2)

assert_that(spy.foo, called().with_args(1, anything(), 20))
assert_that(spy.bar, called().with_args(1, key=anything()))
```

ANY_ARG: greedy argument value wildcard

`ANY_ARG` is a special value that matches any subsequent argument values, including no args. That is, `ANY_ANG` means "any value for any argument from here". If `anything()` is similar to the regular expression `?`, `ANY_ARG` would be equivalent to `*`.

For this reason, it has **no sense** to give other values or matchers after an `ANY_ARG`. It is also applicable to keyword arguments due they have no order. In summary, `ANY_ARG`:

- it must be the last positional argument value.
- it can not be given as keyword value.
- it can not be given together keyword arguments.

Since version 1.7 a `WrongAPI` exception is raised if that situations (see issue 9).

An example:

```
from doublex import ANY_ARG

spy.arg0()
spy.arg1(1)
spy.arg3(1, 2, 3)
spy.arg_karg(1, key1='a')

assert_that(spy.arg0, called())
assert_that(spy.arg0, called().with_args(ANY_ARG))  # equivalent to previous

assert_that(spy.arg1, called())
assert_that(spy.arg1, called().with_args(ANY_ARG))  # equivalent to previous

assert_that(spy.arg3, called().with_args(1, ANY_ARG))
assert_that(spy.arg_karg, called().with_args(1, ANY_ARG))
```

Also for stubs:

```
from doublex import Stub, assert_that, ANY_ARG, is_

with Stub() as stub:
    stub.foo(ANY_ARG).returns(True)
    stub.bar(1, ANY_ARG).returns(True)
```

```
assert_that(stub.foo(), is_(True))
assert_that(stub.foo(1), is_(True))
assert_that(stub.foo(key1='a'), is_(True))
assert_that(stub.foo(1, 2, 3, key1='a', key2='b'), is_(True))

assert_that(stub.foo(1, 2, 3), is_(True))
assert_that(stub.foo(1, key1='a'), is_(True))
```

with_some_args(): asserting just relevant arguments

New in version 1.7.

When a method has several arguments and you need to assert an invocation giving a specific value just for some of them, you may use the `anything()` matcher for the rest of them. That works but the resulting code is a bit dirty:

```
from hamcrest import anything
from doublex import Spy, assert_that, ANY_ARG

class Foo:
    def five_args_method(self, a, b, c, d, e=None):
        return 4

spy = Spy(Foo)
spy.five_args_method(1, 2, 'bob', 4)

# only the 'c' argument is important in the test
assert_that(spy.five_args_method,
            called().with_args(anything(), anything(), 'bob', anything()))
# assert only 'b' argument
assert_that(spy.five_args_method,
            called().with_args(anything(), 2, ANY_ARG))
```

The `with_some_args()` allows to specify just some arguments, assuming all other can take any value. The same example using `with_some_arg()`:

```
from doublex import Spy, assert_that, called

class Foo:
    def five_args_method(self, a, b, c, d, e=None):
        return 4

spy = Spy(Foo)
spy.five_args_method(1, 2, 'bob', 4)

# only the 'c' argument is important in the test
assert_that(spy.five_args_method,
            called().with_some_args(c='bob'))
# assert only 'b' argument
assert_that(spy.five_args_method,
            called().with_some_args(b=2))
```

This method may be used with both keyword and non-keyword arguments.

> **Warning:** Formal argument name is mandatory, so this is only applicable to restricted spies (those that are instantiated giving a collaborator).

never()

Convenient replacement for `hamcrest.is_not()`:

```
from hamcrest import is_not
from doublex import Spy, assert_that, called, never

spy = Spy()

assert_that(spy.m5, is_not(called()))  # is_not() works
assert_that(spy.m5, never(called()))   # but prefer never() due to better error report messages
```

times(): asserting number of calls

```
from hamcrest import anything, all_of, greater_than, less_than
from doublex import Spy, assert_that, called, ANY_ARG, never

spy = Spy()

spy.foo()
spy.foo(1)
spy.foo(1)
spy.foo(2)

assert_that(spy.unknown, never(called()))                    # = 0 times
assert_that(spy.foo, called())                               # > 0
assert_that(spy.foo, called().times(greater_than(0)))        # > 0 (same)
assert_that(spy.foo, called().times(4))                      # = 4
assert_that(spy.foo, called().times(greater_than(2)))        # > 2
assert_that(spy.foo, called().times(less_than(6)))           # < 6

assert_that(spy.foo, never(called().with_args(5)))                    # = 0 times
assert_that(spy.foo, called().with_args().times(1))                   # = 1
assert_that(spy.foo, called().with_args(anything()))                  # > 0
assert_that(spy.foo, called().with_args(ANY_ARG).times(4))            # = 4
assert_that(spy.foo, called().with_args(1).times(2))                  # = 2
assert_that(spy.foo, called().with_args(1).times(greater_than(1)))    # > 1
assert_that(spy.foo, called().with_args(1).times(less_than(5)))       # < 5
assert_that(spy.foo, called().with_args(1).times(
            all_of(greater_than(1), less_than(8))))                   # 1 < times < 8
```

4.4 Ad-hoc stub methods

Create a standalone stub method directly over any instance (even no doubles), with *method_returning()* and *method_raising()*:

```
from doublex import method_returning, method_raising, assert_that

collaborator = Collaborator()
collaborator.foo = method_returning("bye")
assert_that(collaborator.foo(), is_("bye"))

collaborator.foo = method_raising(SomeException)
collaborator.foo()  # raises SomeException
```

```
Traceback (most recent call last):
...
SomeException
```

4.5 Properties

doublex supports stub and spy properties in a pretty easy way in relation to other frameworks like python-mock.

That requires two constraints:

- It does not support "free" doubles. ie: you must give a collaborator in the constructor.
- collaborator must be a new-style class. See the next example.

4.5.1 Stubbing properties

```
from doublex import Spy, assert_that, is_

with Spy(Collaborator) as spy:
    spy.prop = 2  # stubbing 'prop' value

assert_that(spy.prop, is_(2))  # double property getter invoked
```

4.5.2 Spying properties

Continuing previous example:

```
from doublex import Spy, assert_that, never
from doublex import property_got, property_set

class Collaborator(object):
    @property
    def prop(self):
        return 1

    @prop.setter
    def prop(self, value):
        pass

spy = Spy(Collaborator)
value = spy.prop

assert_that(spy, property_got('prop'))  # property 'prop' was read.

spy.prop = 4
spy.prop = 5
spy.prop = 5

assert_that(spy, property_set('prop'))  # was set to any value
assert_that(spy, property_set('prop').to(4))
assert_that(spy, property_set('prop').to(5).times(2))
assert_that(spy, never(property_set('prop').to(6)))
```

4.5.3 Mocking properties

Getting property:

```
from doublex import Mock, assert_that, verify

with Mock(Collaborator) as mock:
    mock.prop

mock.prop

assert_that(mock, verify())
```

Setting property:

```
from doublex import Mock, assert_that, verify

with Mock(Collaborator) as mock:
    mock.prop = 5

mock.prop = 5

assert_that(mock, verify())
```

Using matchers:

```
from hamcrest import all_of, greater_than, less_than
from doublex import Mock, assert_that, verify

with Mock(Collaborator) as mock:
    mock.prop = all_of(greater_than(8), less_than(12))

mock.prop = 10

assert_that(mock, verify())
```

4.6 Stub delegates

The value returned by the stub may be delegated from a function, method or other callable...

```
def get_user():
    return "Freddy"

with Stub() as stub:
    stub.user().delegates(get_user)
    stub.foo().delegates(lambda: "hello")

assert_that(stub.user(), is_("Freddy"))
assert_that(stub.foo(), is_("hello"))
```

It may be delegated from iterables or generators too!:

```
with Stub() as stub:
    stub.foo().delegates([1, 2, 3])

assert_that(stub.foo(), is_(1))
```

```
assert_that(stub.foo(), is_(2))
assert_that(stub.foo(), is_(3))
```

4.7 Stub observers

Stub observers allow you to execute extra code (similar to python-mock "side effects", but easier):

```
class Observer(object):
    def __init__(self):
        self.state = None

    def notify(self, *args, **kargs):
        self.state = args[0]

observer = Observer()
stub = Stub()
stub.foo.attach(observer.notify)
stub.foo(2)

assert_that(observer.state, is_(2))
```

4.8 Mimic doubles

Usually double instances behave as collaborator surrogates, but they do not expose the same class hierarchy, and usually this is pretty enough when the code uses "duck typing":

```
class A(object):
    pass

class B(A):
    pass
```

```
>>> from doublex import Spy
>>> spy = Spy(B())
>>> isinstance(spy, Spy)
True
>>> isinstance(spy, B)
False
```

But some third party library DOES strict type checking using `isinstance()`. That invalidates our doubles. For these cases you can use Mimic's. Mimic class decorates any double class to achieve full replacement instances (Liskov principle):

```
>>> from doublex import Stub, Mimic
>>> spy = Mimic(Spy, B)
>>> isinstance(spy, B)
True
>>> isinstance(spy, A)
True
>>> isinstance(spy, Spy)
True
>>> isinstance(spy, Stub)
True
```

```
>>> isinstance(spy, object)
True
```

4.9 Asynchronous spies

New in version 1.5.1.

Sometimes interaction among your SUT class and their collaborators does not meet a synchronous behavior. That may happen when the SUT performs collaborator invocations in a different thread, or when the invocation pass across a message queue, publish/subscribe service, etc.

Something like that:

```
class Collaborator(object):
    def write(self, data):
        print "your code here"

class SUT(object):
    def __init__(self, collaborator):
        self.collaborator = collaborator

    def some_method(self):
        thread.start_new_thread(self.collaborator.write, ("something",))
```

If you try to test your collaborator is called using a Spy, you will get a wrong behavior:

```
# THE WRONG WAY
class AsyncTests(unittest.TestCase):
    def test_wrong_try_to_test_an_async_invocation(self):
        # given
        spy = Spy(Collaborator)
        sut = SUT(spy)

        # when
        sut.some_method()

        # then
        assert_that(spy.write, called())
```

due to the `called()` assertion may happen before the `write()` invocation, although not always...

You may be tempted to put a sleep before the assertion, but this is a bad solution. A right way to solve that issue is to use something like a barrier. The threading.Event may be used as a barrier. See this new test version:

```
# THE DIRTY WAY
class AsyncTests(unittest.TestCase):
    def test_an_async_invocation_with_barrier(self):
        # given
        barrier = threading.Event()
        with Spy(Collaborator) as spy:
            spy.write.attach(lambda *args: barrier.set)

        sut = SUT(spy)

        # when
        sut.some_method()
        barrier.wait(1)
```

```
        # then
        assert_that(spy.write, called())
```

The `spy.write.attach()` is part of the doublex stub-observer mechanism, a way to run arbitrary code when stubbed methods are called.

That works because the `called()` assertion is performed only when the spy releases the barrier. If the `write()` invocation never happens, the `barrier.wait()` continues after 1 second but the test fails, as must do. When all is right, the barrier waits just the required time.

Well, this mechanism is a doublex builtin (the `async` matcher) since release 1.5.1 providing the same behavior in a clearer way. The next is functionally equivalent to the listing just above:

```
# THE DOUBLEX WAY
class AsyncTests(unittest.TestCase):
    def test_test_an_async_invocation_with_doublex_async(self):
        # given
        spy = Spy(Collaborator)
        sut = SUT(spy)

        # when
        sut.some_method()

        # then
        assert_that(spy.write, called().async(timeout=1))
```

4.10 Inline stubbing and mocking

New in version 1.8: - Proposed by Carlos Ble

Several pyDoubles users ask for an alternative to set stubbing and mocking in a way similar to pyDoubles API, that is, instead of use the double context manager:

```
with Stub() as stub:
    stub.foo(1).returns(100)

with Mock() as mock:
    mock.bar(2).returns(50)
```

You may invoke the `when()` and `expect_call()` functions to get the same setup.

```
stub = Stub()
when(stub).foo(1).returns(100)

mock = Mock()
expect_call(mock).bar(2).returns(50)
```

Note that `when()` and `expect_call()` internally provide almost the same functionality. Two functions are provided only for test readability purposes. `when()` is intented for stubs, spies and proxyspies, and `expect_call()` is intented for mocks.

4.11 calls: low-level access to invocation records

New in version 1.6.3.

Invocation over spy methods are available in the `calls` attribute. You may use that to get invocation argument values and perform complex assertions (i.e: check invocations arguments were specific instances). However, you should prefer `called()` matcher assertions over this. An example:

```
from doublex import Spy, assert_that, ANY_ARG, is_

class TheCollaborator(object):
    def method(self, *args, **kargs):
        pass

with Spy(TheCollaborator) as spy:
    spy.method(ANY_ARG).returns(100)

spy.method(1, 2, 3)
spy.method(key=2, val=5)

assert_that(spy.method.calls[0].args, is_((1, 2, 3)))
assert_that(spy.method.calls[1].kargs, is_(dict(key=2, val=5)))
assert_that(spy.method.calls[1].retval, is_(100))
```

4.12 API

4.12.1 Double classes

class **Stub**([*collaborator*])

class **Spy**([*collaborator*])

class **ProxySpy**([*collaborator*])

class **Mock**([*collaborator*])

class **Mimic**(*double*, *collaborator*)

4.12.2 Stubbing

`Method.`**`raises`**(*exception*)

Stub method will raise the given *exception* when invoked. Method parameters are relevant, and they may be literal values or hamcrest matchers:

```
    with Stub() as stub:
        stub.method().raises(ValueError)
```

See *Stubs raising exceptions*.

`Method.`**`returns`**(*value*)

Stub method will return the given value when invoked:

```
    with Stub() as stub:
        stub.method().returns(100)
```

See *Stub*.

`Method.`**`returns_input`**()

Stub method will return input parameters when invoked:

```
with Stub() as stub:
    stub.method().returns_input()
```

See *Stubs returning input*.

`Method.`**`delegates`**(*delegate*)
Stub method will return values generated by the *delegate*, that may be a function, generator or iterable object:

```
with Stub() as stub:
    stub.method().delegates([1, 2, 4, 8])
```

See *Stub delegates*.

`Method.`**`attach`**(*callable*)
Stub methods are observable. You may attach arbitrary callable that will be invoked any time the stub method does:

```
counter = itertools.count()
stub.method.attach(counter.next)
```

See *Stub observers*.

4.12.3 Matchers

class `never`(*matcher*)
Just a cosmetic alias to the hamcrest matcher `is_not()`. See *never()*.

for Spy methods

class `called`
Asserts a spy method was called:

```
assert_that(spy.method, called())
```

See *called()*.

`called.`**`async`**(*timeout*)
The `called` assertion waits the corresponding invocation a maximum of *timeout* seconds.

Parameters **`timeout`** (*int*) – how many second wait before assume assertion fails.

```
assert_that(spy.method, called().async(1))
```

See *Asynchronous spies*.

`called.`**`times`**(*value*)
The spy method must be invoked *value* times to consider the assertion right. The *value* parameter may an integer or hamcrest matcher as well.

Parameters **`value`** (*int or hamcrest Matcher*) – how many times the method should be called.

```
assert_that(spy.method, called().times(less_that(3)))
```

See *times(): asserting number of calls*.

`called.`**`with_args`**(**args*, ***kargs*)
The spy method must be invoked with the given positional or/and named parameters. All of them may be literal values and hamcrest matchers.

```
assert_that(spy.method, called().with_args("mary", greater_that(4)))
```

See *with_args(): asserting calling argument values*.

called.**with_some_args**(***kargs*)
The spy method must be invoked with AT LEAST the given parameter values. It supports literal values and hamcrest matchers.

```
assert_that(spy.method, called().with_some_args(name="mary"))
```

See *with_some_args(): asserting just relevant arguments*.

for properties

class property_got

class property_set

property_set.**to**(*value*)

See *Properties*.

for mocks

class verify
Checks the given mock meets the given expectations.

```
assert_that(mock, verify())
```

See *Mock*.

class any_order_verify
Checks the given mock meets the given expectations even when the invocation sequence has a different order to the expectations.

```
assert_that(mock, any_order_verify())
```

See *Mock*.

4.12.4 Module level functions

assert_that(*item*, *matcher*)
A convenient replace for the hamcrest *assert_that* method. See *assert_that()*.

wait_that(*item*, *matcher*, *reason=''*, *delta=1*, *timeout=5*)
It test the *matcher* over *item* until it matches or fails after *timemout* seconds, polling the matcher each *delta* seconds.

method_returning(*value*)
Creates an independent Stub method that returns the given value. It may be added to any object:

```
some.method = method_returning(20)
```

See *Ad-hoc stub methods*.

method_raising()
Creates an independent Stub method that raises the given exception. It may be added to any object:

```
    some.method = method_raising(ValueError)
```

See *Ad-hoc stub methods*.

set_default_behavior()
Set the default behavior for undefined Stub methods. The built-in behavior is to return **None**. See *Changing default stub behavior*.

4.13 pyDoubles

doublex started as a attempt to improve and simplify the codebase and API of the pyDoubles framework (by Carlos Ble).

Respect to pyDoubles, doublex has these features:

- Just hamcrest matchers (for all features).
- Only ProxySpy requires an instance. Other doubles accept a class too, but they never instantiate it.
- Properties may be stubbed and spied.
- Stub observers: Notify arbitrary hooks when methods are invoked. Useful to add "side effects".
- Stub delegates: Use callables, iterables or generators to create stub return values.
- Mimic doubles: doubles that inherit the same collaborator subclasses. This provides full LSP for code that make strict type checking.

doublex solves all the issues and supports all the feature requests notified in the pyDoubles issue tracker:

- assert keyword argument values
- assert that a method is called exactly one
- mocks impose invocation order
- doubles have framework public API
- stubs support keyword positional arguments

And some other features requested in the user group:

- doubles for properties
- creating doubles without instantiating real class
- using hamcrest with kwargs

doublex provides the pyDoubles API as a wrapper for easy transition to doublex for pyDoubles users. However, there are small differences. The bigger diference is that pyDoubles matchers are not supported anymore, although you may get the same feature using standard hamcrest matchers. Anyway, formally provided pyDoubles matchers are available as hamcrest aliases.

doublex supports all the pyDoubles features and some more that can not be easily backported. If you are a pyDoubles user you can run your tests using doublex.pyDoubles module. However, we recommed the native doublex API for your new developments.

In most cases the only required change in your code is the import sentence, that change from:

```
import pyDoubles.framework.*
```

to:

```
from doublex.pyDoubles import *
```

See the old pyDoubles documentation at http://pydoubles.readthedocs.org (that was formerly available in the pydoubles.org site).

4.14 Release notes / Changelog

4.14.1 doublex 1.8.3

- Fixed issue 25 Python 3.5 type hints support. See test.
- Fixed issue 23 Several tests failing because hamcrest.core.string_description.StringDescription is not a string anymore.

4.14.2 doublex 1.8.2

- Fixed issue 12. `returns_input()` now may manage several parameters. See test.
- Fixed issue 21. *`method_returning()`* and *`method_raising()`* are now spies. See test.
- Fixed issue 22. See test.
- `delegates()` now accepts dictionaries. See test.

4.14.3 doublex 1.8

- NEW inline stubbing and mocking with `when()` and `expect_call()`. See doc and tests.
- Added support for mocking properties. See doc and tests.
- Testing with tox for Python 2.6, 2.7, 3.2 and 3.3.
- Documentation now at http://python-doublex.readthedocs.org

4.14.4 doublex 1.7.2

- Added support for varargs methods (*args, **kargs). Fixes issue 14.
- NEW tracer mechanism to log double invocations. See test.
- NEW module level `wait_that()` function.

4.14.5 doublex 1.7

- NEW `with_some_args()` matcher to specify just relevant argument values in spy assertions. See doc and tests.
- NEW module level `set_default_behavior()` function to define behavior for non stubbed methods. Thanks to Eduardo Ferro. See doc and tests.

4.14.6 doublex 1.6.6

- bug fix update: Fixes issue 11.

4.14.7 doublex 1.6.5

- bug fix update: Fixes issue 10.

4.14.8 doublex 1.6.4

- Asynchronous spy assertion race condition bug fixed.
- Reading double attributes returns collaborator.class attribute values by default.

4.14.9 doublex 1.6.2

- Invocation stubbed return value is now stored.
- New low level spy API: double "calls" property provides access to invocations and their argument values. Each 'call' has an "args" sequence and "kargs dictionary". This provides support to perform individual assertions and direct access to invocation argument values. (see test and doc).

4.14.10 doublex 1.6

- First release supporting Python 3 (up to Python 3.2) [fixes issue 7].
- Ad-hoc stub attributes (see test).
- Partial support for non native Python functions.
- ProxySpy propagated stubbed invocations too (see test).

4.14.11 doublex 1.5.1

This release includes support for asynchronous spy assertions. See this blog post for the time being, soon in the official documentation.

4.14.12 doublex/pyDoubles 1.5

Since this release, doublex supports the pyDoubles API by means a wrapper. See pyDoubles for details.

In most cases the only required change in your code is the `import` sentence, that change from:

```
import pyDoubles.framework.*
```

to:

```
from doublex.pyDoubles import *
```

If you have problems migrating to the 1.5 release or migrating from pyDoubles to doublex, please ask for help in the discussion forum or in the issue tracker.

CHAPTER 5

Indices and tables

- genindex