pyDoubles Documentation

Release 1.7.2

David Villa Alises

December 26, 2013

Contents

Warning: Since version 1.5 the pyDoubles API is provided as a wrapper to doublex.
However, there are small differences. pyDoubles matchers are not supported anymore, although you may get the same feature using standard hamcrest matchers. Anyway, old pyDoubles matchers are provided as hamcrest aliases, so your old pyDoubles tests should work fine with minimal changes.
In most cases the only required change in your code is the module name, that change from:

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

to:

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

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

This is the pyDoubles documentation, that was available in the (now disappeared) pydoubles.org site.

CHAPTER 1

Overview

1.1 What is pyDoubles?

pyDoubles is a test doubles framework for the Python platform. Test doubles frameworks are also called mocking frameworks. pyDoubles can be used as a **testing** tool **or** as a **Test Driven Development** tool.

It generates **stubs, spies, and mock objects** using a fluent interface that will make your **unit tests more readable**. Moreover, it's been designed to make your tests **less fragile** when possible.

The development of pyDoubles has been completely test-driven from scratch. The project is under continuous evolution, but you can extend the framework with your own requirements. The code is simple and well documented with unit tests.

1.2 Supported test doubles

Find out what test doubles are according to Gerard Meszaros. pyDoubles offers mainly three kind of doubles:

1.2.1 Stub

Replaces the implementation of one or more methods in the object instance which plays the role of collaborator or dependency, returning the value that we explicitly write down in the test. A stub is actually a method but it is also common to use the noun stub for a class with stubbed methods. The stub does not have any kind or memory.

Stubs are used mainly for **state validation** or along with spies or mocks.

1.2.2 Spy

Replaces the implementation as a stub does, but it is also able to register and remember what methods are called during the test execution and how they are invoked.

They are used for **interaction/behavior verification**.

1.2.3 Mock

Contains the same features than the Stub and therefore the Spy, but it is very strict in the behavior specification it should expect from the System Under Tests. Before calling any method in the mock object, the framework should be told (in the test) which methods we expect to be called in order for them to succeed. Otherwise, the test will fail with an "UnexpectedBehavior" exception.

Mock objects are used when we have to be very **precise in the behavior specification**. They usually make the tests more fragile than a spy but still are necessary in many cases. It is common to use mock objects together with stubs in tests.

1.3 New to test doubles?

A unit test is comprised of three parts: Arrange/Act/Assert or Given/When/Then or whatever you want to call them. The scenario has to be created, exercised, and eventually we verify that the expected behavior happened. The test doubles framework is used to create the scenario (create the objects), and verify behavior after the execution but **it does not make sense to invoke test doubles' methods in the test code.** If you call the doubles' methods in the test code, you are testing the framework itself, which has been already tested (better than that, we crafted it using TDD). Make sure the calls to the doubles' methods happen in your production code.

1.4 Why another framework?

pyDoubles is inspired in mockito and jMock for Java, and also inspired in Rhino.Mocks for .Net. There are other frameworks for Python that work really well, but after some time using them, we were not really happy with the syntax and the readability of the tests. Fragile tests were also a problem. Some well-known frameworks available for Python are: mocker, mockito-python, mock, pymox.

pyDoubles is open source and free software, released under the Apache License Version 2.0

Take a look at the project's blog

CHAPTER 2

Documentation

```
class SimpleExample(unittest.TestCase):
   def test_ask_the_sender_to_send_the_report(self):
        sender = spy(Sender())
        service = SavingsService(sender)

        service.analyze_month()
        assert_that_method(sender.send_email).was_called(
                        ).with_args('reports@x.com', ANY_ARG)
```

2.1 Import the framework in your tests

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

If you are afraid of importing everything from the pyDoubles.framework module, you can use custom imports, although it has been carefully designed to not conflict with your own classes:

```
import unittest
from doublex.pyDoubles import stub, spy, mock
from doublex.pyDoubles import when, expect_call, assert_that_method
from doublex.pyDoubles import method_returning, method_raising
```

You can import Hamcrest matchers which are fully supported:

```
from hamcrest import *
```

2.2 Which doubles do you need?

You can choose to stub out a method in a regular object instance, to stub the whole object, or to create three types of spies and two types of mock objects.

2.2.1 Stubs

There are several ways to stub out methods.

Stub out a single method

If you just need to replace a single method in the collaborator object and you don't care about the input parameters, you can stub out just that single method:

```
collaborator = Collaborator() # create the actual object
collaborator.some_calculation = method_returning(10)
```

Now, when your production code invokes the method "some_calculation" in the collaborator object, the framework will return 10, no matter what parameters are passed in as the input.

If you want the method to raise an exception when called use this:

```
collaborator.some_calculation = method_raising(ApplicationException())
```

You can pass in any type of exception.

Stub out the whole object

Now the collaborator instance won't be the actual object but a replacement:

```
collaborator = stub(Collaborator())
```

Any method will return "None" when called with any input parameters. If you want to change the return value you can use the "when" sentence:

```
when(collaborator.some_calculation).then_return(10)
```

Now, when your production code invokes "some_calculation" method, the stub will return 10, no matter what arguments are passed in. You can also specify different return values depending on the input:

```
when(collaborator.some_calculation).with_args(5).then_return(10)
when(collaborator.some_calculation).with_args(10).then_return(20)
```

This means that "collaborator.some_calculation(5)" will return 10, and that it will return 20 when the input is 10. You can define as many input/output specifications as you want:

```
when(collaborator.some_calculation).with_args(5).then_return(10)
when(collaborator.some_calculation).then_return(20)
```

This time, "collaborator.some_calculation(5)" will return 10, and it will return 20 in any other case.

Any argument matches

The special keyword ANY_ARG is a wildcard for any argument in the stubbed method:

```
when(collaborator.some_other_method).with_args(5, ANY_ARG).then_return(10)
```

The method "some_other_method" will return 10 as long as the first parameter is 5, no matter what the second parameter is. You can use any combination of "ANY_ARG" arguments. But remember that if all of them are ANY, you shouldn't specify the arguments, just use this:

```
when(collaborator.some_other_method).then_return(10)
```

It is also possible to make the method return exactly the first parameter passed in:

```
when(collaborator.some_other_method).then_return_input()
```

So this call: `collaborator.some_other_method(10)` wil return `10`.

Matchers

You can also specify that arguments will match a certain function. Say that you want to return a value only if the input argument contains the substring "abc":

```
when(collaborator.some_method).with_args(
        str_containing("abc")).then_return(10)
```

Hamcrest Matchers

Since pyDoubles v1.2, we fully support Hamcrest matchers. They are used exactly like pyDoubles matchers:

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

def test_has_entry_matcher(self):
    list = {'one':1, 'two':2}
    when(self.spy.one_arg_method).with_args(
        has_entry(equal_to('two'), 2)).then_return(1000)
    assert_that(1000, equal_to(self.spy.one_arg_method(list)))

def test_all_of_matcher(self):
    text = 'hello'
    when(self.spy.one_arg_method).with_args(
        all_of(starts_with('h'), instance_of(str))).then_return(1000)
    assert_that(1000, equal_to(self.spy.one_arg_method(text)))
```

Note that the tests above are just showhing the pyDoubles framework working together with Hamcrest, they are not good examples of unit tests for your production code.

The method assert_that comes from Hamcrest, as well as the matchers: has_entry, equal_to, all_of, starts_with, instance_of. Notice that all_of and any_of, allow you to define more than one matcher for a single argument, which is really powerful. For more informacion on matchers, read this blog post

Stub out the whole unexisting object

If the Collaborator class does not exist yet, or you don't want the framework to check that the call to the stub object method matches the actual API in the actual object, you can use an "empty" stub:

```
collaborator = empty_stub()
when(collaborator.alpha_operation).then_return("whatever")
```

The framework is creating the method "alpha_operation" dynamically and making it return "whatever".

The use of empty_stub, empty_spy or empty_mock is not recommended because you lose the API match check. We only use them as the construction of the object is too complex among other circumstances.

2.2.2 Spies

Please read the documentation above about stubs, because the API to define method behaviors is the same for stubs and spies. To create the object:

```
collaborator = spy(Collaborator())
```

After the execution of the system under test, we want to validate that certain call was made:

```
assert_that_method(collaborator.send_email).was_called()
```

That will make the test pass if method "send_email" was invoked one or more times, no matter what arguments were passed in. We can also be precise about the arguments:

```
assert_that_method(collaborator.send_email).was_called().with_args("example@iexpertos.com")
```

Notice that you can combine the "when" statement with the called assertion:

```
def test_sut_asks_the_collaborator_to_send_the_email(self):
    sender = spy(Sender())
    when(sender.send_email).then_return(SUCCESS)
    object_under_test = Sut(sender)

    object_under_test.some_action()

    assert_that_method(sender.send_email).was_called().with_args("example@iexpertos.com")
```

Any other call to any method in the "sender" double will return "None" and will not interrupt the test. We are not telling all that happens between the sender and the SUT, we are just asserting on what we want to verify.

The ANY_ARG matcher can be used to verify the call as well:

```
assert_that_method(collaborator.some_other_method).was_called().with_args(5, ANY_ARG)
```

Matchers can also be used in the assertion:

```
assert_that_method(collaborator.some_other_method).was_called().with_args(5, str_containing("abc"))
```

It is also possible to assert that wasn't called using:

```
assert_that_method(collaborator.some_method).was_never_called()
```

You can assert on the number of times a call was made:

```
assert_that_method(collaborator.some_method).was_called().times(2)
assert_that_method(collaborator.some_method).was_called(
     ).with_args(SOME_VALUE, OTHER_VALUE).times(2)
```

You can also create an "empty_spy" to not base the object in a certain instance:

```
sender = empty_spy()
```

The ProxySpy

There is a special type of spy supported by the framework which is the ProxySpy:

```
collaborator = proxy_spy(Collaborator())
```

The proxy spy will record any call made to the object but rather than replacing the actual methods in the actual object, it will execute them. So the actual methods in the Collaborator will be invoked by default. You can replace the methods one by one using the "when" statement:

```
when(collaborator.some_calculation).then_return(1000)
```

Now "some_calculation" method will be a stub method but the remaining methods in the class will be the regular implementation.

The ProxySpy might be interesting when you don't know what the actual method will return in a given scenario, but still you want to check that some call is made. It can be used for debugging purposes.

2.2.3 Mocks

Before calls are made, they have to be expected:

```
def test_sut_asks_the_collaborator_to_send_the_email(self):
    sender = mock(Sender())
    expect_call(sender.send_email)
    object_under_test = Sut(sender)

    object_under_test.some_action()

    sender.assert_that_is_satisfied()
```

The test is quite similar to the one using a spy. However the framework behaves different. If any other call to the sender is made during "some_action", the test will fail. This makes the test more fragile. However, it makes sure that this interaction is the only one between the two objects, and this might be important for you.

More precise expectations

You can also expect the call to have certain input parameters:

```
expect_call(sender.send_email).with_args("example@iexpertos.com")
```

Setting the return of the expected call

Additionally, if you want to return anything when the expected call occurs, there are two ways:

```
expect_call(sender.send_email).returning(SUCCESS)
```

Which will return SUCCESS whatever arguments you pass in, or:

```
expect_call(sender.send_email).with_args("wrong_email").returning(FAILURE)
```

Which expects the method to be invoked with "wrong_email" and will return FAILURE.

Mocks are strict so if you expect the call to happen several times, be explicit with that:

```
expect_call(sender.send_email).times(2)
```

```
expect_call(sender.send_email).with_args("admin@iexpertos.com").times(2)
```

Make sure the "times" part is at the end of the sentence:

```
expect_call(sender.send_email).with_args("admin@iexpertos.com").returning('OK').times(2)
```

As you might have seen, the "when" statement is not used for mocks, only for stubs and spies. Mock objects use the "expect_call" syntax together with the "assert_that_is_satisfied" (instance method).

2.3 More documentation

The best and most updated documentation are the unit tests of the framework itself. We encourage the user to read the tests and see what features are supported in every commit into the source code repository:

- pyDoublesTests/unit.py

You can also read about what's new in every release in the blog.

CHAPTER 3

Support

3.1 Free support

- Mailing list
- Issue tracker

3.2 Commercial support

The development team of pyDoubles is a software company based in Spain. We are happy to help other companies with the usage and extension of pyDoubles. If you want to have custom features or direct support, please contact us at info@iexpertos.com