doubles Documentation

Release 1.5.1

Jimmy Cuadra

July 30, 2018

Contents

Doubles is a Python package that provides test doubles for use in automated tests.

It provides functionality for stubbing, mocking, and verification of test doubles against the real objects they double. In contrast to the Mock package, it provides a clear, expressive syntax and better safety guarantees to prevent API drift and to improve confidence in tests using doubles. It comes with drop-in support for test suites run by Pytest, Nose, or standard unittest.

CHAPTER 1

Installation

From PyPI:

```
$ pip install doubles
```

From source:

```
$ git clone https://github.com/uber/doubles
$ cd doubles
$ python setup.py install
```

CHAPTER 2

Integration with test frameworks

Doubles includes plugins for automatic integration with popular test runners.

Pytest

Pytest integration will automatically be loaded and activated via setuptools entry points. To disable Doubles for a particular test run, run Pytest as:

```
$ py.test -p no:doubles file_or_directory
```

Nose

Nose integration will be loaded and activated by running Nose as:

```
$ nosetests --with-doubles file_or_directory
```

unittest

Inherit from `doubles.unittest.TestCase` in your test case classes and the Doubles lifecycle will be managed automatically.

Manual integration

If you are using another test runner or need manual control of the Doubles lifecycle, these are the two methods you'll need to use:

1. `doubles.verify` should be called after each test to verify any expectations made. It can be skipped if the test case has already failed for another reason.
2. `doubles.teardown` must be called after each test and after the call to `doubles.verify`.

CHAPTER 3

Differences from Mock

If you've previously used the Mock package, you may be wondering how **Doubles** is different and why you might want to use it. There are a few main differences:

- **Mock** follows what it describes as the "action –> assertion" pattern, meaning that you make calls to test doubles and then make assertions afterwards about how they are used. **Doubles** takes the reverse approach: you declare explicitly how your test doubles should behave, and any expectations you've made will be verified automatically at the end of the test.
- **Mock** has one primary class, also called Mock, which can serve the purpose of different types of test doubles depending on how it's used. **Doubles** uses explicit terminology to help your tests better convey their intent. In particulary, there is a clear distinction between a stub and a mock, with separate syntax for each.
- **Doubles** ensures that all test doubles adhere to the interface of the real objects they double. This is akin to **Mock**'s "spec" feature, but is *not* optional. This prevents drift between test double usage and real implementation. Without this feature, it's very easy to have a passing test but broken behavior in production.
- **Doubles** has a fluid interface, using method chains to build up a specification about how a test double should be used which matches closely with how you might describe it in words.

CHAPTER 4

Terminology

Terminology used when discussing test doubles has often been confused, historically. To alleviate confusion, at least within the scope of using the Doubles library, the following definitions are provided:

test double An object that stands in for another object during the course of a test. This is a generic term that describes all the different types of objects the Doubles library provides.

stub A test double that returns a predetermined value when called.

fake A test double that has a full implementation that determines what value it will return when called.

mock A test double that expects to be called in a certain way, and will cause the test to fail if it is not.

pure double A basic test double that does not modify any existing object in the system.

partial double A test double that modifies a real object from the production code, doubling some of its methods but leaving others unmodified.

verifying double A test double that ensures any methods that are doubled on it match the contract of the real object they are standing in for.

allowance A declaration that one of an object's methods can be called. This is the manner by which stubs are created.

expectation A declaration that one of an object's methods must be called. This is the manner by which mocks are created.

Examples of each of these are provided in the *Usage* section.

CHAPTER 5

Usage

Doubles is used by creating stubs and mocks with the `allow` and `expect` functions. Each of these functions takes a "target" object as an argument. The target is the object whose methods will be allowed or expected to be called. For example, if you wanted to expect a call to something like `User.find_by_id`, then `User` would be the target. Using a real object from your system as the target creates a so-called "partial double."

There are also three constructors, `InstanceDouble`, `ClassDouble`, and `ObjectDouble`, which can be used to create so-called "pure double" targets, meaning they are unique objects which don't modify any existing object in the system.

The details of `allow`, `expect`, and the three pure double constructors follow.

Stubs and allowances

Stubs are doubles which have a predetermined result when called. To stub out a method on an object, use the `allow` function:

```
from doubles import allow

from myapp import User


def test_allows_get_name():
    user = User('Carl')

    allow(user).get_name

    assert user.get_name() is None
```

On the first line of the test, we create a user object from a theoretical class called `User` in the application we're testing. The second line declares an *allowance*, after which `user.get_name` will use the stub method rather than the real implementation when called. The default return value of a stub is `None`, which the third line asserts.

To instruct the stub to return a predetermined value, use the `and_return` method:

```
from doubles import allow

from myapp import User


def test_allows_get_name():
    user = User('Carl')
```

```
    allow(user).get_name.and_return('Henry')

    assert user.get_name() == 'Henry'
```

By default, once a method call has been allowed, it can be made any number of times and it will always return the value specified.

The examples shown so far will allow the stubbed method to be called with any arguments that match its signature. To specify that a method call is allowed only with specific arguments, use `with_args`:

```
from doubles import allow

from myapp import User


def test_allows_set_name_with_args():
    user = User('Carl')

    allow(user).set_name.with_args('Henry')

    user.set_name('Henry')  # Returns None
    user.set_name('Teddy')  # Raises an UnallowedMethodCallError
```

You do not need to specifically call `with_args`, calling the allowance directly is the same as calling `with_args`. The following example is identical to the code above:

```
from doubles import allow

from myapp import User


def test_allows_set_name_with_args():
    user = User('Carl')

    allow(user).set_name('Henry')

    user.set_name('Henry')  # Returns None
    user.set_name('Teddy')  # Raises an UnallowedMethodCallError
```

Multiple allowances can be specified for the same method with different arguments and return values:

```
from doubles import allow

from myapp import User

def test_returns_different_values_for_different_arguments():
    user = User('Carl')

    allow(user).speak.with_args('hello').and_return('Carl says hello')
    allow(user).speak.with_args('thanks').and_return('Carl says thanks')

    assert user.speak('hello') == 'Carl says hello'
    assert user.speak('thanks') == 'Carl says thanks'
```

To specify that a method can only be called *with no arguments*, use `with_no_args`:

```
from doubles import allow

from myapp import User
```

```
def test_allows_greet_with_no_args():
    user = User('Carl')

    allow(user).greet.with_no_args().and_return('Hello!')

    user.greet()  # Returns 'Hello!'
    user.greet('Henry')  # Raises an UnallowedMethodCallError
```

Without the call to `with_no_args`, `user.greet('Henry')` would have returned `'Hello!'`.

Mocks and expectations

Stubs are useful for returning predetermined values, but they do not verify that they were interacted with. To add assertions about double interaction into the mix, create a mock object by declaring an *expectation*. This follows a very similar syntax, but uses `expect` instead of `allow`:

```
from doubles import expect

from myapp import User


def test_allows_get_name():
    user = User('Carl')

    expect(user).get_name
```

The above test will fail with a `MockExpectationError` exception, because we expected `user.get_name` to be called, but it was not. To satisfy the mock and make the test pass:

```
from doubles import expect

from myapp import User


def test_allows_get_name():
    user = User('Carl')

    expect(user).get_name

    user.get_name()
```

Mocks support the same interface for specifying arguments that stubs do. Mocks do not, however, support specification of return values or exceptions. If you want a test double to return a value or raise an exception, use a stub. Mocks are intended for verifying calls to methods that do not return a meaningful value. If the method does return a value, write an assertion about that value instead of using a mock.

Doubling top-level functions

The previous sections have shown examples where methods on classes are stubbed or mocked. It's also possible to double a top-level function by importing the module where the function is defined into your test file. Pass the module to `allow` or `expect` and proceed as normal. In the follow example, imagine that we want to stub a function called `generate_user_token` in the `myapp.util` module:

```
from doubles import allow

from myapp import util, User

def test_get_token_returns_a_newly_generated_token_for_the_user():
    user = User('Carl')

    allow(util).generate_user_token.with_args(user).and_return('dummy user token')

    assert user.get_token() == 'dummy user token'
```

Fakes

Fakes are doubles that have special logic to determine their return values, rather than returning a simple static value. A double can be given a fake implementation with the `and_return_result_of` method, which accepts any callable object:

```
from doubles import allow

from myapp import User


def test_fake():
    user = User('Carl')

    allow(user).greet.and_return_result_of(lambda: 'Hello!')

    assert user.greet() == 'Hello!'
```

Although this example is functionally equivalent to calling `and_return('Hello!')`, the callable passed to `and_return_result_of` can be arbitrarily complex. Fake functionality is available for both stubs and mocks.

Raising exceptions

Both stubs and mocks allow a method call to raise an exception instead of returning a result using the `and_raise` method. Simply pass the object you want to raise as an argument. The following test will pass:

```
from doubles import allow

from myapp import User


def test_raising_an_exception():
    user = User('Carl')

    allow(user).get_name.and_raise(StandardError)

    try:
        user.get_name()
    except StandardError:
        pass
    else:
        raise AssertionError('Expected test to raise StandardError.')
```

If the exception to be raised requires arguments, they can be passed to the Exception constructor directly before `and_raises` is invoked:

```
from doubles import allow

from myapp import User


def test_raising_an_exception():
    user = User('Carl')

    allow(user).get_name.and_raise(NonStandardError('an argument', arg2='another arg'))

    try:
        user.get_name()
    except NonStandardError:
        pass
    else:
        raise AssertionError('Expected test to raise NonStandardError.')
```

Call counts

Limits can be set on how many times a doubled method can be called. In most cases, you'll specify an exact call count with the syntax `exactly(n).times`, which will cause the test to fail if the doubled method is called fewer or more times than you declared:

```
from doubles import expect

from myapp import User

def test_expect_one_call():
    user = User('Carl')

    expect(user).get_name.exactly(1).time

    user.get_name()
    user.get_name()  # Raises a MockExpectationError because it should only be called once
```

The convenience methods `once`, `twice` and `never` are provided for the most common use cases. The following test will pass:

```
from doubles import expect

from myapp import User

def test_call_counts():
    user = User('Carl')

    expect(user).get_name.once()
    expect(user).speak.twice()
    expect(user).not_called.never()

    user.get_name()
    user.speak('hello')
    user.speak('good bye')
```

To specify lower or upper bounds on call count instead of an exact number, use `at_least` and `at_most`:

```
from doubles import expect

from myapp import User

def test_bounded_call_counts():
    user = User('Carl')

    expect(user).get_name.at_least(1).time
    expect(user).speak.at_most(2).times

    user.get_name  # The test would fail if this wasn't called at least once
    user.speak('hello')
    user.speak('good bye')
    user.speak('oops')  # Raises a MockExpectationError because we expected at most two calls
```

Call counts can be specified for allowances in addition to expectations, with the caveat that only upper bounds are enforced for allowances, making `at_least` a no-op.

Partial doubles

In all of the examples so far, we added stubs and mocks to an instance of our production `User` class. These are called a partial doubles, because only the parts of the object that were explicitly declared as stubs or mocks are affected. The untouched methods on the object behave as usual. Let's take a look at an example that illustrates this.:

```
from doubles import allow


class User(object):
    @classmethod
    def find_by_email(cls, email):
        pass

    @classmethod
    def find_by_id(cls, user_id):
        pass

def test_partial_double():
    dummy_user = object()

    allow(User).find_by_email.and_return(dummy_user)

    User.find_by_email('alice@example.com')  # Returns <object object at 0x100290090>
    User.find_by_id(1)  # Returns <User object at 0x1006a8cd0>
```

For the sake of the example, assume that the two class methods on `User` are implemented to return an instance of the class. We create a sentinel value to use as a dummy user, and stub `User` to return that specific object when `User.find_by_email` is called. When we then call the two class methods, we see that the method we stubbed returns the sentinel value as we declared, and `User.find_by_id` retains its real implementation, returning a `User` object.

After a test has run, all partial doubles will be restored to their pristine, undoubled state.

Verifying doubles

One of the trade offs of using test doubles is that production code may change after tests are written, and the doubles may no longer match the interface of the real object they are doubling. This is known as "API drift" and is one possible cause of the situation where a test suite is passing but the production code is broken. The potential for API drift is often used as an argument against using test doubles. **Doubles** provides a feature called verifying doubles to help address API drift and to increase confidence in test suites.

All test doubles created by **Doubles** are verifying doubles. They will cause the test to fail by raising a `VerifyingDoubleError` if an allowance or expectation is declared for a method that does not exist on the real object. In addition, the test will fail if the method exists but is specified with arguments that don't match the real method's signature.

In all the previous examples, we added stubs and mocks for real methods on the `User` object. Let's see what happens if we try to stub a method that doesn't exist:

```
from doubles import allow

from myapp import User


def test_verification():
    user = User('Carl')

    allow(user).foo  # Raises a VerifyingDoubleError, because User objects have no foo method
```

Similarly, we cannot declare an allowance or expectation with arguments that don't match the actual signature of the doubled method:

```
from doubles import allow

from myapp import User


def test_verification_of_arguments():
    user = User('Carl')

    # Raises a VerifyingDoubleArgumentError, because set_name accepts only one argument
    allow(user).set_name.with_args('Henry', 'Teddy')
```

Disabling builtin verification

Some of the objects in Python's standard library are written in C and do not support the same introspection capabilities that user-created objects do. Because of this, the automatic verification features of **Doubles** may not work when you try to double a standard library function. There are two approaches to work around this:

Recommended: Create a simple object that wraps the standard library you want to use. Use your wrapper object from your production code and double the wrapper in your tests. Test the wrapper itself in integration with the real standard library calls, without using test doubles, to ensure that your wrapper works as expected. Although this may seem heavy handed, it's actually a good approach, since it's a common adage of test doubles never to double objects you don't own.

Alternatively, use the `no_builtin_verification` context manager to disable the automatic verification. This is not a recommended approach, but is available if you must use it:

```
from doubles import allow, InstanceDouble, no_builtin_verification
```

```
with no_builtin_verification():
    date = InstanceDouble('datetime.date')

    allow(date).ctime

    assert date.ctime() is None
```

Pure doubles

Often it's useful to have a test double that represents a real object, but does not actually touch the real object. These doubles are called pure doubles, and like partial doubles, stubs and mocks are verified against the real object. In contrast to partial doubles, pure doubles do not implement any methods themselves, so allowances and expectations must be explicitly declared for any method that will be called on them. Calling a method that has not been allowed or expected on a pure double will raise an exception, even if the object the pure double represents has such a method.

There are three different constructors for creating pure doubles, depending on what type of object you're doubling and how it should be verified:

InstanceDouble

`InstanceDouble` creates a pure test double that will ensure its usage matches the API of an instance of the provided class. It's used as follows:

```
from doubles import InstanceDouble, allow


def test_verifying_instance_double():
  user = InstanceDouble('myapp.User')

  allow(user).foo
```

The argument to `InstanceDouble` is the fully qualified module path to the class in question. The double that's created will verify itself against an instance of that class. The example above will fail with a `VerifyingDoubleError` exception, assuming `foo` is not a real instance method.

ClassDouble

`ClassDouble` is the same as `InstanceDouble`, except that it verifies against the class itself instead of an instance of the class. The following test will fail, assuming `find_by_foo` is not a real class method:

```
from doubles import ClassDouble, allow

def test_verifying_class_double():
  User = ClassDouble('myapp.User')

  allow(User).find_by_foo
```

ObjectDouble

`ObjectDouble` creates a pure test double that is verified against a specific object. The following test will fail, assuming `foo` is not a real method on `some_object`:

```
from doubles import ObjectDouble, allow

from myapp import some_object


def test_verifying_object_double():
  something = ObjectDouble(some_object)

  allow(something).foo
```

There is a subtle distinction between a pure test double created with `ObjectDouble` and a partial double created by passing a non-double object to `allow` or `expect`. The former creates an object that does not accept any method calls which are not explicitly allowed, but verifies any that are against the real object. A partial double modifies parts of the real object itself, allowing some methods to be doubled and others to retain their real implementation.

Clearing Allowances

If you ever want to to clear all allowances and expectations you have set without verifying them, use `teardown`:

```
from doubles import teardown, expect

def test_clearing_allowances():
    expect(some_object).foobar

    teardown()
```

If you ever want to to clear all allowances and expectations you have set on an individual object without verifying them, use `clear`:

```
from doubles import clear, expect

def test_clearing_allowances():
    expect(some_object).foobar

    clear(some_object)
```

Patching

`patch` is used to replace an existing object:

```
from doubles import patch
import doubles.testing

def test_patch():
    patch('doubles.testing.User', 'Bob Barker')

    assert doubles.testing.User == 'Bob Barker'
```

Patches do not verify against the underlying object, so use them carefully. Patches are automatically restored at the end of the test.

Patching Classes

`patch_class` is a wrapper on top of `patch` to help you patch a python class with a `ClassDouble`. `patch_class` creates a `ClassDouble` of the class specified, patches the original class and returns the `ClassDouble`:

```
from doubles import patch_class, ClassDouble
import doubles.testing

def test_patch_class():
    class_double = patch_class('doubles.testing.User')

    assert doubles.testing.User is class_double
    assert isinstance(class_double, ClassDouble)
```

Stubbing Constructors

By default `ClassDoubles` cannot create new instances:

```
from doubles import ClassDouble

def test_unstubbed_constructor():
    User = ClassDouble('doubles.testing.User')
    User('Teddy', 1901)  # Raises an UnallowedMethodCallError
```

Stubbing the constructor of a `ClassDouble` is very similar to using `allow` or `expect` except we use: `allow_constructor` or `expect_constructor`, and don't specify a method:

```
from doubles import allow_constructor, ClassDouble
import doubles.testing

def test_allow_constructor_with_args():
    User = ClassDouble('doubles.testing.User')

    allow_constructor(User).with_args('Bob', 100).and_return('Bob')

    assert User('Bob', 100) == 'Bob'
```

The return value of `allow_constructor` and `expect_constructor` support all of the same methods as allow/expect. (e.g. `with_args`, `once`, `exactly`, .etc).

NOTE: Currently you can only stub the constructor of `ClassDoubles`

Stubbing Asynchronous Methods

Stubbing asynchronous methods requires returning futures `and_return_future` and `and_raise_future` do it for you.

Returning Values

Stubbing a method with `and_return_future` is similar to using `and_return`, except the value is wrapped in a `Future`:

```
from doubles import allow, InstanceDouble

def test_and_return_future():
    user = InstanceDouble('doubles.testing.User')
    allow(user).instance_method.and_return_future('Bob Barker')

    result = user.instance_method()
    assert result.result() == 'Bob Barker'
```

Raising Exceptions

Stubbing a method with `and_raise_future` is similar to using `and_raise`, except the exceptions is wrapped in a `Future`:

```
from doubles import allow, InstanceDouble
from pytest import raises

def test_and_raise_future():
    user = InstanceDouble('doubles.testing.User')
    exception = Exception('Bob Barker')
    allow(user).instance_method.and_raise_future(exception)
    result = user.instance_method()

    with raises(Exception) as e:
        result.result()

    assert e.value == exception
```

CHAPTER 6

API

Stubs and mocks

doubles.**allow**(*target*)

Prepares a target object for a method call allowance (stub). The name of the method to allow should be called as a method on the return value of this function:

```
allow(foo).bar
```

Accessing the `bar` attribute will return an `Allowance` which provides additional methods to configure the stub.

Parameters **target** (*object*) – The object that will be stubbed.

Returns An `AllowanceTarget` for the target object.

doubles.**expect**(*target*)

Prepares a target object for a method call expectation (mock). The name of the method to expect should be called as a method on the return value of this function:

```
expect(foo).bar
```

Accessing the `bar` attribute will return an `Expectation` which provides additional methods to configure the mock.

Parameters **target** (*object*) – The object that will be mocked.

Returns An `ExpectationTarget` for the target object.

doubles.**allow_constructor**(*target*)

Set an allowance on a `ClassDouble` constructor

This allows the caller to control what a ClassDouble returns when a new instance is created.

Parameters **target** (*ClassDouble*) – The ClassDouble to set the allowance on.

Returns an `Allowance` for the __new__ method.

Raise `ConstructorDoubleError` if target is not a ClassDouble.

doubles.**expect_constructor**(*target*)

Set an expectation on a `ClassDouble` constructor

Parameters **target** (*ClassDouble*) – The ClassDouble to set the expectation on.

Returns an `Expectation` for the __new__ method.

Raise `ConstructorDoubleError` if target is not a ClassDouble.

doubles.**patch**(*target*, *value*)
Replace the specified object

Parameters

- **target** (*str*) – A string pointing to the target to patch.
- **value** (*object*) – The value to replace the target with.

Returns A `Patch` object.

doubles.**patch_class**(*target*)
Replace the specified class with a ClassDouble

Parameters

- **target** (*str*) – A string pointing to the target to patch.
- **values** (*obj*) – Values to return when new instances are created.

Returns A `ClassDouble` object.

class doubles.allowance.**Allowance**(*target*, *method_name*, *caller*)
An individual method allowance (stub).

and_raise(*exception*, **args*, ***kwargs*)
Causes the double to raise the provided exception when called.

If provided, additional arguments (positional and keyword) passed to *and_raise* are used in the exception instantiation.

Parameters **exception** (*Exception*) – The exception to raise.

and_return(**return_values*)
Set a return value for an allowance

Causes the double to return the provided values in order. If multiple values are provided, they are returned one at a time in sequence as the double is called. If the double is called more times than there are return values, it should continue to return the last value in the list.

Parameters **return_values** (*object*) – The values the double will return when called,

and_return_result_of(*return_value*)
Causes the double to return the result of calling the provided value.

Parameters **return_value** (*any callable object*) – A callable that will be invoked to determine the double's return value.

with_args(**args*, ***kwargs*)
Declares that the double can only be called with the provided arguments.

Parameters

- **args** – Any positional arguments required for invocation.
- **kwargs** – Any keyword arguments required for invocation.

with_no_args()
Declares that the double can only be called with no arguments.

class doubles.expectation.**Expectation**(*target*, *method_name*, *caller*)
An individual method expectation (mock).

with_args(**args*, ***kwargs*)
Declares that the double can only be called with the provided arguments.

Parameters

- **args** – Any positional arguments required for invocation.
- **kwargs** – Any keyword arguments required for invocation.

with_no_args()
Declares that the double can only be called with no arguments.

Pure doubles

class doubles.**InstanceDouble**(*path*, ***kwargs*)
A pure double representing an instance of the target class.

Any kwargs supplied will be set as attributes on the instance that is created.

```
user = InstanceDouble('myapp.User', name='Bob Barker')
```

Parameters **path** (*str*) – The absolute module path to the class.

class doubles.**ClassDouble**(*path*)
A pure double representing the target class.

```
User = ClassDouble('myapp.User')
```

Parameters **path** (*str*) – The absolute module path to the class.

class doubles.**ObjectDouble**(*target*)
A pure double representing the target object.

```
dummy_user = ObjectDouble(user)
```

Parameters **target** (*object*) – The object the newly created ObjectDouble will verify against.

Test lifecycle

doubles.**verify**()
Verify a mock

Verifies any mocks that have been created during the test run. Must be called after each test case, but before teardown.

doubles.**teardown**()
Tears down the current Doubles environment. Must be called after each test case.

Exceptions

exception doubles.exceptions.**ConstructorDoubleError**
An exception raised when attempting to double the constructor of a non ClassDouble.

exception doubles.exceptions.**MockExpectationError**
An exception raised when a mock fails verification.

exception doubles.exceptions.**UnallowedMethodCallError**
An exception raised when an unallowed method call is made on a double.

exception doubles.exceptions.**VerifyingBuiltinDoubleArgumentError**
An exception raised when attempting to validate arguments of a builtin.

exception doubles.exceptions.**VerifyingDoubleArgumentError**
An exception raised when attempting to double a method with arguments that do not match the signature of the real method.

exception doubles.exceptions.**VerifyingDoubleError**(*method_name*, *doubled_obj*)
An exception raised when attempting to double a method that does not exist on the real object.

exception doubles.exceptions.**VerifyingDoubleImportError**
An exception raised when attempting to create a verifying double from an invalid module path.

CHAPTER 7

FAQ

Common Issues

When I double __new__, it breaks other tests, why?

This feature is deprecated, I recommend using the `patch_class` method, which fixes this issue and is much cleaner.

I get a `VerifyingDoubleError` "Cannot double method ... does not implement it", whats going on?

Make sure you are using a version of doubles greater than 1.0.1. There was a bug prior to 1.0.1 that would not allow you to mock callable objects.

I get a `VerifyingBuiltinDoubleArgumentError` "... is not a Python func", what is going on?

Python does not allow doubles to look into builtin functions and asked them what their call signatures are. Since we can't do this it is impossible to verify the arguments passed into a stubbed method. By default if doubles cannot inspect a function it raises a `VerifyingBuiltinDoubleArgumentError`, this is most common with builtins. To bypass this functionality for builtins you can do:

```
import functools

from doubles import no_builtin_verification, allow


with no_builtin_verification():
    allow(functools).reduce

    # The test that uses this allowance must be within the context manager.
    run_test()
```

Patches

How can I make SomeClass(args, kwargs) return my double?

Use `patch_class` and `allow_constructor`:

```
from doubles import patch_class, allow_constructor

import myapp

def test_something_that_uses_user():
    patched_class = patch_class('myapp.user.User')
    allow_constructor(patched_class).and_return('Bob Barker')

    assert myapp.user.User() == 'Bob Barker'
```

`patch_class` creates a `ClassDouble` of the class specified, patches the original class and returns the `ClassDouble`. We then stub the constructor which controls what is is returned when an instance is created.

`allow_constructor` supports all of the same functionality as `allow`:

```
from doubles import patch_class, allow_constructor

import myapp

def test_something_that_uses_user():
    bob_barker = InstanceDouble('myapp.user.User')

    patched_class = patch_class('myapp.user.User')
    allow_constructor(patched_class).with_args('Bob Barker', 100).and_return(bob_barker)

    assert myapp.user.User('Bob Barker', 100) is bob_barker
```

How can I patch something like I do with mock?

Doubles also has `patch` but it isn't a decorator:

```
from doubles import allow, patch

import myapp

def test_something_that_uses_user():
    patch('myapp.user.User', 'Bob Barker')

    assert myapp.user.User == 'Bob Barker'
```

Patches do not verify against the underlying object, so use them carefully. Patches are automatically restored at the end of the test.

Expectations

How do I assert that a function is not called?

If you expect the `send_mail` method never to be called on `user`:

```
expect(user).send_mail.never()
```

CHAPTER 8

Changelog

1.5.1 (2018-7-24)

- Fix bug which breaks automatic teardown of top-level expectations between test cases

1.5.0 (2018-6-07)

- Report unsatisfied expectations as failures instead of errors.

1.4.0 (2018-4-25)

- Fix bug in unsatisfied *with_args_validator* exceptions. Note this may cause some tests being run with the *unittest*

runner that used to pass to fail.

1.3.2 (2018-4-17)

- Fix bug in *and_raise*

1.3.1 (2018-4-16)

- Support Pytest 3.5
- Support Exceptions with custom args
- Cleanup test runner integration docs
- Update is_class check, use builtin method
- Cleanup some grammar in failure messages

1.2.1 (2016-3-20)

- Make expectation failure messages clearer

1.2.0 (2016-3-2)

- update pytest integration for version >=2.8
- Support arbitrary callables on class

1.1.3 (2015-10-3)

- Fix bug when restoring stubbed attributes.

1.1.2 (2015-10-3)

- Support stubbing callable attributes.

1.1.1 (2015-9-23)

- Optimized suite by using a faster method of retrieving stack frames.

1.1.0 (2015-8-23)

- Native support for futures: *and_return_future* and *and_raise_future*

1.0.8 (2015-3-31)

- Allow with_args_validator to work with expectations

1.0.7 (2015-3-17)

- Added __name__ and __doc__ proxying to ProxyMethod objects.
- Expectations can return values and raise exceptions.
- Add with_args_validator, user_defined arg validators.
- Validate arguments of a subset builtin objects (dict, tuple, list, set).
- Update FAQ.

1.0.6 (2015-02-16)

- Add with_args short hand syntax
- Improve argument verification for mock.ANY and equals
- Fix pep issues that were added to flake8

1.0.5 (2015-01-29)

- Started tracking changes
- Add expect_constructor and allow_constructor
- Add patch and patch_class
- Add clear
- Clarify some error messages

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*

d

A

C

D

E

I

M

O

P

T

U

V

W