Stub Documentation

Release 0.1

Iain Lowe

October 21, 2014

Contents

A method stub or simply stub in software development is a piece of code used to stand in for some other programming functionality. A stub may simulate the behavior of existing code (such as a procedure on a remote machine) or be a temporary substitute for yet-to-be-developed code. Stubs are therefore most useful in porting, distributed computing as well as general software development and testing.

—Wikipedia [1]

The `stub` library allows you to temporarily replace the behaviour of callables and the value of attributes on objects.

[1] http://en.wikipedia.org/wiki/Method_stub

CHAPTER 1

Table of contents

1.1 Getting Started

The basic workhorse function of the `stub` library is `stub.stub()`

1.1.1 Stubbing Callables

The basic syntax for stubbing function behaviour is:

```
>>> from stub import stub
>>> target = lambda: 5 # This can be any callable object
>>> repl = lambda: 6 # This can also be any callable object
>>> f = stub(target, repl)
>>> target() # The target object now has the behaviour of the replacement object
6
```

Where `target` and `repl` are any *callable* objects. This function actually modifies the code of `target` in place, so all imports in other libraries automagically get the newest version of the stubbed-out behaviour.

Stubbing a bound function

To stub out a bound function, call `stub.stub()` with the target and a replacement function:

```
>>> from stub import stub
>>> class A:
...     def f(self):
...         return 5
...
>>> repl = lambda x: 6
>>> a = A()
>>> a.f()
5
>>> f = stub(a.f, repl)
>>> a.f()
6
```

The `stub.stub()` function delegates to `stub.stub_bound_function()` when the `target` object is a bound function.

Stubbing unbound functions

To stub an unbound function, call `stub.stub(f, repl)()` where *f* is the function to stub and *repl* is the function with which to replace it.

```
>>> def f(x):
...   return x + 1
...
>>> import stub
>>> def f_replacement(x):
...   return x + 2
...
>>> f(5)
6
>>> x = stub.stub(f, f_replacement)
>>> f(5)
7
```

The `stub.stub()` function delegates to `stub.stub_unbound_function()` when the `target` object is an unbound function.

1.1.2 Stubbing Attribute Values

To stub out attribute values call `stub.stub()` with new attribute values as keyword-arguments:

```
>>> from stub import stub
>>> class A(object):
...     def __init__(self):
...         self.x = 5
...
>>> a = A()
>>> a.x
5
>>> x = stub(a, x=10)
>>> a.x
10
```

The `stub.stub()` function delegates to `stub.stub_attributes()` when called with keyword-arguments.

1.1.3 Unstubbing

Once an attribute value or callable behaviour has been stubbed out, it can be useful to access the previous value or behaviour.

The unstubbed behaviour of a stubbed function `f` can be accessed calling the `f.unstubbed()` function:

```
>>> from stub import stub
>>> f = lambda: 5
>>> repl = lambda: 7
>>> x = stub(f, repl)
>>> f()
7
>>> f.unstubbed()
5
```

The unstubbed value of a stubbed attribute `x` on an object `obj` can be accessed by calling `obj.x.unstubbed()`:

```
>>> from stub import stub
>>> class A(object):
...     def __init__(self):
...         self.x = 5
...
>>> a = A()
>>> a.x
5
>>> x = stub(a, x=7)
>>> a.x
7
>>> a.x.unstubbed()
5
```

To unstub a stubbed-out callable, `target`, call `target.unstub()`:

```
>>> from stub import stub
>>> target = lambda: 5
>>> repl = lambda: 7
>>> x = stub(target, repl)
>>> target()
7
>>> target.unstub()
>>> target()
5
```

To unstub an individual attribute `x` on an object, `target`, call `obj.x.unstub()`:

```
>>> from stub import stub
>>> class A(object):
...     def __init__(self):
...         self.x = 5
...
>>> a = A()
>>> x = stub(a, x=7)
>>> a.x
7
>>> a.x.unstub()
>>> a.x
5
```

To unstub all stubbed-out attributes on an object, `target`, call `obj.unstub()`:

```
>>> from stub import stub
>>> class A(object):
...     def __init__(self):
...         self.x = 5
...         self.y = 7
...
>>> a = A()
>>> stub(a, x=7, y=10)
>>> a.x, a.y
(7, 10)
>>> a.unstub()
>>> a.x, a.y
(5, 7)
```

To unstub **all** stubbed-out functions and attributes, call `stub.unstub_all()`

```
>>> from stub import stub, unstub_all
>>> # Setup some base objects to stub
>>> class A(object):
...     def __init__(self):
...         self.x = 5
...         self.y = 7
...
>>> a = A()
>>> target = lambda: 5
>>> repl = lambda: 7
>>> # Stub out callable behaviour and object attributes
>>> stub(a, x=7, y=10)
>>> x = stub(target, repl)
>>> a.x, a.y, target() # Check that stubbing worked
(7, 10, 7)
>>> unstub_all() # Unstub everything
>>> a.x, a.y, target() # Check that unstubbing worked
(5, 7, 5)
```

Warning: The `stub.unstub_all()` function will affect **all** stubbed-out functions, methods and attributes **everywhere** in your current runtime. It is important also to **never** fiddle with `stub._stubbed_callables` and `stub._modified_objects` since these collections track stubbed-out objects.

1.2 API Documentation

1.2.1 `stub`

1.2.2 `stubbing`

1.3 Stubbing vs. Mocking

We've done a lot of testing of Python systems over the past couple of years. The best approach we've found so far for replacing the behaviour of test objects at runtime is stubbing.

We've read a lot about mocking and even written mocking libraries for other languages. But there is something about the way that mocks need to be manipulated that feels distinctly un-Pythonic.

Unlike some other languages, Python allows you to access most of the internals of objects easily at runtime. Using this facility, we have developed the `stub` library to help with replacing functionality on functions and objects.

1.3.1 Separation of concerns

When you can modify objects and functions for testing purposes and confine those modifications to testing code, your production code becomes much cleaner and easier to read. You can do away with all the large `if debug:` blocks and keep everything clear and concise.

This problem is usually solved with complex type hierarchies providing levels of abstraction that allow developers to plug testing implementations into the pre-defined interface. What's funny/sad about this is that it is pure duplication of effort. We already have an interface, it's just that we need to hook into it. Most of the function definitions and current type hierarchies do not need to be further abstracted in order to support testing.

A prime example is database connector modules. Obviously, you expect this module to actually connect to a database. Equally obvious is the fact that you don't want this module connecting to a database for every single one of it's thousands of unit-tests. So most projects have an abstraction layer with an `AbstractAccessor` superclass that is then subclassed into `MySQLAccessor` and `TestAccessor` classes. This is not a terrible solution, but it *does* increase the cognitive overhead of dealing with the hot, or most common, path of execution.

When we write code that is clear in intention, it is easy to maintain and update. Each successive layer of abstraction we add makes this more difficult. So keeping the test code *completely* separate from the production code just makes sense.

This philosophy leads us to prefer code like this:

```
class MySQLAccessor:
        def __init__(self):
                pass

        def connect(self, user=None, pass=None, dbhost=None):
                do_something()

def myfunc(x):
        return do_something_real()

# module mylib_test
import stub

def myfunc_repl(x):
        return 5 + x

stub.stub(myfunc, myfunc_repl)
```

But while stubbing may seem easy when you look at replacing behaviour on instance objects, stubbing functional code turns out to be a whole 'nother ball of wax...

1.3.2 But I *did* import it!

One of the most frustrating things when testing is needing to replace the behaviour of a function from another module.

When the function is used in multiple places in the codebase and each place uses it's own `import` statement, each namespace has a binding to the original function so simple monkeypatching of the source module won't help:

```python
# Module module_a
import mylib

do_something(mylib.myfunc())

# Module module_b
import mylib

do_something_else(mylib.myfunc())

# Module mylib

def myfunc():
        return do_something_we_dont_want()
```

In this example, the naive approach to replacing the implementation of `mylib.myfunc` is to monkeypatch the *mylib* module like so:

```python
# Module mod_mylib
import mylib

def myfunc_repl():
        return do_something_desirable()

mylib.myfunc = myfunc_repl
```

Unfortunately, this code must execute *before* the code in *module_a* and *module_b* or else they will get handles on the original *myfunc* before it's definition is replaced. In order to update the code executed by all handles to the original *myfunc* it is necessary to modify the internal code object that the function object wraps. In this way, any modules that have already imported the function get the updated functionality.

The *stub* library handles this properly for both unbound functions and object methods. Allowing you to replace the above monkeypatching code with the following:

```python
# Module mod_mylib
import stub
import mylib

def myfunc_repl():
        return do_something_desirable()

stub(mylib.myfunc, myfunc_repl)
```

This code can run at any time regardless of imports into other modules and will correctly update the code of *myfunc* so that all handles execute the new instructions.

1.4 Unit tests

`stub` is (of course) fully unit-tested. Run `make test` to run the full suite or `make help` for other test options.