mimic Documentation

Release 0.0.1

Gavin McQuillan

February 11, 2013

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION TO MIMIC

1.1 What is Mimic?

Mimic is a mock library for python that is based on Google's Pymox, a fanstastic testing library, which is in turn based on EasyMock – a Java mock object framework.

Mimic allows you to write true unit tests even in situations in which your code is dependent on external systems, in situations in which dependency injection won't work, or would otherwise be too complicated.

1.1.1 Mimic Test Philosophy

Mimic is a bit more complex than many other mocking libraries. This is a strength and a weakness. The way Mimic tests are meant to be run is in the following order:

- specify expectations
- enter replay mode

So the first part of your test ends up being about setting up the scenario for mimic, and then the second part – after you enter replay mode – is about calling the code you hope to test from you test function.

This two-step process is a little extra work from the onset, but it's a hidden strength when you realize that Mimic holds you to the expectations you set: if you don't call a method you mock out, you get an error; if you call a method you weren't expecting, you get an error. It has a kind of symmetry that many developers find easy to trust because of its explicitness.

1.1.2 Why fork?

There are a couple of features that have been needed for a while, including:

- Move the codebase over to github (and thereby git) in order to allow for more community participation
- PEP8 compliant method names
- Experimental Python 3 support
- Complete, comprehensive documentation
- Continuous Integration
- **Fixes which have been rejected from pymox proper**
  - Nosetests fixes for 'one-character-per-line' exception output

Most importantly, though, a library this good needs active maintenance. It's been a few years now since the latest release. While this is a relatively mature code-base, there are a number of outstanding issues, which don't seem to be getting any traction.

1.2 Getting Started

1.2.1 Installing

You can download mimic from PyPI using pip or easy_install:

```
pip install mimic
```

1.2.2 Source Code and Issue Tracker

The sourcecode is available on github at https://github.com/gmcquillan/mimic/.

CHAPTER

TWO

MIMIC TUTORIAL

There are a few core concepts to understand about how Mimic works. Essentially, there's the part of the test where you setup your expectations, and then there's the part where you put your mocks into replay mode and call your code like normal.

Warning: Be careful when stubbing out your dependencies, mimic enfoces the contract you setup with it. If you say something gets called and it doesn't, mimic will raise an exception. You must provide a precise, deterministic view into what these Mock objects would do in regular service.

Note: The pymox project also has decent documentation.

2.1 Basics

Here's a rundown of the stages of a mimic-based test:

- Mimic instance
- Mocking out objects
- Replaying the mock objects
- Verifying and Unsetting the stubs (or ending Replay mode)

2.1.1 Mimic Instance

One way or another, you need a mimic instance from which to issue your commands for which class, methods, or other structures need to be made into mock objects.

In many examples, you might see a situation like this:

```
from mimic import Mimic
mime = Mimic()
```

Often this will happen in a test classes `setUp` method. However, you can save yourself the trouble by having your test class inherit from `mimic.MimicTestBase`:

When you do this, you get a `self.mimic` instance for free. However, that's not the only reason to do so. The other advantage is that the "Unsetting stubs" step will be done automatically at the end of each test method (*more on this later*).

```
class MyTests(mimc.MimicTestBase):

    def test_something(self):
        self.mimic.stub_out_with_mock(...)
```

2.1.2 Mocking Out Objects

Mocking Out A Function Call

A vast majority of mocking can just be done by calling `stub_out_with_mock`, this is good for situations in which you just need to override a particular function call so it doesn't interact with an external system (database), and/or you need to control the return values that the function returns.

```
# Now assuming that your test classes inherit from MimicTestBase
self.mimic.stub_out_with_mock(my_module, 'my_func')
my_module.my_func(mimic.ignore_arg()).and_return('Completed')
```

Mocking Out An Object

In situations where you need to access attributes and call functions on an object

```
my_module = self.mimic.create_mock_anything()
my_module.my_func(mimic.ignore_arg()).and_return('Completed')
```

Mocking Out A Class

In other situations you need to mock out the creation of an instance within the code that's being tested. In those cases use `stub_out_class_with_mocks`.

See *Stubbing Out A Class*

2.1.3 Replaying Mock Objects

After setting expectations, we trigger `replay mode` which means that we can make our calls for testing now.

```
# Set expectations
self.mimic.replay_all()

# Call your code
# Make your assertions
self.assertTrue(my_func())
```

2.1.4 Unsetting Stubs/Verification

After all the mocks have played out (successfully hopefully!) we need to let Mimic know that it's time to count all the calls and arguments that we setup in our expectations.

```
self.mimic.verify_all()
```

Note: This isn't necessary if you're inheriting from `mimic.MimicTestBase`! self.mimic.verify_all() will be called for you in that case!

CHAPTER

THREE

ADVANCED

3.1 Variable Parameters to Mock Functions

3.1.1 IgnoreArg()

3.1.2 And()

3.1.3 Or()

3.1.4 Is()

3.1.5 IsA()

3.1.6 IsAlmost()

3.1.7 StringContains()

3.1.8 Regex()

3.1.9 In()

3.1.10 Not()

3.1.11 ContainsKeyValue()

3.1.12 ContainsAttributeValue()

3.1.13 SameElementAs()

3.2 Calling Mock Objects Multiple Times

Mimic is pretty strict about which functions are called, and how many times they're called. This could be tedious if the same function is called with the same Parameters and you were required to setup those expectations repeatedly.

In a situation in which the function `my_call` is called three times during the course of a test, you could do this:

```
my_object.my_call().and_return(True)
my_object.my_call().and_return(True)
my_object.my_call().and_return(True)
```

However, you can chain a `multiple_times()` call into the first call.

```
my_object.my_call().multiple_times().and_return(True)
# Done! :)
```

Now, if you wanted to make sure it got called three times and no more or no less:

```
my_object.my_call().multiple_times(3).and_return(True)
```

3.3 Stubbing Out A Class

If you need to make an instance with the characteristic of a particular class, and whose instance attributes and methods you wish to be able to control, try `stub_out_class_with_mocks`. This is particularly useful when an object is instanstiated within the code tested by your test.

```
self.mimic.stub_out_class_with_mocks(my_module, 'MyClass')
mock_object = my_module.MyClass()
# Setup any expectations you want for MyClass
```

Now you have an object that you can set any expectations you need to override for your testing.

3.4 MockAnything Objects

These are sort of like a mock object created by `stub_out_class_with_mocks` however, you don't even need to specify a class. It's an empty vessel with which any expectations you want. This is useful whenever you might need an object, but the code you're testing isn't responsible for its creation (e.g. you can pass it into the function).

```
fake_result = self.mimic.create_mock_anything()
fake_result.result = 'Some fake result data'
fake_connection = self.mimic.create_mock_anything()
fake_connection.query(mimic.ignore_arg()).and_return(fake_result)
```

3.5 Stubbing Out Python Builtins

For example, if you need to mock out a python builtin such as `open`, the following code would work:

```
# Assuming you've setup your mimic instance as self.mimic
fake_conf_file = StringIO.StringIO('')
self.mimic.stub_out_with_mock(sys.modules['__builtin__'], 'open')
sys.modules['__builtin__'].open('path/to/file.txt', 'r').and_return(
    fake_conf_file
)

self.mimic.replay_all()

# Calls you would need to make that interact with filesystems, etc.
```

CHAPTER

FOUR

TODO

Things that need doing for the project to flourish.

4.1 High-level Projects

- Full Python3 support, preferably with backwards compatibility to 2.7. There is ongoing work on the `python3` branch in this repo. If you currently work in python3 and would like to use pymox or mimic, please take a look.
  - Current status for the `python3` branch is *72* failing tests out of 230. But basic mocking and replay of mock objects does seem to work.
  - The `python3` branch relies on the six module.

4.2 Low-hanging Fruit

- Convert all of Mimic and Mimic tests to be PEP8 compatible.
- Use RST docstrings to give use autodoc capabilities with Sphinx

CHAPTER

FIVE

INDICES AND TABLES

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