Using the testthatMockr package

Mark Hochguertel

April 9, 2016

1 Introduction

The *testthatMockr* package (referred to as *Mockr* from now on) is designed to be used with Hadley Wickham's *testthat* package for unit testing. Actually, *Mockr* isn't going to be much use to you unless you intend to be conducting unit testing with *testthat*.

This Vignette is a small tutorial to demonstrate the functionality of *Mockr*. Subsequent sections step through the different aspects currently implemented.

What *Mockr* intends to provide is the ability to mock out parts of your program that you don't need the details for as part of your test. This allows you to focus your efforts on the object actually being tested, as opposed to spending far too much time creating and setting up supporting objects just so you can check the behaviour of the objects under test.

This is where *Mockr* comes in, by allowing you to quickly mock up some objects and methods and get on with the real business of testing.

1.1 Inspiration

Aside from the *testthat* package, the inspiration for the desired functionality of *Mockr* has been drawn heavily from the *Python* package *Mock*. In fact, *Mockr* has been an attempt to port that package over to R. It's still a way off being able to replicate all the functionality in *Python Mock*, but certainly for my own unit testing *Mockr* has proved very handy.

1.2 Further reading

This Vignette is intended to show you the usage of the *Mockr* package. I'll leave the discussion about mock objects (and related topics) to people who are far more qualified.

For further detail and/or examples the *Python Mock* package has very good documentation: http://www.voidspace.org.uk/python/mock/index.html

This also shows the other potential features which might be implemented into *Mockr* in the future.

Interested readers might also want to check out the unit tests for *Mockr* as demonstrations of the functionality being modeled.

2 Making mock objects

Intro

The `Mock` object is an S4 object with two main slots:

- `call.results` An environment to record calls that are made on the `Mock`.
- `methods` An environment containing the methods which have been registered to the `Mock` along with any return values specified by the user.

The reason for setting the call.results slot as an environment is to eliminate the need to return the `Mock` from the method to capture the results. In this way the method can return something else entirely, but the call.records can still be captured in the mutable call.results environment.

The reason for using an environment for the methods slot is more one of convenience rather than necessity. This allows multiple mock objects to be assigned methods with one call the `mockMethod` without having to return them as a list or similar.

A mock object can be created simply by calling `Mock()`, and will be ready for service.

```
## Loading mockR
```

```
mock <- Mock()
str(mock)

## Formal class 'mock' [package "mockR"] with 2 slots
##   ..@ methods:<environment: 0x7fbbeabea2b8>
##   ..@ calls  :<environment: 0x7fbbeabe9128>
```

`Mock` objects by themselves are not particularly interesting. They get a little more interesting when given the name of another class to imitate.

```
s4.object <- new("MyS4object")
getSlots("MyS4object")

##  child
## "list"
```

```
# The S4 object will complain because it requires a specific type:
s4.object@child <- Mock()
```

```
## [1] "assignment of an object of class \"mock\" is not valid ..."
```

```
# But using a specification, the Mock is accepted:
s4.object@child <- Mock("list")
```

So far we know a little about the `Mock` internals, how to create them, and giving them a class to imitate. The really useful features of `Mocks` comes into play when assigning them methods.

3 Making mock methods

Making mock methods is really where *Mockr* and `Mock` objects come in handy. In essence the mock method is one which has the same name and signature of some method that already exists (or maybe is yet to be developed) but has behaviour particularly suited to testing.

In effect when a mock method is in play, it short circuits the normal operations that would be conducted, and records what happened in that method call on the mock object. This can be used to check that a particular method was called on a mock, or throw an error if an undesired method was called. And it can also be used to check that the right arguments were passed to the call. Or it may just be useful to not have to build a bunch of objects just to return a value needed for the test, the mock method can be set up to automatically return a desired value when called.

Most of the time the mock object will behave just like any other object in R. It will give a result based on what the function does.

Note that if the mock object is given a specification it becomes a sub-class of that specification. This means that it will have all the slots of the specification class, and more importantly all the existing methods (that you don't want to replace with mock methods) will, or should, behave as normal. This means that the mock can be trusted to imitate the behaviour of the specification class except for the specific methods which want to be controlled (i.e. the mock methods).

```
trySomething <- function(x) "I tried"
mock <- Mock()
trySomething(mock)

## [1] "I tried"
```

If however we tell the mock that a particular method is special, through a call to `mockMethod`, then the mock object will pay attention to the call, and will be able to report on this information.

```
my_mock_method <- function(object) "Normal function"
test_function <- function(object) {
        my_mock_method(object)
}
mock <- Mock()
mockMethod(mock, "my_mock_method")
test_function(mock)

## Error in confirm_registered_method(call):  Unexpected method call
'my'

test_function("argument")

## [1] "Normal function"
```

In the example above, we told the mock object that the function `my_mock_method` was to be turned into a mock method. In doing so, any calls to `my_mock_method` will use a special version if the first argument is a mock object, otherwise they will still act like the normal function or method.

During the setup of the mock method, within the call to `mockMethod`, the name of the method is assigned to the mock object. This becomes a "registered" method for that mock.

If we try and call this method on a mock for which it is not registered, it will generate an error.

```
mock2 <- Mock()
```

```
my_mock_method(mock2)
```

```
## [1] "Unexpected method call 'my'"
```

When a registered method is called on the mock it will record the activity and allow this to be checked later with the *Mockr* utility function `called_once` from within `expect_that`.

```
mock <- Mock()
mockMethod(mock, "plot")
# Typical usage is like this:
expect_that(mock, called_once("plot"))

## Error:  mock expected one call, was called 0 times

# In testthat it doesn't throw an error,
# but a test failure message like so:
called_once("plot")(mock)

## Not expected: expected one call, was called 0 times.

# Now we perform a call...
plot(mock)

## NULL

# and the test passes.
called_once("plot")(mock)

## As expected: called once
```

Of course the above is a trivial example, but hopefully it will become evident how this may be used to check that certain events have happened during a test.

Following are some examples of other use cases:

Assigning a method call with a return value.

```
method_return <- function(arg) "Normal"
mock <- Mock()
method_return(mock)

## [1] "Normal"
```

```
mockMethod(mock, "method_return", return.value = "Mocked!")
method_return(mock)

## [1] "Mocked!"

called_once("method_return")(mock)

## As expected: called once
```

Assigning the same method to multiple mocks at the same time.

```
mock1 <- Mock()
mock2 <- Mock()
mockMethod(list(mock1, mock2), "mock_method")
result1 <- mock_method(mock1)
result2 <- mock_method(mock2)
called_once("mock_method")(mock1)

## As expected: called once

called_once("mock_method")(mock2)

## As expected: called once
```

Checking arguments passed to mock calls.

```
mock <- Mock()
mockMethod(mock, "method_with_args")
result <- method_with_args(mock, 1, 2)
called_once_with("method_with_args", 1, 2)(mock)

## As expected: match
```

Returning different values for each mock.

```
mock1 <- Mock()
mock2 <- Mock()
mockMethod(mock1, "mock_method", return.value = "first_mock")
mockMethod(mock2, "mock_method", return.value = "second_mock")
# performing some calls...
mock_method(mock1)

## [1] "first_mock"

mock_method(mock2)

## [1] "second_mock"

mock_method(mock1)

## [1] "first_mock"
```

```
# 'mock_method' was called twice on 'mock1'
called_once("mock_method")(mock1)

## Not expected: expected one call, was called 2 times.

# this one should pass
called_once("mock_method")(mock2)

## As expected: called once
```