The Why and How of Unit Testing

Post01 for Stat 133

Salim Damerdji

Motivation

Imagine you've been working on a large project in R for weeks. All of a sudden a script doesn't work like it used to, and you're not sure how the underlying problem was introduced. You could spend hours trying to find the bug and determine what caused it.

Situations like this are why data scientists use unit testing [1]. With unit tests, programmers can run a script of tests that automatically checks all the smaller tasks - or 'units' - involved in a larger project. If an error crops up, you can quickly identify where it cropped up and what sort of problem it is.

In this post, I will discuss how to write good unit tests with the package testthat.

Background

There are three resources for unit testing in R. There is RUnit, testthat, and svUnit. In this post, I will discuss testthat, the package written by Hadley Wickham.

Hadley Wickham, the man behind testthat, ggplot2, and dplyr.

Like with any package, let's start by loading the library:

```
library(testthat)
```

```
## Warning: package 'testthat' was built under R version 3.4.2
```

You can test code in the same R script or in another R script. If you want to test code in another R script, use the following command:

```
source("file_being_tested.R")
```

If you want to test code in the same R script, no further steps are required of you.

In general, a test works by seeing whether some R code produces the expected results. When asked, testthat will output a description of which parts of your code succeed your tests and which fail. This output helps you determine where errors are introduced.

This output is organized only if you make it organized. To this end, testthat offers you the following hierarchy to organize your tests and their corresponding outputs [5] [7].

Within a **context,** there are tests. You group tests under one general context. Wickham suggests that most scripts need only one context [7]. You can set the context by declaring:

```
context("Example tests for post01")
```

The argument in context() should describe the theme that connects your tests. When you run your tests, everything will be outputted under this will label, until the script ends or a new context is set.

A **test** checks a particular functionality. Tests can take two arguments: a description of the test (that gets outputted when the tests are run); and **expectations** of how the functionality should work. Expectations are the most fine-grained this organization gets.

Here's an example of what the organization might look like. You may have a context of 'data processing;' then a test of seeing whether you can sucessfully add a column ('avg_col') that averages two other columns; and finally an expectation that avg_col is a numeric and a second expectation that avg_col actually is the average of the other two columns. *All this organization helps because it organizes the output of running*

your tests, so you can tell yourself what functionalities are tied to which test outputs.

Examples

Suppose we had the following function.

```
# outputs the hypotenuse of a right triangle
pythagoras <- function(a, b){
  if (missing(b)) { # if only one argument 'a' is passed in,
                    # the function uses 'a' twice.
    pythagoras(a,a)
  }
  else {
    sqrt(a^2 + b^2) # pythagorean theorem
  }
}
```

Let's test this code. We expect that the hypotenuse is 5 for a hypotenuse for a triangle with a side length 3 and a side length 4. So, we put 5 as the second argument in expect_equal().

We also want the output to be a numeric. Let's check this is the case by putting 'numeric' as the second argument in expect_is().

```
test_that("Check pythagoras", {
  # this 1st argument is outputted when the console reaches this line
  # it explains what the following lines describe
  expect_equal(pythagoras(3, 4), 5)
  expect_is(pythagoras(3,4), 'numeric')
})
```

The code works as expected, but it could be better. The above chunk only tested the 'else' clause in pythagoras(), but we have not confirmed that the 'if' clause works too. In other words, our test only covered 50% of the code. This is not ideal **test coverage**. [3]

The less test coverage you have, the less likely your tests will catch bugs when they crop up. So, let's fix our test with a small change:

```
test_that("Check pythagoras completely", {

  #check what happens with two arguments
  expect_equal(pythagoras(3, 4), 5)
  expect_is(pythagoras(3,4), 'numeric')

  #check what happens with one argument
  expect_equal(pythagoras(3), sqrt(18))
  expect_is(pythagoras(3), 'numeric')
})
```

So we have seen how to use expect_equal() and expect_is(). Testthat comes with many other functions you can use for expectations. Here is a short list of useful ones:

- expect_lt() checks whether the object is less than the expected numerical value.
- expect_lte() checks whether the object is less than or equal to the expected numerical value.
- expect_g() checks whether the object is greater than the expected numerical value.
- expect_gte() checks whether the object is greater than or equal to the expected numerical value.
- expect_identical() checks whether the object is identical to the expected value.
- expect_equal() is like expect_identical(), but allows for some room for error, which is helpful in case of numerical rounding.

More functions are explained in the documentation [1].

One other function that is worth noting is auto_test(). This function automatically tests your code *every* time you save an edit [6]. If you use this function, you'll never accidentally let a bug stew in your code again.

Discussion

Let's talk more about what makes for a good unit test. We have already talked about the importance of code coverage.

Another element is that your tests should break up your code into small, free-standing tasks [2]. The whole point of unit tests is to precisely pinpoint which line of code is off. You can only do this if your tests pinpoint functionalities that only take up a few lines [4].

This means you may need to create "mock objects" to test on [4]. Recall our earlier example of a data frame where you add a column ('avg_col') that is an average of two other columns. In your testing script, you don't want to use the same dataframe as the one in the script being tested on. Otherwise, your test on whether you formed avg_col correctly will depend on earlier tasks like data gathering and data tidying. So, you should create a new, simple dataframe to test on in your testing script. This way, each test isolates a unique and isolated aspect of your code.

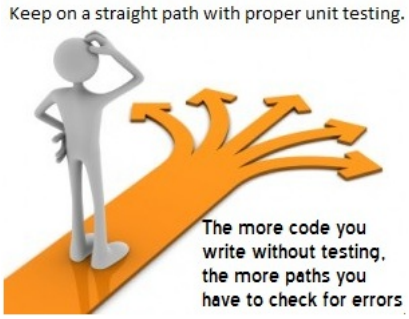
One thing to be weary of: you don't want to test absolutely everything [1]. If you do, you will have many tests that check trivially simple lines of code. This would be a waste of your time. Thus, you want to reach a balance where you test everything that could go wrong, and nothing that won't.

Conclusions

We have discussed how to use testthat to create unit tests. You can test code in the same or a different script as the code being tested on. Then you can organize your unit tests by contexts, tests, and expectations.

Good unit tests have good test coverage, fragment big tasks into many smaller ones, avoid testing trivial things, and keep the smaller tasks

independent from each other. With all this in mind, you can use unit tests to avoid getting lost solving a bug in a big project.



Don't be this person. Use unit tests!

References

[1] http://r-pkgs.had.co.nz/tests.html

[2] https://stackoverflow.com/questions/652292/what-is-unit-testing-and-how-do-you-do-it

[3] https://www.r-statistics.com/2014/11/analyzing-coverage-of-r-unit-tests-in-packages-the-testcoverage-package/

[4] https://stackoverflow.com/questions/61400/what-makes-a-good-unit-test

[5] https://katherinemwood.github.io/post/testthat/

[6] http://www.johnmyleswhite.com/notebook/2010/08/17/unit-testing-in-r-the-bare-minimum/

[7] https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf