testthat: getting started with testing

Hadley Wickham

Abstract Software testing is important, but many of us don't do it because it is frustrating and boring. **testthat** is a new testing framework for R that is easy to learn and use, and integrates with your existing workflow. This paper shows how, with illustrations from existing packages.

Introduction

Testing should be something that you do all the time, but it's normally painful and boring. `testthat` tries to make testing as painless as possible, so you do it as often as possible. To make that happen, `testthat`:

- Provides functions that make it easy to describe what you expect a function to do, including catching errors, warnings and messages.
- Easily integrates in your existing workflow, whether it's informal testing on the command line, building test suites, or using `R CMD check`.
- Can re-run tests automatically as you change your code or tests.
- Displays test progress visually, showing a pass, fail or error for every expectation. If you're using the terminal, it'll even colour the output.

`testthat` draws inspiration from the xUnit family of testing packages, as well as from many of the innovative ruby testing libraries like rspec[1], testy[2], bacon[3] and cucumber[4]. I have used what I think works for R, and abandoned what doesn't, creating a testing environment that is philosophically centred in R.

Why test?

I wrote `testthat` because I discovered I was spending too much time recreating bugs that I had previously fixed. While I was writing the original code or fixing the bug, I'd perform many interactive tests to make sure the code worked, but I never had a system for retaining these tests and running them, again and again. I think that this is a common development practice of R programmers: it's not that we don't test our code, it's that we don't store our tests so they can be re-run automatically.

In part, this is because existing R testing packages, such as RUnit (Burger et al., 2009) and svUnit (Grosjean, 2009), require a lot of up-front work to get started. One of the motivations of **testthat** is to make the initial effort as small as possible, so that you can start off slowly and gradually ramp up the formality and rigour of your tests.

It will always require a little more work to turn your casual interactive tests into reproducible scripts: you can no longer visually inspect the output, so instead you have to write code that does the inspection for you. However, this is an investment in the future of your code that will pay off in:

- Decreased frustration. Whenever I'm working to a strict deadline I always seem to discover a bug in old code. Having to stop what I'm doing to fix the bug is a real pain. This happens less when I do more testing, and I can easily see which parts of my code I can be confident in by looking at how well they are tested.
- Better code structure. Code that's easy to test is usually better designed. I have found writing tests makes me extract out the complicated parts of my code into separate functions that work in isolation. These functions are easier to test, have less duplication, are easier to understand and are easier to re-combine in new ways.
- Less struggle to pick up development after a break. If you always finish a session of coding by creating a failing test (e.g. for the feature you want to implement next) it's easy to pick up where you left off: your tests let you know what to do next.
- Increased confidence when making changes. If you know that all major functionality has a test associated with it, you can confidently make big changes with out worrying about accidentally breaking something. For me, this is particularly useful when I think of a simpler way to accomplish a task - often my simpler solution is only simpler because I've forgotten an important use case!

Test structure

testthat has a hierarchical structure made up of expectations, tests and contexts.

- An **expectation** describes what the result of a computation should be. Does it have the right

[1] http://rspec.info/
[2] http://github.com/ahoward/testy
[3] http://github.com/chneukirchen/bacon
[4] http://wiki.github.com/aslakhellesoy/cucumber/

value and right class? Does it produce error messages when you expect it to? There are 11 types of built in expectations.

- A **test** groups together multiple expectations to test one function, or tightly related functionality across multiple functions. A test is created with the `test_that` function.
- A **context** groups together multiple tests that test related functionality.

These are described in detail below. Expectations give you the tools to convert your visual, interactive experiments into reproducible scripts; tests and contexts are just ways of organising your expectations so that when something goes wrong you can easily track down the source of the problem.

Expectations

An expectation is the finest level of testing; it makes a binary assertion about whether or not a value is as you expect. An expectation is easy to read, since it is nearly a sentence already: `expect_that(a, equals(b))` reads as "I expect that a will equal b". If the expectation isn't true, **testthat** will raise an error.

There are 11 built in expectations:

- `equals()` uses `all.equal()` to check for equality with numerical tolerance.

```
# Passes
expect_that(10, equals(10))
# Also passes
expect_that(10, equals(10 + 1e-7))
# Fails
expect_that(10, equals(10 + 1e-6))
# Definitely fails!
expect_that(10, equals(11))
```

- `is_identical_to()` uses `identical()` to check for exact equality.

```
# Passes
expect_that(10, is_identical_to(10))
# Fails
expect_that(10, is_identical_to(10 + 1e-10))
```

- `is_equivalent_to()` is a more relaxed version of `equals()` that ignores attributes:

```
# Fails
expect_that(c("one" = 1, "two" = 2),
  equals(1:2))
# Passes
expect_that(c("one" = 1, "two" = 2),
  is_equivalent_to(1:2))
```

- `is_a()` checks that an object `inherit()`s from a specified class.

```
model <- lm(mpg ~ wt, data = mtcars)
# Passes
expect_that(model, is_a("lm"))
# Fails
expect_that(model, is_a("glm"))
```

- `matches()` matches a character vector against a regular expression. The optional `all` argument controls where all elements or just one element need to match. This code is powered by `str_detect()` from the **stringr** package.

```
string <- "Testing is fun!"
# Passes
expect_that(string, matches("Testing"))
# Fails, match is case-sensitive
expect_that(string, matches("testing"))
# Passes, match can be a regular expression
expect_that(string, matches("t.+ting"))
```

- `prints_text()` matches the printed output from an expression against a regular expression.

```
a <- list(1:10, letters)
# Passes
expect_that(str(a), prints_text("List of 2"))
# Passes
expect_that(str(a), prints_text(fixed("int [1:10]"))
```

- `shows_message()` checks that an expression shows a message:

```
# Passes
expect_that(library(mgcv),
  shows_message("This is mgcv"))
```

- `gives_warning()` expects that you get a warning.

```
# Passes
expect_that(log(-1), gives_warning())
expect_that(log(-1),
  gives_warning("NaNs produced"))
# Fails
expect_that(log(0), gives_warning())
```

- `throws_error()` verifies that the expression throws an error. You can also supply a regular expression which is applied to the text of the error.

```
# Fails
expect_that(1 / 2, throws_error())
# Passes
expect_that(1 / "a", throws_error())
# But better to be explicit
expect_that(1 / "a",
  throws_error("non-numeric argument"))
```

Full	Short cut
`expect_that(x, is_true())`	`expect_true(x)`
`expect_that(x, is_false())`	`expect_false(x)`
`expect_that(x, is_a(y))`	`expect_is(x, y)`
`expect_that(x, equals(y))`	`expect_equal(x, y)`
`expect_that(x, is_equivalent_to(y))`	`expect_equivalent(x, y)`
`expect_that(x, is_identical_to(y))`	`expect_identical(x, y)`
`expect_that(x, matches(y))`	`expect_matches(x, y)`
`expect_that(x, prints_text(y))`	`expect_output(x, y)`
`expect_that(x, shows_message(y))`	`expect_message(x, y)`
`expect_that(x, gives_warning(y))`	`expect_warning(x, y)`
`expect_that(x, throws_error(y))`	`expect_error(x, y)`

Table 1: Expectation shortcuts

- `is_true()` is a useful catchall if none of the other expectations do what you want - it checks that an expression is true. `is_false()` is the complement of `is_true()`.

If you don't like the readable, but verbose, `expect_that` style, you can use one of the shortcut functions described in Table 1.

You can also write your own expectations. An expectation should return a function that compares its input to the expected value and reports the result using `expectation()`. `expectation()` has two arguments: a boolean indicating the result of the test, and the message to display if the expectation fails. Your expectation function will be called by `expect_that` with a single argument: the actual value. The following code shows the simple `is_true` expectation. Most of the other expectations are equally simple, and if you want to write your own, I'd recommend reading the source code of **testthat** to see other examples.

```
is_true <- function() {
  function(x) {
    expectation(
      identical(x, TRUE),
      "isn't true"
    )
  }
}
```

Running a sequence of expectations is useful because it ensures that your code behaves as expected. You could even use an expectation within a function to check that the inputs are what you expect. However, they're not so useful when something goes wrong: all you know is that something is not as expected, not anything about where the problem is. Tests, described next, organise expectations into coherent blocks that describe the overall goal of that set of expectations.

Tests

Each test should test a single item of functionality and have an informative name. The idea is that when a test fails, you should know exactly where to look for the problem in your code. You create a new test with `test_that`, with parameters name and code block. The test name should complete the sentence "Test that" and the code block should be a collection of expectations. When there's a failure, it's the test name that will help you figure out what's gone wrong.

Figure 1 shows one test of the `floor_date` function from **lubridate** (Wickham and Grolemund, 2010). There are 7 expectations that check the results of rounding a date down to the nearest second, minute, hour, etc. Note how we've defined a couple of helper functions to make the test more concise so you can easily see what changes in each expectation.

Each test is run in its own environment so it is self-contained. The exceptions are actions which have effects outside the local environment. These include things that affect:

- the filesystem: creating and deleting files, changing the working directory, etc.
- the search path: package loading & detaching, `attach`.
- global options, like `options()` and `par()`.

When you use these actions in tests, you'll need to clean up after yourself. Many other testing packages have set-up and teardown methods that are run automatically before and after each test. These are not so important with **testthat** because you can create objects outside of the tests and rely on R's copy-on-modify semantics to keep them unchanged between test runs. To clean up other actions you can use regular R functions.

You can run a set of tests just by `source()`ing a file, but as you write more and more tests, you'll probably want a little more infrastructure. The first

```
test_that("floor_date works for different units", {
  base <- as.POSIXct("2009-08-03 12:01:59.23", tz = "UTC")

  is_time <- function(x) equals(as.POSIXct(x, tz = "UTC"))
  floor_base <- function(unit) floor_date(base, unit)

  expect_that(floor_base("second"), is_time("2009-08-03 12:01:59"))
  expect_that(floor_base("minute"), is_time("2009-08-03 12:01:00"))
  expect_that(floor_base("hour"),   is_time("2009-08-03 12:00:00"))
  expect_that(floor_base("day"),    is_time("2009-08-03 00:00:00"))
  expect_that(floor_base("week"),   is_time("2009-08-02 00:00:00"))
  expect_that(floor_base("month"),  is_time("2009-08-01 00:00:00"))
  expect_that(floor_base("year"),   is_time("2009-01-01 00:00:00"))
})
```

Figure 1: A test case from the **lubridate** package.

part of that infrastructure is contexts, described below, which give a convenient way to label each file, helping to locate failures when you have many tests.

Contexts

Contexts group tests together into blocks that test related functionality, and are established with the code `context("My context")`. Normally there is one context per file, but you can have more if you want, or you can use the same context in multiple files.

Figure 2 shows the context that tests the operation of the `str_length` function in **stringr** (Wickham, 2010). The tests are very simple, but cover two situations where `nchar()` in base R gives surprising results.

Workflow

So far we've talked about running tests by `source()`ing in R files. This is useful to double-check that everything works, but it gives you little information about what went wrong. This section shows how to take your testing to the next level by setting up a more formal workflow. There are three basic techniques to use:

- run all tests in a file or directory `test_file()` or `test_dir()`.
- automatically run tests whenever something changes with `autotest`.
- have `R CMD check` run your tests.

Testing files and directories

You can run all tests in a file with `test_file(path)`. Figure 3 shows the different between `test_file` and `source` for the tests in Figure 2, as well as those same tests for `nchar`. You can see the advantage of `test_file` over `source`: instead of seeing the first failure, you see the performance of all tests.

Each expectation is displayed as either a green dot (indicating success) or a red number (including failure). That number indexes into a list of further details, printed after all tests have been run. What you can't see is that this display is dynamic: new dot is printed each time a test passes, and it's rather satisfying to watch.

`test_dir` will run all of the test files in a directory, assuming that test files start with `test` (so it's possible to intermix regular code and tests in the same directory). This is handy if you're developing a small set of scripts rather than a complete package. The following shows the output from the **stringr** tests. You can see there are 12 contexts with between 2 and 25 expectations each. As you'd hope in a released package, all the tests pass.

```
> test_dir("inst/tests/")
String and pattern checks : ......
Detecting patterns : .........
Duplicating strings : ......
Extract patterns : ..
Joining strings : ......
String length : .........
Locations : ............
Matching groups : ..............
Test padding : ....
Splitting strings : ........................
Extracting substrings : ...................
Trimming strings : ........
```

If you want a more minimal report, suitable for display on a dashboard, you can use a different reporter. **testthat** comes with three reporters: stop, minimal and summary. The stop reporter is the default and `stop()`s whenever a failure is encountered, and the summary report is the default for `test_file` and `test_dir`. The minimal reporter is shown below: it prints `.` for success, `E` for an error and `F` a failure. The following output shows the results of running the **stringr** test suite with the minimal reporter.

```
context("String length")

test_that("str_length is number of characters", {
  expect_that(str_length("a"), equals(1))
  expect_that(str_length("ab"), equals(2))
  expect_that(str_length("abc"), equals(3))
})

test_that("str_length of missing is missing", {
  expect_that(str_length(NA), equals(NA_integer_))
  expect_that(str_length(c(NA, 1)), equals(c(NA, 1)))
  expect_that(str_length("NA"), equals(2))
})

test_that("str_length of factor is length of level", {
  expect_that(str_length(factor("a")), equals(1))
  expect_that(str_length(factor("ab")), equals(2))
  expect_that(str_length(factor("abc")), equals(3))
})
```

Figure 2: A complete context from the **stringr** package that tests the `str_length` function for computing string length.

```
> source("test-str_length.r")
> test_file("test-str_length.r")
.........

> source("test-nchar.r")
Error: Test failure in 'nchar of missing is missing'
* nchar(NA) not equal to NA_integer_
'is.NA' value mismatch: 0 in current 1 in target
* nchar(c(NA, 1)) not equal to c(NA, 1)
'is.NA' value mismatch: 0 in current 1 in target
> test_file("test-nchar.r")
...12..34

1. Failure: nchar of missing is missing ---------------------------------
nchar(NA) not equal to NA_integer_
'is.NA' value mismatch: 0 in current 1 in target

2. Failure: nchar of missing is missing ---------------------------------
nchar(c(NA, 1)) not equal to c(NA, 1)
'is.NA' value mismatch: 0 in current 1 in target

3. Failure: nchar of factor is length of level --------------------------------
nchar(factor("ab")) not equal to 2
Mean relative difference: 0.5

4. Failure: nchar of factor is length of level --------------------------------
nchar(factor("abc")) not equal to 3
Mean relative difference: 0.6666667
```

Figure 3: Results from running the `str_length` context, as well as results from running a modified version that uses `nchar`. `nchar` gives the length of `NA` as 2, and converts factors to integers before calculating length. These tests ensures that `str_length` doesn't make the same mistakes.

```
> test_dir("inst/tests/", "minimal")
```

Autotest

Tests are most useful when run frequently, and `autotest` takes that idea to the limit by rerunning your tests whenever your code or tests changes. `autotest()` has two arguments, `code_path` and `test_path`, which point to a directory of source code and tests respectively.

Once run, `autotest()` will continuously scan both directories for changes. If a test file is modified, it will test that file; if a code file is modified, it will reload that file and rerun all tests. To quit, you'll need to press Ctrl + Break on windows, Escape in the mac gui, or Ctrl + C if running from the command line.

This promotes a workflow where the *only* way you test your code is through tests. Instead of modify-save-source-check you just modify and save, then watch the automated test output for problems.

R CMD check

If you are developing a package, you can have your tests automatically run by `R CMD check`. I recommend storing your tests in `inst/tests/` (so users also have access to them), and then including one file in `tests/` that runs all of the package tests. The `test_package(package_name)` function makes this easy. It:

- Expects your tests to be in the `inst/tests/` directory
- Evaluates your tests in the package namespace (so you can test non exported functions)
- Throws an error at the end if there are any test failures. This means you'll see the full report of test failures and `R CMD check` won't pass unless all tests pass.

This setup has the additional advantage that users can make sure your package works correctly in their run-time environment.

Future work

There are two additional features I'd like to incorporate in future versions:

- Code coverage. It's very useful to be able to tell exactly what parts of your code have been tested. I'm not yet sure how to achieve this in R, but it might be possible with a combination of `RProf` and **codetools** (Tierney, 2009).
- Graphical display for `auto_test`. I find that the more visually appealing I make testing, the more fun it becomes. Coloured dots are pretty primitive, so I'd also like to provide a GUI widget that displays test output.

Bibliography

M. Burger, K. Juenemann, and T. Koenig. *RUnit: R Unit test framework*, 2009. R package version 0.4.22.

P. Grosjean. *SciViews-R: A GUI API for R*. UMH, Mons, Belgium, 2009.

L. Tierney. *codetools: Code Analysis Tools for R*, 2009. R package version 0.2-2.

H. Wickham. *stringr: Make it easier to work with strings.*, 2010. R package version 0.4.

H. Wickham and G. Grolemund. *lubridate: Make dealing with dates a little easier*, 2010. R package version 0.1.

Hadley Wickham
Department of Statistics
Rice University
6100 Main St MS#138
Houston TX 77005-1827
USA
hadley@rice.edu