R packages (/) by Hadley Wickham

Table of contents

Contents

How to contribute (/contribute.html)

Edit this page (https://github.com/hadley/r-pkgs/edit/master/tests.rmd)

Testing

Testing is a vital part of package development. It ensures that your code does what you want it to do. Testing, however, adds an additional step to your development workflow. The goal of this chapter is to show you how to make this task easier and more effective by doing formal automated testing using the testthat package.

Up until now, your workflow probably looks like this:

1. Write a function.
2. Load it with Ctrl/Cmd + Shift + L or `devtools::load_all()`.
3. Experiment with it in the console to see if it works.
4. Rinse and repeat.

While you *are* testing your code in this workflow, you're only doing it informally. The problem with this approach is that when you come back to this code in 3 months time to add a new feature, you've probably forgotten some of the informal tests you ran the first time around. This makes it very easy to break code that used to work.

I started using automated tests because I discovered I was spending too much time re-fixing bugs that I'd already fixed before. While writing code or fixing bugs, I'd perform interactive tests to make sure the code worked. But I never had a system which could store those tests so I could re-run them as needed. I think that this is a common practice among R programmers. It's not that you don't test your code, it's that you don't automate your tests.

In this chapter you'll learn how to graduate from using informal ad hoc testing, done at the command line, to formal automated testing (aka unit testing). While turning casual interactive tests into reproducible scripts requires a little more work up front, it pays off in four ways:

- Fewer bugs. Because you're explicit about how your code should behave you will have fewer bugs. The reason why is a bit like the reason double entry book-keeping works: because you describe the behaviour of your code in two places, both in your code and in your tests, you are able to check one against the other. By following this approach to testing, you can be sure that bugs that you've fixed in the past will never come back to haunt you.

- Better code structure. Code that's easy to test is usually better designed. This is because writing tests forces you to break up complicated parts of your code into separate functions that can work in isolation. This reduces duplication in your code. As a result, functions will be easier to test, understand and work with (it'll be easier to combine them in new ways).

- Easier restarts. If you always finish a coding session by creating a failing test (e.g. for the next feature you want to implement), testing makes it easier for you to pick up where you left off: your tests will let you know what to do next.

- Robust code. If you know that all the major functionality of your package has an associated test, you can confidently make big changes without worrying about accidentally breaking something. For me, this is particularly useful when I think I have a simpler way to accomplish a task (usually the reason my solution is simpler is that I've forgotten an important use case!).

If you're familiar with unit testing in other languages, you should note that there are some fundamental differences with testthat. This is because R is, at heart, more a functional programming language than an object oriented programming language. For instance, because R's main OO systems (S3 and S4) are based on generic functions (i.e., methods belong to functions not classes), testing approaches built around objects and methods don't make much sense.

Test workflow

To set up your package to use testthat, run:

```
devtools::use_testthat()
```

This will:

1. Create a `tests/testthat` directory.
2. Adds testthat to the `Suggests` field in the `DESCRIPTION`.
3. Creates a file `tests/testthat.R` that runs all your tests are when `R CMD check` runs. (You'll learn more about that in automated checking (check.html#check).)

Once you're set up the workflow is simple:

1. Modify your code or tests.
2. Test your package with Ctrl/Cmd + Shift + T or `devtools::test()`.
3. Repeat until all tests pass.

The testing output looks like this:

```
Expectation : ...........
rv : ...
Variance : ....123.45.
```

Each line represents a test file. Each `.` represents a passed test. Each number represents a failed test. The numbers index into a list of failures that provides more details:

```
1. Failure(@test-variance.R#22): Variance correct for discrete uniform rvs -----
VAR(dunif(0, 10)) not equal to var_dunif(0, 10)
Mean relative difference: 3

2. Failure(@test-variance.R#23): Variance correct for discrete uniform rvs -----
VAR(dunif(0, 100)) not equal to var_dunif(0, 100)
Mean relative difference: 3.882353
```

Each failure gives a description of the test (e.g., "Variance correct for discrete uniform rvs"), its location (e.g., "@test-variance.R#22"), and the reason for the failure (e.g., "VAR(dunif(0, 10)) not equal to var_dunif(0, 10)"). The goal is to pass all the tests.

Test structure

A test file lives in `tests/testthat/`. Its name must start with `test`. Here's an example of a test file from the stringr package:

```
library(stringr)
context("String length")

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

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

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

Tests are organised hierarchically: **expectations** are grouped into **tests** which are organised in **files**:

- An **expectation** is the atom of testing. It describes the expected result of a computation: Does it have the right value and right class? Does it produce error messages when it should? An expectation automates visual checking of results in the console. Expectations are functions that start with `expect_`.
- A **test** groups together multiple expectations to test the output from a simple function, a range of possibilities for a single parameter from a more complicated function, or tightly related functionality from across multiple functions. This is why they are sometimes called **unit** as they test one unit of functionality. A test is created with `test_that()` .
- A **file** groups together multiple related tests. Files are given a human readable name with `context()`.

These are described in detail below.

Expectations

An expectation is the finest level of testing. It makes a binary assertion about whether or not a function call does what you expect. All expectations have a similar structure:

- They start with `expect_`.
- They have two arguments: the first is the actual result, the second is what you expect.

- If the actual and expected results don't agree, testthat throws an error.

While you'll normally put expectations inside tests inside files, you can also run them directly. This makes it easy to explore expectations interactively. There are almost 20 expectations in the testthat package. The most important are discussed below.

- There are two basic ways to test for equality: `expect_equal()`, and `expect_identical()`. `expect_equal()` is the most commonly used: it uses `all.equal()` to check for equality within a numerical tolerance:

  ```
  expect_equal(10, 10)
  expect_equal(10, 10 + 1e-7)
  expect_equal(10, 11)
  #> Error: 10 not equal to 11
  #> Mean relative difference: 0.09090909
  ```

  If you want to test for exact equivalence, or need to compare a more exotic object like an environment, use `expect_identical()`. It's built on top of `identical()`:

  ```
  expect_equal(10, 10 + 1e-7)
  expect_identical(10, 10 + 1e-7)
  #> Error: 10 is not identical to 10 + 1e-07. Differences:
  #> Objects equal but not identical
  ```

- `expect_match()` matches a character vector against a regular expression. The optional `all` argument controls whether all elements or just one element needs to match. This is powered by `grepl()` (additional arguments like `ignore.case = FALSE` or `fixed = TRUE` are passed on down).

  ```
  string <- "Testing is fun!"

  expect_match(string, "Testing")
  # Fails, match is case-sensitive
  expect_match(string, "testing")
  #> Error: string does not match 'testing'. Actual value: "Testing is fun!"

  # Additional arguments are passed to grepl:
  expect_match(string, "testing", ignore.case = TRUE)
  ```

- Four variations of `expect_match()` let you check for other types of result: `expect_output()`, inspects printed output; `expect_message()`, messages; `expect_warning()`, warnings; and `expect_error()` errors.

```
a <- list(1:10, letters)

expect_output(str(a), "List of 2")
expect_output(str(a), "int [1:10]", fixed = TRUE)

expect_message(library(mgcv), "This is mgcv")
```

With `expect_message()`, `expect_warning()`, `expect_error()` you can leave the second argument blank if you just want to see if a message, warning or error is created. However, it's normally better to be explicit, and provide some text from the message.

```
expect_warning(log(-1))
expect_error(1 / "a")

# But always better to be explicit
expect_warning(log(-1), "NaNs produced")
expect_error(1 / "a", "non-numeric argument")

# Failure to produce a warning or error when expected is an error
expect_warning(log(0))
#> Error: log(0) no warnings given
expect_error(1 / 2)
#> Error: 1/2 code did not generate an error
```

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

```
model <- lm(mpg ~ wt, data = mtcars)
expect_is(model, "lm")
expect_is(model, "glm")
#> Error: model inherits from lm not glm
```

- `expect_true()` and `expect_false()` are useful catchalls if none of the other expectations do what you need.
- Sometimes you don't know exactly what the result should be, or it's too complicated to easily recreate in code. In that case the best you can do is check that the result is the same as last time. `expect_equal_to_reference()` caches the result the first time its run, and then compares it to subsequent runs. If for some reason the result does change, just delete the cache (*) file and re-test.

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. You don't know the goal of the expectation. Tests, described next, organise expectations into coherent blocks that describe the overall goal of a set of expectations.

Writing tests

Each test should have an informative name and cover a single unit of functionality. The idea is that when a test fails, you'll know what's wrong and where in your code to look for the problem. You create a new test using `test_that()`, with test name and code block as arguments. The test name should complete the sentence "Test that …". The code block should be a collection of expectations.

It's up to you how to organise your expectations into tests. The main thing is that the message associated with the test should be informative so that you can quickly narrow down the source of the problem. Try to avoid putting too many expectations in one test - it's better to have more smaller tests than fewer larger tests.

Each test is run in its own environment and is self-contained. However, testthat doesn't know how to cleanup after actions affect the R landscape:

- The filesystem: creating and deleting files, changing the working directory, etc.
- The search path: `library()`, `attach()`.
- Global options, like `options()` and `par()`.

When you use these actions in tests, you'll need to clean up after yourself. While 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 you can 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.

What to test

> Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead. — Martin Fowler

There is a fine balance to writing tests. Each test that you write makes your code less likely to change inadvertently; but it also can make it harder to change your code on purpose. It's hard to give good general advice about writing tests, but you might find these points helpful:

- Focus on testing the external interface to your functions - if you test the internal interface, then it's harder to change the implementation in the future because as well as modifying the code, you'll also need to update all the tests.
- Strive to test each behaviour in one and only one test. Then if that behaviour later changes you only need to update a single test.

- Avoid testing simple code that you're confident will work. Instead focus your time on code that you're not sure about, is fragile, or has complicated interdependencies. That said, I often find I make the most mistakes when I falsely assume that the problem is simple and doesn't need any tests.
- Always write a test when you discover a bug. You may find it helpful to adopt the test-first philospphy. There you always start by writing the tests, and then write the code that makes them pass. This reflects an important problem solving strategy: start by establishing your success critieria, how you know if you've solved the problem.

Skipping a test

Sometimes it's impossible to perform a test - you may not have an internet connection or you may be missing an important file. Unfortunately, another likely reason follows from this simple rule: the more machines you use to write your code, the more likely it is that you won't be able to run all of your tests. In short, there are times when, instead of getting a failure, you just want to skip a test. To do that, you can use the `skip()` function - rather than throwing an error it simply prints an `S` in the output.

```
check_api <- function() {
  if (not_working()) {
    skip("API not available")
  }
}

test_that("foo api returns bar when given baz", {
  check_api()
  ...
})
```

Building your own testing tools

As you start to write more tests, you might notice duplication in your code. For example, the following code shows one test of the `floor_date()` function from `library(lubridate)`. There are seven expectations that check the results of rounding a date down to the nearest second, minute, hour, etc. There's a lot of duplication (which increases the chance of bugs), so we might want to extract common behaviour into a new function.

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

  expect_equal(floor_date(base, "second"),
    as.POSIXct("2009-08-03 12:01:59", tz = "UTC"))
  expect_equal(floor_date(base, "minute"),
    as.POSIXct("2009-08-03 12:01:00", tz = "UTC"))
  expect_equal(floor_date(base, "hour"),
    as.POSIXct("2009-08-03 12:00:00", tz = "UTC"))
  expect_equal(floor_date(base, "day"),
    as.POSIXct("2009-08-03 00:00:00", tz = "UTC"))
  expect_equal(floor_date(base, "week"),
    as.POSIXct("2009-08-02 00:00:00", tz = "UTC"))
  expect_equal(floor_date(base, "month"),
    as.POSIXct("2009-08-01 00:00:00", tz = "UTC"))
  expect_equal(floor_date(base, "year"),
    as.POSIXct("2009-01-01 00:00:00", tz = "UTC"))
})
```

I'd start by defining a couple of helper functions to make each expectation more concise. That allows each test to fit on one line, so you can line up actual and expected values to make it easier to see the differences:

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

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

We could go a step further and create a custom expectation function:

```
base <- as.POSIXct("2009-08-03 12:01:59.23", tz = "UTC")

expect_floor_equal <- function(unit, time) {
  expect_equal(floor_date(base, unit), as.POSIXct(time, tz = "UTC"))
}
expect_floor_equal("year", "2009-01-01 00:00:00")
```

However, if the expectation fails this doesn't give very informative output:

```
expect_floor_equal("year", "2008-01-01 00:00:00")
#> Error: floor_date(base, unit) not equal to as.POSIXct(time, tz = "UTC")
#> Mean absolute difference: 31622400
```

Instead you can use a little non-standard evaluation (http://adv-r.had.co.nz/Computing-on-the-language.html) to produce something more informative. The key is to use `bquote()` and `eval()`. In the `bquote()` call below, note the use of `.(x)` - the contents of `()` will be inserted into the call.

```
expect_floor_equal <- function(unit, time) {
  as_time <- function(x) as.POSIXct(x, tz = "UTC")
  eval(bquote(expect_equal(floor_date(base, .(unit)), as_time(.(time)))))
}
expect_floor_equal("year", "2008-01-01 00:00:00")
#> Error: floor_date(base, "year") not equal to as_time("2008-01-01 00:00:00")
#> Mean absolute difference: 31622400
```

This sort of refactoring is often worthwhile because removing redundant code makes it easier to see what's changing. Readable tests give you more confidence that they're correct.

```
test_that("floor_date works for different units", {
  as_time <- function(x) as.POSIXct(x, tz = "UTC")
  expect_floor_equal <- function(unit, time) {
    eval(bquote(expect_equal(floor_date(base, .(unit)), as_time(.(time)))))
  }

  base <- as_time("2009-08-03 12:01:59.23")
  expect_floor_equal("second", "2009-08-03 12:01:59")
  expect_floor_equal("minute", "2009-08-03 12:01:00")
  expect_floor_equal("hour",   "2009-08-03 12:00:00")
  expect_floor_equal("day",    "2009-08-03 00:00:00")
  expect_floor_equal("week",   "2009-08-02 00:00:00")
  expect_floor_equal("month",  "2009-08-01 00:00:00")
  expect_floor_equal("year",   "2009-01-01 00:00:00")
})
```

Test files

The highest-level structure of tests is the file. Each file should contain a single `context()` call that provides a brief description of its contents. Just like the files in the `R/` directory, you are free to organise your tests any way that you like. But again, the two extremes are clearly bad (all tests in one file, one file per test). You need to find a happy medium that works for you. A good starting place is to have one file of tests for each complicated function.

CRAN notes

CRAN will run your tests on all CRAN platforms: Windows, Mac, Linux and Solaris. There are a few things to bear in mind:

- Tests need to run relatively quickly - aim for under a minute. Place `skip_on_cran()` at the beginning of long-running tests that shouldn't be run on CRAN - they'll still be run locally, but not on CRAN.
- Note that tests are always run in the English language (`LANGUAGE=EN`) and with C sort order (`LC_COLLATE=C`). This minimises spurious differences between platforms.
- Be careful about testing things that are likely to be variable on CRAN machines. It's risky to test how long something takes (because CRAN machines are often heavily loaded) or to test parallel code (because CRAN runs multiple package tests in parallel, multiple cores will not always available). Numerical precision can also vary across platforms (it's often less precise on 32-bit versions of R) so use `expect_equal()` rather than `expect_identical()`.