Assertions and Testing

Jared Bennett

10 September, 2019

References

- Package Tutorial
- testthat Paper
- assertthat github
- Package Example

Assertion vs. Testing

One uses assertions to check inputs and the state of the program during execution, while tests check that a function behaves as expected given a set of inputs (including checking that the assertions work and that what should be errors are indeed errors).

Tests and Assertions are similar in that,

- Both are part of specification.
- They are helpful during design, but in different ways.

Tests and Assertions are different in that,

- Usually, assertions have a smaller span, because an assertion in a method must depend only on the object and method parameters, otherwise you get too many dependencies.
- Assertions belong to the class, so can document class properties and assumptions that are not public. Unit tests can only check externally visible properties.
- Assertions are part of executing code, so the amount of computation you can do in an assertion is limited. You can eliminate assertions from product code in the name of speed, but then you sacrifice graceful error handling.
- Tests only test a small number of cases. Assertions work with live data, which can be more varied and more realistic. They purport to cover infinitely many cases; they establish universal truths rather then point truths.

Assertions and `"assertthat"`

Learning Objectives

- Benefits of Assertions
- Introduction to the R package "assertthat"
- Write simple functions and their assertions

assertthat provides a drop in replacement for `stopifnot()` that makes it easy to check the pre- and post-conditions of a function, while producing useful error messages.

```
x <- 1:10
stopifnot(is.character(x))
```

```
## Error: is.character(x) is not TRUE
```

```
assert_that(is.character(x))
```

```
## Error: x is not a character vector
```

```
assert_that(length(x) == 5)
```

```
## Error: length(x) not equal to 5
```

```
assert_that(is.numeric(x))
```

```
## [1] TRUE
```

This is a good defensive programming technique, and is useful as source-code documentation: you can see exactly what your function expects when you come back to it in the future. It is partly a response to the lack of static typing in R.

assertthat can be installed either from CRAN or github (CRAN is the stable version, github usually has the current dev version):

```
install.packages('assertthat')
```

Some Useful Assertions

As well as all the functions provided by R, assertthat provides a few more that are useful:

- `is.flag(x)`: is x TRUE or FALSE? (a boolean flag)
- `is.string(x)`: is x a length 1 character vector?
- `has_name(x, nm)`, `x %has_name% nm`: does x have component nm?
- `has_attr(x, attr)`, `x %has_attr% attr`: does x have attribute attr?
- `is.count(x)`: is x a single positive integer?
- `are_equal(x, y)`: are x and y equal?
- `not_empty(x)`: are all dimensions of x greater than 0?
- `noNA(x)`: is x free from missing values?
- `is.dir(path)`: is path a directory?
- `is.writeable(path)`/`is.readable(path)`: is path writeable/readable?
- `has_extension(path, extension)`: does file have given extension?

`assert_that`, `see_if`, and `validate_that`

There are three main functions in assertthat:

- `assert_that()` signals an error
- `see_if()` returns a logical value, with the error message as an attribute.
- `validate_that()` returns TRUE on success, otherwise returns the error as a string.

Writing Your Own Assertions

If you're writing your own assertions, you can provide custom error messages using the `on_failure()` helper function:

```
is_odd <- function(x) {
  assert_that(is.numeric(x), length(x) == 1)
  x %% 2 == 1
}
assert_that(is_odd(2))
```

```
## Error: is_odd(x = 2) is not TRUE
```

```
on_failure(is_odd) <- function(call, env) {
  paste0(deparse(call$x), " is even")
}
assert_that(is_odd(2))
```

```
## Error: 2 is even
```

The on_failure callback is called with two arguments, the unevaluated function call , and env, and the environment in which the assertion was executed. This allows you to choose between displaying values or names in your error messages.

Also note the use of assert_that() in the new function: assertions flow through function calls ensuring that you get a useful error message at the top level:

```
assert_that(is_odd("b"))
```

```
## Error: x is not a numeric or integer vector
```

```
assert_that(is_odd(1:2))
```

```
## Error: length(x) not equal to 1
```

Testing and "testthat"

Learning Objectives

- Benefits of Unit Tests
- Introduction to the R package "testthat"
- Write simple functions and their unit tests
- Test your code

Now 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!).

About "testthat"

- "testthat" provides a testing framework for R that is easy to learn and use
- "testthat" has a hierarchical structure made up of:
  - expectations
  - tests
  - contexts
- A **context** involves **tests** formed by groups of **expectations**
- Each structure has associated functions:
  - `expect_that()` for expectations
  - `test_that()` for groups of tests
  - `context()` for contexts

```
# remember to install "testthat"
install.packages("testthat")

# load it
library(testthat)
```

List of Common Expectation Functions

Function	Description
`expect_true(x)`	expects that `x` is `TRUE`
`expect_false(x)`	expects that `x` is `FALSE`
`expect_null(x)`	expects that `x` is `NULL`
`expect_type(x)`	expects that `x` is of type `y`
`expect_is(x, y)`	expects that `x` is of class `y`
`expect_length(x, y)`	expects that `x` is of length `y`
`expect_equal(x, y)`	expects that `x` is equal to `y`
`expect_equivalent(x, y)`	expects that `x` is equivalent to `y`
`expect_identical(x, y)`	expects that `x` is identical to `y`
`expect_lt(x, y)`	expects that `x` is less than `y`
`expect_gt(x, y)`	expects that `x` is greater than `y`
`expect_lte(x, y)`	expects that `x` is less than or equal to `y`
`expect_gte(x, y)`	expects that `x` is greater than or equal `y`
`expect_named(x)`	expects that `x` has names `y`
`expect_matches(x, y)`	expects that `x` matches `y` (regex)
`expect_message(x, y)`	expects that `x` gives message `y`
`expect_warning(x, y)`	expects that `x` gives warning `y`
`expect_error(x, y)`	expects that `x` throws error `y`

Motivation

To understand how "testthat" works, let's start with the `standardize()` function.

```
#' @title Standardize
#' @description Computes z-scores (scores in standard units)
#' @param x numeric vector
#' @param na.rm whether to remove missing values
#' @return vector of standard scores
#' @examples
#'  a <- runif(5)
```

```
#'  z <- standardize(a)
#'  mean(z)
#'  sd(z)
standardize <- function(x, na.rm = FALSE) {
  z <- (x - mean(x, na.rm = na.rm)) / sd(x, na.rm = na.rm)
  return(z)
}
```

When writing a function, we typically test it like this:

```
a <- c(2, 4, 7, 8, 9)
z <- standardize(a)
z
```

```
## [1] -1.3719887 -0.6859943  0.3429972  0.6859943  1.0289915
```

We can check the mean and standard deviation of `z` to make sure `standardize()` works correctly:

```
# zero mean
mean(z)
```

```
## [1] 0
```

```
# unit std-dev
sd(z)
```

```
## [1] 1
```

Then we keep testing a function with more extreme cases:

```
y <- c(1, 2, 3, 4, NA)
standardize(y)
```

```
## [1] NA NA NA NA NA
```

```
standardize(y, na.rm = TRUE)
```

```
## [1] -1.1618950 -0.3872983  0.3872983  1.1618950         NA
```

and even more cases:

```
alog <- c(TRUE, FALSE, FALSE, TRUE)
standardize(alog)
```

```
## [1]  0.8660254 -0.8660254 -0.8660254  0.8660254
```

Writing Tests

Instead of writing a list of more or less informal test, we are going to use the functions provide by `"testthat"`.

To learn about the testing functions, we'll consider the following testing vectors:

- `x <- c(1, 2, 3)`
- `y <- c(1, 2, NA)`
- `w <- c(TRUE, FALSE, TRUE)`
- `q <- letters[1:3]`

Testing with "normal" Input

The core of `"testthat"` consists of **expectations**; to write expectations you use functions of the form `expect_xyz()` such as `expect_equal()`, `expect_integer()` or `expect_error()`.

```
x <- c(1, 2, 3)
z <- (x - mean(x)) / sd(x)

expect_equal(standardize(x), z)
expect_length(standardize(x), length(x))
expect_type(standardize(x), 'double')
```

Notice that when an expectation runs successfully, nothing appears to happen. But that's good news. If an expectation fails, you'll typically get an error, here are some failed tests:

```
# different expected output
expect_equal(standardize(x), x)
```

```
## Error: standardize(x) not equal to `x`.
## 3/3 mismatches (average diff: 2)
## [1] -1 - 1 == -2
## [2]  0 - 2 == -2
## [3]  1 - 3 == -2
```

```
# different expected length
expect_length(standardize(x), 2)
```

```
## Error: standardize(x) has length 3, not length 2.
```

```
# different expected type
expect_type(standardize(x), 'character')
```

```
## Error: standardize(x) has type `double`, not `character`.
```

Testing with "missing values"

Let's include a vector with missing values

```
y <- c(1, 2, NA)
z1 <- (y - mean(y, na.rm = FALSE)) / sd(y, na.rm = FALSE)
z2 <- (y - mean(y, na.rm = TRUE)) / sd(y, na.rm = TRUE)

expect_equal(standardize(y), z1)
expect_length(standardize(y), length(y))
expect_equal(standardize(y, na.rm = TRUE), z2)
expect_length(standardize(y, na.rm = TRUE), length(y))
expect_type(standardize(y), 'double')
```

Testing with "logical" input

Let's now test `standardize()` with a logical vector:

```
w <- c(TRUE, FALSE, TRUE)
z <- (w - mean(w)) / sd(w)

expect_equal(standardize(w), z)
```

```r
expect_length(standardize(w), length(w))
expect_type(standardize(w), 'double')
```

Function `test_that()`

Now that you've seen how the expectation functions work, the next thing to talk about is the function `test_that()` which you'll use to group a set of expectations

Looking at the previous test examples with the "normal" input vector, all the expectations can be wrapped inside a call to `test_that()`. The first argument of `test_that()` is a string indicating what is being tested, followed by an R expression with the expectations.

```r
test_that("standardize works with normal input", {
  x <- c(1, 2, 3)
  z <- (x - mean(x)) / sd(x)

  expect_equal(standardize(x), z)
  expect_length(standardize(x), length(x))
  expect_type(standardize(x), 'double')
})
```

Likewise, all the expectations with the vector containing missing values can be wrapped inside another call to `test_that()` like this:

```r
test_that("standardize works with missing values", {
  y <- c(1, 2, NA)
  z1 <- (y - mean(y, na.rm = FALSE)) / sd(y, na.rm = FALSE)
  z2 <- (y - mean(y, na.rm = TRUE)) / sd(y, na.rm = TRUE)

  expect_equal(standardize(y), z1)
  expect_length(standardize(y), length(y))
  expect_equal(standardize(y, na.rm = TRUE), z2)
  expect_length(standardize(y, na.rm = TRUE), length(y))
  expect_type(standardize(y), 'double')
})
```

And last, but not least, the expectations with the logical vector can be grouped in a `test_that()` call:

```r
test_that("standardize handles logical vector", {
  w <- c(TRUE, FALSE, TRUE)
  z <- (w - mean(w)) / sd(w)

  expect_equal(standardize(w), z)
  expect_length(standardize(w), length(w))
  expect_type(standardize(w), 'double')
})
```

Testing Structure

As we mentioned in the introduction, there is a hierarchical structure for the tests that is made of *expectations* that are grouped in *tests*, which are in turn considered to be part of some *context*. In other words:

> A **context** involves **tests** formed by groups of **expectations**

The formal way to implement the tests is to include them in a separate `R` script file, e.g. `tests.R`.

Suppose you are working on a project with some file structure like the one below. Automated tests are stored in a `test/` directory, containing the auto-run script and your test functions:

```
project/
  R/
  data/
  man/
  src/
  tests/
    testthat/
      test-myFunc1.R
      test-myFunc2.R
    testthat.R
  DESCRIPTION
  NEWS
  NAMESPACE
  ...
```

The content of `testthat.R` may look like this:

```
library(testthat)
test_check("project")
```

While the content of `test-myFunc1.R` will look similar to:

```
# context with one test that groups expectations
context("Tests for Standardize")

test_that("standardize works with normal input", {
  x <- c(1, 2, 3)
  z <- (x - mean(x)) / sd(x)

  expect_equal(standardize(x), z)
  expect_length(standardize(x), length(x))
  expect_type(standardize(x), 'double')
})


test_that("standardize works with missing values", {
  y <- c(1, 2, NA)
  z1 <- (y - mean(y, na.rm = FALSE)) / sd(y, na.rm = FALSE)
  z2 <- (y - mean(y, na.rm = TRUE)) / sd(y, na.rm = TRUE)

  expect_equal(standardize(y), z1)
  expect_length(standardize(y), length(y))
  expect_equal(standardize(y, na.rm = TRUE), z2)
  expect_length(standardize(y, na.rm = TRUE), length(y))
  expect_type(standardize(y), 'double')
})


test_that("standardize handles logical vector", {
  w <- c(TRUE, FALSE, TRUE)
  z <- (w - mean(w)) / sd(w)

  expect_equal(standardize(w), z)
```

```
  expect_length(standardize(w), length(w))
  expect_type(standardize(w), 'double')
})
```

Runing the tests

If your working directory is the `code/` directory, then you could run the tests in `tests.R` from the R console using the function `test_file()`

```
# (assuming that your working directory is "code/")
# run from R console
library(testthat)
test_file("tests.R")
```

```
## v |  OK F W S | Context
##
/ |   0       | Tests for Standardize
v |  11       | Tests for Standardize
##
## == Results =====================================================================================
## OK:       11
## Failed:   0
## Warnings: 0
## Skipped:  0
```