Assertions and Testing

Zoe Vernon

10 September, 2020

References and useful links

- Testing section of the R packages tutorial by Hadley Wickham
- GitHub for assertthat package by Hadley Wickham
- Assertions and testing tutorial in Python

Learning Objectives

- Understand the benefits of assertions and testings as well as the differences between the two.
- Introduction to the R package `assertthat`.
- Introduction to the R package `testthat`.
- Practice writing assertions and tests on your own.

Assertion vs. Testing

Assertions check the internal state of a function. For example, consider a function `add(x, y)` which returns `x + y`. The function assumes `x` is numeric, and an assertion would confirm that this is in the case and return as error if not. On the other hand, tests check that a function produces the expected output for various inputs. For example, ensuring that `add(1, 2)` returns the number 3. Tests may include checks that assertions are working properly.

Tests and assertions are similar in that,

- Both are part of ensuring programs run correctly and aspects of defensive programming.

- Both should check small pieces of the code while providing useful error messages, so they tell you exactly where the issue arises.

A couple of important differences between assertions and tests are below:

Assertions	Testing
Depends only on the object and method parameters.	Can depend on global variables.
Document function properties that are not public.	Can only check externally visible properties.
Work with live data, so check cover infinitely many cases.	Test a small number of cases.
Excecuted in function calls, so amount of computation should be limited.	

Assertions and `assertthat`

An assertion is a statement in a function or progam that must be true for it to continue. There are three types of assertions:

1. Pre-conditions: statements that must be true at the beginning of the function for it to work. Mostly, this involves checking that inputs to the function are in the expected form.
2. Invariants: statements that must be true at intermediate points in the function. For example, checking that the output from a computation is positive before using the `sqrt()` function.
3. Post-conditions: statements that must be true at the end of a function. For example, if you write a function that must return a vector of length `n` with all positive numbers you would ensure that is the case after all the computation has been performed.

`assertthat` is a R package that provides functionality for adding assertions to functions, while producing useful error messages. Calls to `assert_that` are similar to `stopifnot` function from base R. Consider the examples below:

```
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
```

In addition to giving useful error messages adding assertions to your function allows you to document exactly what your function expects. This is particularly useful if you come back to the function after a while and need to recall exactly what it does.

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

```
install.packages('assertthat')
devtools::install_github("hadley/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?

Three main functions: `assert_that`, `see_if`, and `validate_that`

These are the three primary functions from the package:

- `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.

Here is an example of the differences. When the assertion is `TRUE` they all return `TRUE` and continue with the excecution of the function.

```r
# example functions to see differences in assertthat functions
returnStringAssert <- function(x){
  assert_that(is.string(x))

  return(x)
}
returnStringSeeIf <- function(x){
  see_if(is.string(x))

  return(x)
}
returnStringValidate <- function(x){
  validate_that(is.string(x))

  return(x)
}

returnStringAssert("a")
```

```
## [1] "a"
```

```r
returnStringSeeIf("a")
```

```
## [1] "a"
```

```r
returnStringValidate("a")
```

```
## [1] "a"
```

When the assertion is `FALSE` the functions have different output and function. `assert_that` will return an error and halt excecution of the function. `see_if` and `validate_that` will not stop the excecution.

```r
returnStringAssert(c("a", "b"))
```

```
## Error: x is not a string (a length one character vector).
```

```r
returnStringSeeIf(c("a", "b"))
```

```
## [1] "a" "b"
```

```r
returnStringValidate(c("a", "b"))
```

```
## [1] "a" "b"
```

However, when called outside of function they will give the error messages as described above.

`assert_that` returns an error

```r
assert_that(is.string(c("a", "b")))
```

```
## Error: c("a", "b") is not a string (a length one character vector).
```

`see_if` returns `FALSE` with an error message attribute

```
see_if(is.string(c("a", "b")))
```

```
## [1] FALSE
## attr(,"msg")
## [1] "c(\"a\", \"b\") is not a string (a length one character vector)."
```

`validate_that` returns the error message as a string

```
validate_that(is.string(c("a", "b")))
```

```
## [1] "c(\"a\", \"b\") is not a string (a length one character vector)."
```

Writing Your Own Assertions

You can also write you own assertions with custom error messages. There are two ways to do this. The first is using the `on_failure()` function. Below is an example of how this works:

```
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
```

Also note theat the assertions from our original `is_odd()` function flow through the function call from `on_failure()`, so we still get the appropriate error messages when we pass a non-numeric or vector value to `is_odd()`.

Another, option is to add a new assertion that checks whether the number is odd and add a custome message directly to the assertion:

```
is_odd2 <- function(x) {
  assert_that(is.numeric(x), length(x) == 1)
  assert_that(x %% 2 == 1, msg = paste(x, "is even"))
  x %% 2 == 1
}
assert_that(is_odd2(2))
```

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

Testing

Assertions allow us to check aspects of functions as they are being excuted, while unit tests help ensure the output from a function is what we expect. A common approach to testing is to use the command line to informally check whether your code works on a few examples. Units tests are a more formal framework for testing that allows you to continue running the same tests as you update your function. Hadley Wickam describes four main areas that proper testings will help improve your code:

1. **Fewer bugs**: When setting up unit tests you have a formal place that describes your expectation of function behavior. Having two places where the function is document allows you to check one against the other.

2. **Better code structure**: Tests should only check accuracy of small portions of code, so that you can easily find the source of error. This forces you to write more modular code.
3. **Easier restarts**: Tests help you remember where you left off and what the next step in your code should be. It is good practice to write tests first, followed by the function to execute the desired result.

4. **Robust code**: By having tests in place for all portions of your code you can make changes while knowing that you can easily check if those changes produce an error and where to go to fix it.

`testthat`

The `testthat` package provides a framework for writing and performing tests in R. There are two pieces of the `testthat` package, which form a hierarchal structure for doing testing.

1. Tests: tests are the top of the hiercharchy. Usually for a single function that is being tested there will be multiple tests. For example, we may have one test that inspects results for normal inputs and another test for inputs with missing values. Use the `test_that()` function.

2. Expectations: each test is made up of a series of expectations that describe the expected output of a function (e.g. length, type, value). Use the `expect_that()` function.

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`

`testthat` example

To understand how `testthat` works, we will consider the `standardize()` function, which takes a vector `x`, subtracts the mean of the vector, and then divides by the standard deviation. Notice the assertions in the function checking pre-conditions!

```r
standardize <- function(x, na.rm = FALSE) {
  # assertions on input
  assert_that(is.vector(x))
  assert_that(is.flag(na.rm))

  # do computation
  z <- (x - mean(x, na.rm = na.rm)) / sd(x, na.rm = na.rm)
  return(z)
}
```

Informal testing

When writing a function, we informally testings usually looks something like this:

```r
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:

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

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

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

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

Then we keep testing a function with more extreme cases:

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

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

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

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

and even more cases:

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

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

Using `testthat` instead

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)
```

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

Combining multiple expectations into a test with `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')
})
```

Running tests

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

If your working directory is the `sections/03/` directory, then you could run the tests in `tests-standardize.R` from the R console using the function `test_file()`

```
# (assuming that your working directory is "sections/03/")
# run from R console
test_file("tests/tests-standardize.R")
```

```
## v |  OK F W S | Context
## / |   0       | tests-standardizev |  11       | tests-standardize
##
## == Results ===============================================================================
## Duration: 0.2 s
##
## OK:       11
## Failed:   0
## Warnings: 0
## Skipped:  0
```

We see that all 11 of the tests were passed, so it seems like our function is working as expected.

To see what the output of `test_file()` looks like when tests fail I included a version of `standarize` which adds a 1 to the end of function called `standarizeWrong` in the `functions.R` file. In this case we expect the tests to fail and that is what we see:

```
# (assuming that your working directory is "sections/03/")
# run from R console
test_file("tests/tests-standardize-wrong.R")
```

```
## v |  OK F W S | Context
## / |   0       | tests-standardize-wrongx |   8 3     | tests-standardize-wrong
## ------------------------------------------------------------------------------------------
## tests-standardize-wrong.R:9: failure: standardize works with normal input
## standardizeWrong(x) not equal to `z`.
## 3/3 mismatches (average diff: 1)
## [1] 0 - -1 == 1
## [2] 1 -  0 == 1
## [3] 2 -  1 == 1
##
## tests-standardize-wrong.R:22: failure: standardize works with missing values
## standardizeWrong(y, na.rm = TRUE) not equal to `z2`.
## 2/3 mismatches (average diff: 1)
## [1] 0.293 - -0.707 == 1
## [2] 1.707 -  0.707 == 1
##
## tests-standardize-wrong.R:32: failure: standardize handles logical vector
## standardizeWrong(w) not equal to `z`.
## 3/3 mismatches (average diff: 1)
## [1]  1.577 -  0.577 == 1
## [2] -0.155 - -1.155 == 1
## [3]  1.577 -  0.577 == 1
## ------------------------------------------------------------------------------------------
##
## == Results ===============================================================================
## OK:       8
## Failed:   3
## Warnings: 0
```

```
## Skipped:  0
```

Here we see that 3 tests failed, namely that our output is not equal to the value that we expect it to be. This allows us to go back to the function and assess what may be going wrong.

Practice problems

Although it is not required to code in this manner, we are going to practice working in a test-driven format. It is a coding practice where you first write the tests, then write a function that will pass those tests, and update the function and tests as needed. Coding in this way is called test-driven development.

Suppose we want to write a function `calculator(x, y, operation)` that takes in two numbers `x` and `y` as well as a string `operation` indicating whether to perform addition, subtraction, multiplication, or division. This function should return a numeric value.

1. Write a test that will check whether `calculator` (which you have not written yet) returns an error when `x` and `y` are not numeric and when `operation` is not in the expected set of operations (i.e. addition, subtraction, multiplication, and division). Save these tests in a file called `tests-calculator.R`. Hint: use the expectation `expect_error()`. You may want to write custom error messages in your assertions.

2. Start writing your `calculator` function to pass the tests written in 1). You the `assertthat` package to produce errors if `x` or `y` are not numeric or when `operation` is not in the expected set of operations. You can choose what you expect the user to call each operation in the function. Save this function as `calculator.R`

3. Use the `test_files("tests-calculator.R")` to see if your function is operating as you hope. Note this assumes you are in the directory that holds `tests-calculator.R`.

4. Write a test that checks whether the addition piece of your calculator produces the correct results with the following input:

    1. `x = 1` and `y = 9`
    2. `x = 100`, `y = -5` Also, check that the value returned is a scalar. Save these tests in a file called `tests-calculator.R`. If you think of any other tests feel free to add them.

5. Add the addition functionality to `calculator()` and call `test_files("tests-calculator.R")` again.

6. Continue iterating through this process for substraction, multiplication, and division. Make sure your function elegantly handles division when the demoninator is 0.

7. If you have time, add new functionality to your calulator (e.g. square root).