Assertions and Testing

Zoe Vernon, Andrew Vaughn, James Duncan

2022-09-09

Table of contents

Overview

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 testing 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.

Purpose of Assertions and Testing

We all want our code to be correct the first time we write it. The unfortunate reality is that we all make mistakes when coding, either because of "silly mistakes" (indexing errors, incorrect syntax, using a wrong variable name, etc.) or because of a fundamental misunderstanding of the problem we are trying to solve. While print statements and writing test cases can help reduce coding errors, it is desirable to have a formal, structured way to test our code to ensure that it is functioning how we want it to. It is here that the `assertthat` and `testthat` packages in R prove useful.

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 within the body of the function would confirm that this is in the case and return an error if not.

On the other hand, tests (sometimes referred to as "unit 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, for example by confirming that an error is thrown when the user calls `add("potato", 2)`.

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.

Here's a summary table comparing the two:

Assertions	Tests
Take the perspective of the developer.	Take the perspective of the user.
Assert that the developer knows what they're doing.	Test what the user can do.
Check internal function states.	Check function results given a specific input.
Typically found within a function alongside source code.	Typically kept in a directory separate from source code.
Can run every time the code is called.	Run periodically during development at specific moments (e.g., before creating a git commit).

Assertions	Tests
Amount of computation should be limited to avoid slowing down source code.	For large packages with many functions, can take many minutes or even hours to run.
Should depend only on local states (e.g., the function's arguments and internal variables).	Can depend on global state (e.g., global options).
Double as inline documentation of source code.	Can be helpful for the code design process (i.e., test-driven development).

Assertions and `assertthat`

An assertion is a statement in a function or program that must be `TRUE` for it to continue and throws an error if `FALSE`. 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 always 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 to the user about their inputs, adding assertions to your function allows you to document exactly what you as the developer expect to happen at intermediate

points. 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):

- Installation from CRAN:

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

- Installation from GitHub (requires the `remotes` package):

```
remotes::install_github("hadley/assertthat")
```

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

These are the three primary functions from the package:

- `assert_that()` signals (i.e., throws) an error. This is primarily what you will use in your functions.
- `see_if()` returns a logical value, with the error message as an attribute, but no error is thrown.
- `validate_that()` returns `TRUE` on success and 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 execution of the function.

```
# these functions will help us see the differences in assertthat's 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"
```

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

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

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

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

When the assertion is `FALSE` the functions have different behaviors. `assert_that()` will throw an error, halting furthering execution of the function immediately. `see_if()` and `validate_that()` will not stop the execution, allowing the function to continue with bad state.

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

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

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

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

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

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

However, all three will give the error message

`assert_that` signals an error:

```
  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 the error message as an attribute:

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

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

```
  attr(see_if_result, "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)."
```

While in general `assert_that()` is likely to be your go-to, you might prefer to use `see_if()` or `validate_that()` in cases where you first want to inspect the error message and perhaps check other aspects of your function's state before eventually signaling an error (e.g., using `stop()` with a custom message) so that function execution does not continue with the bad state.

```
err_msg <- attr(see_if_result, "msg")
stop("see_if() returned FALSE because ", err_msg, call. = FALSE)
```

```
Error: see_if() returned FALSE because c("a", "b") is not a string (a length one character vector).
```

Writing Your Own Assertions

While you could use `see_if()` or `validate_that()` to create custom error messages as in the previous example, `assertthat` already provides a couple of ways to do so.

The first is by adding a new assertion that checks whether the number is odd and add a custom message directly to the assertion:

```
is_odd <- function(x) {
  # your custom assertion checking functions can have their own assertions!
  # you can check multiple conditions by separating them with a ,
  assert_that(is.numeric(x), length(x) == 1)

  # here is the main assertion
  assert_that(x %% 2 == 1, msg = paste("x =", x, "is even"))
}

assert_that(is_odd(2))
```

```
Error: x = 2 is even
```

The second is to using the `on_failure()` function, which allows you to use more complex logic to create your assertion failure messages. Below is an example of how this works:

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

```
$srcref
function(x) {
  assert_that(is.numeric(x), length(x) == 1)
```

```
  x %% 2 == 1
}
```

```
  assert_that(is_odd2(2))
```

```
Error: is_odd2(x = 2) is not TRUE
```

```
  on_failure(is_odd2) <- function(call, env) {
    paste("x =", deparse(call$x), " is even")
  }
  attributes(is_odd2)
```

```
$srcref
function(x) {
  assert_that(is.numeric(x), length(x) == 1)
  x %% 2 == 1
}

$fail
function(call, env) {
  paste("x =", deparse(call$x), " is even")
}
```

```
  assert_that(is_odd2(2))
```

```
Error: x = 2  is even
```

The assertions from our original `is_odd()` function flow through the function we assigned to the `fail` attribute of `is_odd()` by using `on_failure()`, so we still get the appropriate error messages when we pass a non-numeric or vector value to `is_odd()`.

Some Additional Useful Assertions

`assertthat` provides a few additional assertions above and beyond what base R provides that can be quite 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`?

Testing and `testthat`

Assertions allow us to check aspects of function state as the function is being executed, while unit tests help ensure that the output from a function is what we expect given some set of inputs.

A common approach to testing is to use the R console to informally check whether your code works on a few examples. Units tests formalize this approach by providing a framework for testing that allows you to re-run tests as you update your functions. You are likely already using the first approach, so there is no reason to waste your code and energy by not saving it for later use.

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 expectations for function behavior. This serves as a form of internal documentation and helps ensure that your code does what you intend.
2. **Better code structure**: Tests should only check accuracy of small portions of code, allowing you to easily find sources 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, and then write the function that achieves the desired result. This practice is called "test-driven development".
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.

The `testthat` package provides a framework for writing and performing tests in R. There are two pieces of the `testthat` package, forming a hierarchical structure for 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 `testthat` Expectations

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`

Function	Description
`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`

Motivating 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 to check pre-conditions!

```
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, the informal process of testing usually looks something like this, executed line-by-line in the R console:

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

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

Then you might look at the mean and standard deviation of `z` to see if `standardize()` appears to be working as expected:

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

```
[1] 0
```

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

```
[1] 1
```

Next, you might keep testing the 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 so on for different types of inputs:

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

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

This approach is fine and encouraged for interactive development, but don't waste all this energy! Hold on to your testing code for a rainy day.

Using `testthat` expectations

Instead of just writing a list of more or less informal tests in the R console, we are going to use the functions provide by `testthat`.

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

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

The "happy path": Testing with "normal" input

The core of `testthat` consists of *expectations*; to write expectations you use functions from the `testthat` package starting with `expect_` 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'.
```

Edge cases: Testing function robustness

It's important to be creative when testing and get into the mindset of the user of your code. You might be the only user, but your perspective when developing code vs. when you use it later on are not one in the same. Think about the range of inputs the user might give your functions and how your function should behave in cases that don't fall directly on the happy path.

Testing inputs with `NA`

Let's include a vector with missing values, which we want to handle.

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

You may be able to think of other edge cases that would be helpful to test for this function. While it is not practical to attempt to write a test for every possible input, you should think carefully about how your users may reasonably interact with your code and what sorts of inputs may lead to unexpected bugs or shortcomings even when the inputs are reasonable.

Testing for expected errors

While we might assume that `standardize()` will already rightly throw an error when given the character vector `q`, it's still a good idea to test for the failures that we expect. This is especially important for catching bad bugs where an error should be thrown but isn't, allowing function execution to continue with incorrect state. Even if we try to guard against bad state with assertions, testing can still help catch any incorrect logic in those assertions.

Here's the current error we get when calling `standardize()` with character vector input. (Notice the warning signaled by using `mean()` on a character vector.)

```
q <- letters[1:3]
standardize(q)
```

```
Warning in mean.default(x, na.rm = na.rm): argument is not numeric or logical:
returning NA
```

```
Error in x - mean(x, na.rm = na.rm): non-numeric argument to binary operator
```

We can confirm that an error is signaled using `expect_error()`:

```
expect_error(standardize(q))
```

```
Warning in mean.default(x, na.rm = na.rm): argument is not numeric or logical:
returning NA
```

> **!** Always use the regexp argument when using `expect_error()`
>
> Using `expect_error()` without the `regexp` argument, as in the above example, is almost never what you want!

`expect_error()` will pass if *any* error is signaled, including ones that we aren't expecting! Therefore, it's good practice to use the `regexp` argument in `expect_error()` to match the error message you expect to see. In this case, we'll also use `fixed = TRUE` to exactly match the error message, rather than use a more general regular expression.

```
expected_msg <- "non-numeric argument to binary operator"
expect_error(standardize(q), regexp = expected_msg, fixed = TRUE)
```

```
Warning in mean.default(x, na.rm = na.rm): argument is not numeric or logical:
returning NA
```

Together with the `class` and `...` arguments (see `?testthat::expect_error`), it's possible to create very sophisticated logic for matching specific errors.

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.

```
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 passed
```

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

```
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 passed
```

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

```
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')
})
```

```
Test passed
```

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")
```

```
== Testing tests-standardize.R =====================================================

[ FAIL 0 | WARN 0 | SKIP 0 | PASS 0 ]
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 1 ]
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 2 ]
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 3 ]
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 4 ]
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 5 ]
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 6 ]
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 7 ]
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 8 ]
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 9 ]
```

```
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 10 ]
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 11 ] Done!
```

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")
```

```
== Testing tests-standardize-wrong.R ========================================

[ FAIL 0 | WARN 0 | SKIP 0 | PASS 0 ]
[ FAIL 1 | WARN 0 | SKIP 0 | PASS 0 ]
[ FAIL 1 | WARN 0 | SKIP 0 | PASS 1 ]
[ FAIL 1 | WARN 0 | SKIP 0 | PASS 2 ]
[ FAIL 1 | WARN 0 | SKIP 0 | PASS 3 ]
[ FAIL 1 | WARN 0 | SKIP 0 | PASS 4 ]
[ FAIL 2 | WARN 0 | SKIP 0 | PASS 4 ]
[ FAIL 2 | WARN 0 | SKIP 0 | PASS 5 ]
[ FAIL 2 | WARN 0 | SKIP 0 | PASS 6 ]
[ FAIL 3 | WARN 0 | SKIP 0 | PASS 6 ]
[ FAIL 3 | WARN 0 | SKIP 0 | PASS 7 ]
[ FAIL 3 | WARN 0 | SKIP 0 | PASS 8 ]

-- Failure (tests-standardize-wrong.R:9:3): 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

-- Failure (tests-standardize-wrong.R:22:3): 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

-- Failure (tests-standardize-wrong.R:32:3): 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
```

```
[ FAIL 3 | WARN 0 | SKIP 0 | PASS 8 ]
```

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). Use 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).

Acknowledgements

This lab was originally authored by Zoe Vernon and updated incrementally by Andrew Vaughn and James Duncan.