Using tinytest

Mark van der Loo

June 5, 2019

Contents

Reading guide

Readers of this document are expected to know how to write R functions and have a basic understanding of a package source directory structure.

1 Purpose of this package: unit testing

The purpose of *unit testing* is to check whether a function gives the output you expect, when it is provided with certain input. So unit testing is all about comparing *desired* outputs with *realized* outputs. The purpose of this package is to facilitate writing, executing and analyzing unit tests.

2 Expressing tests

Suppose we define a function translating pounds (lbs) to kilograms inaccurately.

```
R>  lbs2kg <- function(x){
      if ( x < 0 ){
        stop(sprintf("Expected nonnegative weight, got %g",x))
      }
      x/2.20
    }
```

We like to check a few things before we trust it.

```
R>  library(tinytest)
R>  expect_equal(lbs2kg(1), 1/2.2046)
```

```
----- FAILED[data]: <-->
 call| expect_equal(lbs2kg(1), 1/2.2046)
 diff| Mean relative difference: 0.002086546
```

```
R>  expect_error(lbs2kg(-3))
```

```
----- PASSED      : <-->
 call| expect_error(lbs2kg(-3))
```

The value of an `expect_*` function is a `logical`, with some attributes that record differences, if there are any. These attributes are used to pretty-print the results.

```
R>  isTRUE( expect_true(2 == 1 + 1) )
```

```
[1] TRUE
```

2.1 Test functions

Currently, the following expectations are implemented.

Function	**what it looks for**
`expect_equal(current, target)`	equality (using `all.equal`)
`expect_equivalent(current, target)`	equality, ignoring attributes
`expect_identical(current, target)`	equality, (using, `identical`)
`expect_true(current)`	does 'current' evaluate to `TRUE`
`expect_false(current)`	does 'current' evaluate to `FALSE`
`expect_error(current, pattern)`	error message matching `pattern`
`expect_warning(current, pattern)`	warning message matching `pattern`

Here, `target` is the intended outcome and `current` is the observed outcome. Also, `pattern` is interpreted as a regular expression.

```
R>  expect_error(lbs2kg(-3), pattern="nonnegative")

----- PASSED      : <-->
 call| expect_error(lbs2kg(-3), pattern = "nonnegative")

R>  expect_error(lbs2kg(-3), pattern="foo")

----- FAILED[xcpt]: <-->
 call| expect_error(lbs2kg(-3), pattern = "foo")
 diff| The error message:
 diff| 'Expected nonnegative weight, got -3'
 diff| does not match pattern 'foo'
```

2.2 Alternative syntax

The syntax of the test functions should be familiar to users of the `testthat` package[1]. In test files only, you can use equivalent functions in the style of `RUnit`[2]. To be precise, for each function of the form `expect_lol` there is a function of the form `checkLol`.

2.3 Interpreting the output and print options

Let's have a look at an example again.

```
R>  expect_false( 1 + 1 == 2 )

----- FAILED[data]: <-->
 call| expect_false(1 + 1 == 2)
 diff| Expected FALSE, got TRUE
```

The output of these functions is pretty self-explanatory, nevertheless we see that the output of these expect-functions consist of

- The result: FAILED or PASSED.
- The type of failure (if any) between square brackets. Current options are as follows.
  - [data] there are differences between observed and expected values.
  - [attr] there are differences between observed and expected attributes, such as column names.
  - [xcpt] an exception (warning, error) was expected but not observed.
- When relevant (see §3), the location of the test file and the relevant line numbers.
- When necessary, a summary of the differences between observed and expected values or attributes.
- The test call.

The result of an expect_ function is a tinytest object. You can print them in long format (default) or in short, one-line format like so.

```
R>  print(expect_equal(1+1, 3), type="short")
```

```
FAILED[data]: <--> expect_equal(1 + 1, 3)
```

print method

Functions that run multiple tests return an object of class tinytests (notice the plural). Since there may be a lot of test results, **tinytest** tries to be smart about printing them. The user has ultimate control over this behaviour. See

```
R>  ?print.tinytests
```

for a full specification of the options.

3 Test files

In **tinytest**, tests are scripts, interspersed with statements that perform checks. An example test file in tinytest can look like this.

```
# contents of test_addOne.R

addOne <- function(x) x + 2

expect_true(addOne(0) > 0)

hihi <- 1
```

```
    expect_equal(addOne(hihi), 2)
```

A particular file can be run using run_test_file

```
R>  run_test_file("test_addOne.R", verbose=FALSE)
```

```
----- FAILED[data]: test_addOne.R<8--8>
 call| expect_equal(addOne(hihi), 2)
 diff| Mean relative difference: 0.3333333

Showing 1 out of 2 test results; 1 tests failed
```

We use verbose=FALSE to avoid cluttering the output in this vignette. By default, verbosity is turned on, and a colorized counter is shown while tests are run. It shows number of tests uncolored, number of failures in red and number of passes in green. If you work with a terminal that does not support ANSI color codes, or if you are uncomfortable reading these colors, use color=FALSE or set options(tt.pr.color=FALSE).

The numbers between <-> indicate at what lines in the file the failing test can be found. By default only failing tests are printed. You can store the output and print all of them.

```
R>  test_results <- run_test_file("test_addOne.R", verbose=FALSE)
R>  print(test_results, passes=TRUE)
```

```
----- PASSED      : test_addOne.R<5--5>
 call| expect_true(addOne(0) > 0)
 ----- FAILED[data]: test_addOne.R<8--8>
 call| expect_equal(addOne(hihi), 2)
 diff| Mean relative difference: 0.3333333
```

Or you can set

```
R>  options(tt.pr.passes=TRUE)
```

to print all results during the active R session.

To run all test files in a certain directory, we can use run_test_dir

```
R>  run_test_dir("/path/to/your/test/directory")
```

By default, this will run all files of which the name starts with test_, but this is customizable.

3.1 Summarizing test results, getting the data

To create some results, run the tests in this package.

```
R>  out <- run_test_dir(system.file("tinytest", package="tinytest")
           , verbose=FALSE)
```

The results can be turned into data using `as.data.frame`. `as.data.frame`

```
R>  head(as.data.frame(out), 3)
```

```
  result                                     call diff short
1   TRUE      expect_true(ignore(checkTrue)(TRUE)) <NA>  <NA>
2   TRUE    expect_true(ignore(checkFalse)(FALSE)) <NA>  <NA>
3   TRUE expect_true(ignore(checkEqual)(1 + 1, 2)) <NA>  <NA>
                file first last
1 test_RUnit_style.R     4    4
2 test_RUnit_style.R     5    5
3 test_RUnit_style.R     6    6
```

The last two columns indicate the line numbers where the test was defined.

A ‘summary‘ of the output gives a table with passes and fails per file. `summary`

```
R>  summary(out)
```

```
tinytests object with 37 results, 37 passing, 0 failing

                    Results
File                 Tests passes fails
  test_RUnit_style.R     5      5     0
  test_env_A.R           2      2     0
  test_env_B.R           6      6     0
  test_file.R            2      2     0
  test_tiny.R           22     22     0
  Total                 37     37     0
```

3.2 Programming over tests, ignoring test results

Test scripts are just R scripts interspersed with tests. The test runners make sure that all test results are caught, unless you tell them not to. For example, since the result of a test is a `logical` you can use them as a condition.

```
R>  if ( expect_equal(1 + 1, 2) ){
        expect_true( 2 > 0)
```

```
    }
```

Here, the second test (`expect_true(2 > 0)`) is only executed if the first test results in `TRUE`. In any case the result of the first test will be caught in the test output, when this is run with `run_test_file` `run_test_dir`, `test_all`, `build_install_test` or through `R CMD check` using `test_package`.

If you want to perform the test, but not record the test result you can do the following (note the placement of the brackets). `ignore`

```
R>  if ( ignore(expect_equal)(1+1, 2) ){
      expect_true(2>0)
    }
```

```
----- PASSED      : <-->
 call| expect_true(2 > 0)
```

Other cases where this may be useful is to perform tests in a loop, e.g. when there is a systematic set of cases to test.

3.3 Running order and side effects

It is a generally a good idea to write test files that are independent from each other. This means that the order of running them is unimportant for the test results and test files can be maintained independently. The function `run_test_file` and by extension `run_test_dir`, `test_all`, and `test_package` encourage this by resetting

- options, set with `options()`;
- environment variables, set with `Sys.setenv()`

after a test file is executed.

To escape this behavior, use `base::Sys.setenv()` respectively `base::options()`. Alternatively use `remove_side_effects=FALSE`.

```
R>  run_test_dir("/path/to/my/testdir"
               , remove_side_effects = FALSE)
R>  test_all("/path/to/my/testdir"
               , remove_side_effects = FALSE)
R>  # Only in tests/tinytest.R:
R>  test_package("PACKAGENAME", remove_side_effects=FALSE)
```

Test files are sorted and run based on the current locale. This means that the order of execution is in general not platform-independent. You can control the sorting behavior interactively or by setting `options(tt.collate)`. To be precise, adding

```
R>  options(tt.collate="C")
```

to /tests/tinytest.R before running test_package will ensure bytewise sorting on most systems. See also help("run_test_dir").

4 Testing packages

Using **tinytest** for your package is pretty easy.

1. Testfiles are placed in /inst/tinytest. The testfiles all have names starting with test (for example test_haha.R).
2. In the file /tests/tinytest.R you place the code

```
if ( requireNamespace("tinytest", quietly=TRUE) ){
  tinytest::test_package("PACKAGENAME")
}
```

3. In your DESCRIPTION file, add **tinytest** to Suggests:.

You can automatically create a minimal running test infrastructure with the setup_tinytest function. setup_tinytest

```
R>  setup_tinytest("/path/to/your/package")
```

In a terminal, you can now do

```
R CMD build /path/to/your/package
R CMD check PACKAGENAME_X.Y.Z.tar.gz
```

and all tests will run.

To run all the tests interactively, make sure that all functions of your new package are loaded. After that, run test_all

```
R>  test_all("/path/to/your/package")
```

where the default package directory is the current working directory.

4.1 Build–install–test interactively

The most realistic way to unit-test your package is to build it, install it and then run all the tests. The function

```
R>    build_install_test("/path/to/your/package")
```

does exactly that. It builds and installs the package in a temporary directory, starts a fresh R session, loads the newly installed package and runs all tests. The return value is a `tinytests` object.

The package is built without manual or vignettes to speed up the whole process.

4.2 Using data stored in files

When your package is tested with `test_package`, **tinytest** ensures that your working directory is the testing directory (by default `tinytest`). This means you can read files that are stored in your folder directly.

Suppose that your package directory structure looks like this (default):

```
/inst
    /tinytest
        /test.R
        /women.csv
```

Then, to check whether the contents of `women.csv` is equal to the built-in `women` dataset, the content of `test.R` looks as follows.

```
R>  dat <- read.csv("women.csv")
R>  expect_equal(dat, women)
```

4.3 Skipping tests on CRAN

There are limits to the amount of time a test can take on CRAN. For longer running code it is desirable to automatically toggle these tests off on CRAN, but to run them during development.

You can not really skip tests at CRAN, because there is no certain way to detect whether a package is tested at one of the CRAN's machines. However, tests that are run with `test_all`, `run_test_dir`, or `run_test_file` can be toggled on and off as follows.

```
R>  if ( at_home() ){
      expect_equal(2, 1+1)
    }
```

If a test file is run using `test_all` or `test_dir` then by default the code following the if-conditions is executed. It will be skipped on CRAN since tests are initiated with `test_package` in that case. It is possible to switch the tests off by `test_all(..., at_home=FALSE)` and similar for `test_dir`.

If you want R CMD check to behave differently at home than from CRAN then the tinytest.R file in pkg/tests should look something like this.

```
if (requireNamespace("tinytest", quietly=TRUE)){
  # Do a local check on the environment
  on_cran <- identical(Sys.getenv("ONCRAN"),"TRUE")
  # now run tinytest
  tinytest::test_package("PACKAGENAME", at_home = !on_cran)
}
```

In this setting, all tests that that depend on at_home() being TRUE will run, unless the environment variable "ONCRAN" is set to "TRUE".

It is up to the package author to decide how to test for this. Some authors use special versioning in their DESCRIPTION file to detect whether R CMD check is run on CRAN or not[1]. Any approach will require some kind of manual intervention.

4.4 Testing your package after installation

Supposing your package is called **hehe** and the **tinytest** infrastructure is used, the following commands run all of **hehe**'s tests.

```
R> library(hehe)
R> library(tinytest)
R> run_test_dir( system.file("tinytest",package="hehe") )
```

This can come in handy when a user of **hehe** reports a bug and you want to make sure all tested functionality works on the user's system.

5 A few tips on packages and unit testing

5.1 Make your package spherical

Larger packages typically consist of functions that are visible to the users (exported functions) and a number of functions that are only used by the exported functions. For example:

```
R> # exported, user-visible function
R> inch2cm <- function(x){
     x*conversion_factor("inch")
```

[1]See this stackoverflow question.

```
    }
R>  # not exported function, package-internal
R>  conversion_factor <- function(unit){
      confac <- c(inch=2.54, pound=1/2.2056)
      confac[unit]
    }
```

We can think of the exported functions as the *surface* of the package and all the other functions as the *volume*. The surface is what a user sees, the volume is what the developer sees. The surface is how a user interacts with a package.

If the surface is small (few functions exported), users are limited in the ways they can interact with your package and that means there is less to test. So as a rule of thumb, it is a good idea to keep the surface small. Since a sphere has the smallest surface-to-volume ratio possible, I refer to this rule as *keep your package spherical*.

By the way, the technical term for the surface of a package is API (application program interface).

5.2 Test the surface, not the volume

Unexpected behavior (a bug) is often discovered when someone who is not the developer starts using code. Bugfixing implies altering code and it may even require you to refactor large chunks of code that is internal to a package. If you defined extensive tests on non-exported functions, this means you need to rewrite the tests as well. As a rule of thumb, it is a good idea to test only the behaviour at the surface, so as a developer you have more freedom to change the internals. This includes rewriting and renaming internal functions completely.

By the way, it is bad practice to change the surface, since that means you are going to break other people's code. Nobody likes to program against an API that changes frequently, and everybody hates to program against an API that changes unexpectedly.

5.3 How many tests do I need?

When you call a function, you can think of its arguments flowing through a certain path from input to output. As an example, let's take a look again at a new, slightly safer unit conversion function.

```
R>  pound2kg <- function(x){
      stopifnot( is.numeric(x) )
      if ( any(x < 0) ){
        warning("Found negative input, converting to positive")
        x <- abs(x)
      }
      x/2.2046
    }
```

If we call `lbs2kg` with argument 2, we can write:

```
    2 -> /2.2046 -> output
```

If we call `lbs2kg` with argument -3 we can write

```
   -3 -> abs() -> /2.2046 -> output
```

Finally, if we call `pound2kg` with `"foo"` we can write

```
   "foo" -> stop() -> Exception
```

So we have three possible paths. In fact, we see that every nonnegative number will follow the first path, every negative number will follow the second path and anything nonnumeric follows the third path. So the following test suite fully tests the behaviour of our function.

```
R>      expect_equal(pound2kg(1), 1/2.2046  )
R>      # test for expected warning, store output
R>      expect_warning( out <- pound2kg(-1) )
R>      # test the output
R>      expect_equal( out, 1/2.2046)
R>      expect_error(pound2kg("foo"))
```

The number of paths of a function is called its *cyclomatic complexity*. For larger functions, with multiple arguments, the number of paths typically grows extremely fast, and it quickly becomes impossible to define a test for each and every one of them. If you want to get an impression of how many tests one of your functions in needs in principle, you can have a look at the **cyclocomp** package of Gábor Csárdi[3].

Since full path coverage is out of range in most cases, developers often strive for something simpler, namely *full code coverage*. This simply means that each line of code is run in at least one test. Full code coverage is no guarantee for bugfree code. Besides code coverage it is therefore a good idea to think about the various ways a user might use your code and include tests for that.

To measure code coverage, I recommend using the **covr** package by Jim Hester[4]. Since **covr** is independent of the tools or packages used for testing, it also works fine with **tinytest**.

5.4 It's not a bug, it's a test!

If users of your code are friendly enough to submit a bug report when they find one, it is a good idea to start by writing a small test that reproduces the error and add that to your test suite. That way, whenever you work on your code, you can be sure to be alarmed when a bug reappears.

Tests that represent earlier bugs are sometimes called *regression tests*. If a bug reappears during development, software engineers sometimes refer to this as a *regression*.

References

[1] Unit Testing for R Hadley Wickham (2016). testthat: Get Started with Testing. The R Journal, vol. 3, no. 1, pp. 5–10, 2011

[2] Matthias Burger, Klaus Juenemann and Thomas Koenig (2018). RUnit: R Unit Test Framework R package version 0.4.32.

[3] cyclocomp: cyclomatic complexity of r code Gábor Csárdi (2016) R package version 1.1.0

[4] covr: Test Coverage for Packages Jim Hester (2018) R package version 3.2.1