Using tinytest

Mark van der Loo

February 21, 2023 | Package version 1.4.1

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| Expected '0.45359702440352', got '0.454545454545455'
```

```
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 expects
`expect_equal(current, target)`	equality (using `all.equal`)
`expect_equivalent(current, target)`	equality, ignoring attributes
`expect_identical(current, target)`	equality, (using, `identical`)
`expect_length(current, length)`	check lenght of object
`expect_true(current)`	`current` evaluates to `TRUE`
`expect_false(current)`	`current` evaluates to `FALSE`
`expect_match(current, pattern)`	All strings in `current` match `pattern`
`expect_inherits(current, class)`	`current` inherits from `class`
`expect_null(current)`	`current` evaluates to `NULL`
`expect_error(current, pattern)`	error message matching `pattern`
`expect_warning(current, pattern)`	warning message matching `pattern`
`expect_message(current, pattern)`	message matching `pattern`
`expect_silent(current)`	expect no warnings or errors (just run)
`expect_stdout(current, pattern)`	expect output to `stdout` matching `pattern`
`expect_equal_to_reference(current, file)`	expect object equal to object stored in RDS file
`expect_equivalent_to_reference(current, file)`	expect object equivalent to object stored in RDS file

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, info="My personal message to the tester" )

----- FAILED[data]: <-->
 call| expect_false(1 + 1 == 2, info = "My personal message to the tester")
 diff| Expected FALSE, got TRUE
 info| My personal message to the tester
```

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

- The result: `FAILED`, `PASSED` or `SIDEFX`. The latter only occurs when side effects are monitored (see §3.4)
- 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 side effects are monitored, and the result is SIDEFX, a side effect was observed. The type of side effect is reported between square brackets.
  - [envv] An environmental variable was created, changed, or deleted.
  - [wdir] The working directory has changed.
  - [file] A file operation occurred in the test directory or one of its subdirectories.
- When relevant (see §3), the location of the test file and the relevant line numbers.
- The test call.
- When relevant, a summary of the differences between observed and expected values or attributes, or a summary of the observed side effect.
- When present, a user-defined information message.

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 ?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=0)
```

```
----- FAILED[data]: test_addOne.R<8--8>
 call| expect_equal(addOne(hihi), 2)
 diff| Expected '2', got '3'

Showing 1 out of 2 results: 1 fails, 1 passes (31ms)
```

We use verbose=0 to avoid cluttering the output in this vignette. By default, verbosity is turned on, and a counter is shown while tests are run. The counter is colorized on terminals supporting ANSI color extensions. If you are uncomfortable reading these colors or prefer colorless output, 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=0)
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| Expected '2', got '3'
```

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=0)
```

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)
File                    Results fails passes
  test_RUnit_style.R          5     0      5
  test_call_wd.R              1     1      0
  test_env_A.R                2     0      2
  test_env_B.R                6     0      6
  test_extensibility.R        1     0      1
  test_file.R                23     0     23
  test_gh_issue_108.R        24     0     24
  test_gh_issue_17.R          2     0      2
  test_gh_issue_32.R          3     0      3
  test_gh_issue_51.R          1     0      1
  test_gh_issue_58.R          2     0      2
  test_gh_issue_86.R          4     0      4
  test_init.R                 1     0      1
```

```
test_tiny.R               79     0     79
test_utils.R               4     0      4
Total                    158     1    157
```

3.2 Programming over tests, ignoring test results, exiting early

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.

It is possible to exit a test file prematurely. For example when there are a number of tests that are not relevant or possible on some OS, you can do the following. `exit_file`

```
R>  if ( Sys.info()[['sysname']] == "Windows"){
      exit_file("Cannot test this on Windows")
    }
```

This will cause `run_test_file` to stop file execution, print the message, and report the information gathered up to where `exit` was called. A function like `test_all` will then continue with the next file, so testing is not aborted completely.

There is also a convenience function called `exit_if_not` that functions similarly to base R's `stopifnot`. It accepts a comma-separated list of expressions, and if any one of them does not result in a single `TRUE` the execution of the test file is stopped with a message. So you can do things like this.

```
R>  exit_if_not(requireNamespace("slartibartfast", quietly=TRUE)
                , packageVersion("slartibartfast") >= "1.0.0")
```

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

```
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")`.

3.4 Monitoring side effects

The term 'side effect' is the technical expression for the situation where a function or expression changes something outside of its scope. Examples include creating, removing, or changing variables in R's global work space, R options, or environment variables of your operating system. We will call such variables or options *external variables*.

To test for side-effects once, use the `side_effects` argument to any of the test runners. For example

```
R>  test_package("pkg", side_effects=TRUE)
```

There is control over which side-effects to track. For example to prevent tracking changes in the working directory, do the following.

```
R>  test_package("pkg", side_effects=list(pwd=FALSE))
```

If you add `report_side_effects()` anywhere in a test file, certain external variables are monitored from that point on, and for that file only. It can be switched off again by calling `report_side_effects(FALSE)` anywhere in the file. The reporting functionality will compare the external state before and after every expression in the test file is run and report any changes.

At the moment, effects that can be monitored include environment variables, locale settings, the present working directory, and file operations in the test directory.

Below is an example of a test file where side effects are recorded. The third line creates an explicit side effect by creating a new environment variable called `hihi` with the value `"lol"`.

```
# contents of test_se.R
report_side_effects()
expect_equal(1+1, 2)
Sys.setenv(hihi="lol")
expect_equal(1+1, 3)
Sys.setenv(hihi="lulz ftw")
```

Running the test file yields an object of class `tinytests` as usual, only now changes in environment variables are reported.

```
R>  run_test_file("test_se.R", verbose=1)
```

```
test_se.R.....................    2 tests 1 fails 2 side-effects  4ms
----- SIDEFX[envv]: test_se.R<3--3>
 call| Sys.setenv(hihi = "lol")
```

```
diff| Added envvar 'hihi' with value 'lol'
----- FAILED[data]: test_se.R<4--4>
call| expect_equal(1 + 1, 3)
diff| Expected '3', got '2'
----- SIDEFX[envv]: test_se.R<5--5>
call| Sys.setenv(hihi = "lulz ftw")
diff| Changed envvar 'hihi' from 'lol' to 'lulz ftw'

Showing 3 out of 4 results: 1 fails, 1 passes, 2 side effects (4ms)
```

Note that as discussed in Section §3.3, **tinytest** will unset the environment variable `hihi` automatically after running the file because it was set directly by the author of the test file using `Sys.setenv`. The real value of the reporting functionality is that it also reports on external variables that are touched by other functions than those you call explicitly in the file.

Reading and comparing versions of external variables takes some time. Especially when it requires a call to the operating service such as a request for values of environment variables. We therefore recommend this to be used only when you suspect a side effect. Or, for example to execute `report_side_effects()` conditional on `at_home()`.

It is not possible to catch all types of side effects, even in principle, using the **tinytest** reporting functionality. Examples include: packages that keep a global variable or environment within their namespace to store some state, and packages that rely on compiled code where there are global objects within the shared object.

Side effects are to be avoided as a general and strong principle, but sometimes there is little or no choice. In Section 7.5 we give some tips on how to properly handle such situations.

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 Testing functions that are not exported: use ':::'

In Section 7.1 it is argued that unit tests should as a rule of thumb focus on the functions that are visible to the user. However, there are cases where it may be preferred to test an internal function. For example when there are two user-visible functions that call the same underlying, unexported function with different arguments. Or when one of the internal functions implements a numerical algorithm that requires thorough testing.

To test functions in your package that are not visible to users that load your package, use the triple-colon operator like so.

```
R>  output = pkg:::some_internal_function(1)
R>  expect_equal(output, 2)
```

This is perfectly ok, and is also accepted by `R CMD check -as-cran`.

Tinytest does not make those internal functions directly callable like some other unit testing packages do. Making internal functions callable means that

1. **tinytest** needs to simulate loading a package, except for the namespace restrictions;
2. there may be significant differences between the environment in which you test the functions, and the environment which a user sees when loading the package;
3. the way the package is loaded during testing may differ from the way it is loaded when a user loads it;
4. exported functions are not clearly distinguished from internal functions in the test code.

accurately simulating how R loads a package is no small matter. It would require a significant epansion of `tinytest`'s code base that would have to be kept synchronised with the way R loads packages (possibly with backport options when R would change in that area).

So in short: let's keep things simple and let R do what it knows how to do.

4.3 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)
```

4.4 Skipping tests on CRAN

It is not possible to detect whether a test is running on CRAN. This means we are forced to detect that we are running tests in our own environment.

In the following example we use the host name to detect if we are running on our own machine and explicitly pass this information to `test_package`.

```
# contents of pkgdir/tests/tinytest.R
if ( requireNamespace("tinytest", quietly=TRUE) ){
   home <- identical( Sys.info()["nodename"], "YOURHOSTNAME" )
   tinytest::test_package("PKGNAME", at_home = home)
}
```

Other ways to detect whether you are running 'at home' include

- Set a custom environment variable (from your OS) and detect it with `Sys.getenv`.
  ```
  home <- identical( Sys.getenv("HONEYIMHOME"), "TRUE" )
  ```
- Use 4-number package versioning while developing and 3-number versioning for CRAN releases[1].
  ```
  home <- length(unclass(packageVersion("PKGNAME"))[[1]]) == 4
  ```

When tests are run interactively

All the interactive test runners have `at_home=TRUE` by default, so while you are developing all tests are run, unless you exclude them explicitly.

```
R> run_test_file("test_hehe.R", verbose=0)
```

All ok, 1 results (1ms)

```
R> run_test_file("test_hehe.R", verbose=0, at_home=FALSE)
```

All ok, 0 results (1ms)

Here is an overview of test runners and their default setting for `at_home`.

Function	Default `at_home`	Intended use
`run_test_file`	`TRUE`	Interactive by developer
`run_test_dir`	`TRUE`	Interactive by developer
`test_all`	`TRUE`	Interactive by developer
`build_install_test`	`TRUE`	Interactive by developer
`test_package`	`FALSE`	`R CMD check`, or after installation by user.

4.5 Testing your package after installation

Supposing your package is called **hehe** and the **tinytest** infrastructure is used. If the package is installed, the following command runs **hehe**'s tests.

```
R> tinytest::test_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.

4.6 Using extension packages

It is possible for other packages to add custom assertions (`expect`-functions). To use such a package:

[1] As recommended here by Dirk Eddelbuettel.

1. Add the extension package to the `Suggests:` field in the `DESCRIPTION` file.
2. Add `using(pkg)` to *each* test file that use the extensions (see `?using`).

using

When multiple extension packages are loaded, and when there are name collisions, the packages loaded later takes precedence over the ones loaded earlier (as usual in R). This includes assertions exported by **tinytest**.

Note. Other than in regular R, it is not possible to disambiguate functions using namespace resolution as in `pkg::expect_something`, because in that case the test result will not be caught by **tinytest**.

The API for building extension packages is described in `?register_tinytest_extension`.

4.7 Mocking databases

The **dittodb** package[5] is capable of mocking pre-recorded database requests. To use extensions like `dittodb`, put the package in `Suggests:` in the `DESCRIPTION` and load it in the test file. The package captures responses from SQL connections and saves them to R files. Therefore, capture the requested response prior to testing by wrapping your request in `start_db_capturing()` and `stop_db_capturing()`. Optionally, specify the file path you want your mocks to be saved. For examples using `dittodb`, see here.

In the testing scripts, load the package and wrap your tests in `with_mock_path(path, with_mock_db())` like

```
R> require(dittodb)
R> with_mock_path(
       system.file("<path-to-your-mocks>", package = "myPackage"),
       with_mock_db({
       # <unit tests which rely on database connections>
       })
   )
```

5 Testing with environmental variables

In **tinytest** you can run tests with custom environment variable settings easily. Just add the `set_env` argument to any of `run_test_file()`, `run_test_dir()` or `test_package()`. Here is an example.

```
R> test_package("tinytest", set_env = list(WA_BABALOOBA="BA_LA_BAMBOO"))
```

With this option, the environment variable will be set during testing and unset afterwards. Setting and unsetting environment variables like this will not be recorded as a side effect. If there is code in the test file that changes this variable, then it is recorded.

Do note that R uses some environment variables that are read during startup, such as `_R_OPTIONS_STRINGS_AS_FACTORS_`. Setting these at runtime has no effect.

6 Running tests in parallel

In **tinytest**, a file should be considered a closed unit: no information created in one test file should be used in another. Under this condition, tests can automatically run in parallel by running different files in different R sessions.

Running code in parallel takes some careful consideration around setting up a cluster, running the tests, and closing the cluster of preparing it for the next run. Depending on the test runner used, there are different levels of control and responsibility for the user to prepare the program for parallelization. Below we describe them from less easier to more control.

build_install_test

This function creates and installs a package in a temporary location. By setting the ncpu parameter, the number of cores used at the testing phase can be increased.

```
R> build_install_test("/path/to/your/package", ncpu=2)
```

We already mentioned that the order in which files are run is in principle system-dependent and it is a good practice not to rely on it. Under parallel situations, all bets on file order are off.

test_package.

This function assumes that a package is installed. It can gather any information necessary to parallelize a test run. The simplest way to parallelize is to specify the number of CPUs used.

```
R> test_package("PACKAGENAME", ncpu=2)
```

Here, test_package will

1. Set up a local cluster using parallel::makeCluster.
2. Load the package on each R instance of the cluster.
3. Run test files in parallel over the cluster.
4. Collect the results and close the cluster.

In stead of just passing the number of CPUs it is possible to pass a cluster object. In that case test_package will still load the package on each node. However, note that if the package gets updated and reinstalled, it should also be reloaded. It is in general hard to completely unload a package in R (see ?detach and ?unloadNamespace for some details on artifacts that will not be removed). So our advice is to restart a cluster for each test run.

run_test_dir, test_all

These function assumes that all functionality needed to run the tests is loaded. They accept an object of type cluster. The user is responsible for setting up the nodes.

```
R> cl <- parallel::makeCluster(4, outfile="")
R> parallel::clusterCall(cl, source, "R/myfunctions.R")
R> run_test_dir("inst/tinytest", cluster=cl)
```

where the argument outfile="" ensures that messages from each node are forwarded to the master node. It is possible to keep the cluster 'alive', so modifications can be made to "R/myfunctions.R" and then run for example the following.

```
R> parallel::clusterCall(cl, source, "R/myfunctions.R")
R> test_all(cluster=cl)
R> stopCluster(cl)
```

For heavy test routines it is thus possible to keep a test cluster up to offload computations.

For more complex situations, including packages that use S4 classes, or compiled code, (re)loading takes more effort than sourcing a few R files. In this cases it is often easier to restart a clean cluster for each test round.

7 A few tips on packages and unit testing

7.1 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")
   }
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).

7.2 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.

7.3 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**.

7.4 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*.

7.5 Side effects are the Devil

Since side-effects manipulate variables outside of the scope of a function, or even outside of R, they can cause bugs that are hard to reproduce. R offers a mechanism to ensure that a function leaves the outside world as it was, once your code stops running.

Suppose you need to change working directory within a function, `source` a file and return to the working directory. A naive way to do this is like so.

`on.exit`

```
R> bad_function <- function(file){
     oldwd <- getwd()
     setwd(dirname(file))
     source(basename(file))
     setwd(oldwd)
   }
```

This all works fine, untill `file` contains faulty code and throws an error. As result, the execution of `bad_function` will stop and leave the user in a changed working directory. With `on.exit` you can define code that will be carried out before the function exits, either normally or with an error.

```
R> good_function <- function(file){
     oldwd <- getwd()
```

```
  on.exit(setwd(oldwd))
  setwd(dirname(file))
  source(basename(file))
}
```

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

[5] dittodb: A Test Environment for Database Requests Jonathan Keane and Mauricio Vargas (2020). R package version 0.1.1