SCTO PLATFORMS

SCTO Validation Platform

Writing function tests

Release date: 2025-01-01

This document is an integral component of the SCTO Validation Platform

---

SCTO PLATFORMS

Document development, review and version history

Development and Review

Authored/revised by:

Name	Date
Alan Haynes,[1] Elio Carreras[2]	2025-01-01

Version History

Version	Date	Author(s)	Summary of Changes
1.0	2025-01-01	Alan Haynes,	Initial version

The `validation` package contains a set of functions to assist in the running and reporting of tests that have been published on the SCTO platform. The tests themselves are located in the validation_tests repository. The results of the tests are then published as issues in the pkg_validation repository.

Tests are written using functions from the testthat package, and can be downloaded, run, and reported using the functions in the `validation` package.

1 High level summary

This section gives a very high level overview of the steps in the process. Refer to the following sections for more in depth information.

1. fork the validation_tests repository to your own GitHub account
2. clone the forked repository to your local machine
3. use `validation::test_skeleton()` to create the necessary files for testing a new package or function
4. write the tests using the `testthat` package
5. test your new tests with `validation::test()`
6. commit your changes to git and push them to your GitHub fork
7. create a pull request to the validation_tests repository
8. address any issues highlighted by the reviewer

2 Core principles

- the unit of testing is the function, not the package
- it is not necessary to test all functions, only those that are used in the product
  - e.g. assuming that the lme4 package is classified as being high risk, it may be that initially only `lmer` is tested.
  - Some time later, `glmer` is used in a product, and so it is necessary to test this function as well.
- tests should be written in a way that they can be run automatically
- tests are written using the `testthat` package
- tests are documented sufficiently to allow others to understand what is being tested and why (e.g. the documentation establishes the user requirements)
  - this principle is not thoroughly applied in this document as it is informative only. See existing examples in the repository.

[1] Senior Statistician, Department of Clinical Research (DCR), University of Bern
[2] Senior Statistical Programmer, SAKK

3 `test_skeleton` helps build the structure

The `test_skeleton` function can be used to create the relevant folder structure for testing a new package, or adding a file for testing additional functions. In the code below, substitute `pkg` with the name of the package to be tested and add the names of the function(s) you want to test in the `funs` argument.

```
test_skeleton("pkg", funs = c("fun", "fun2", "etc"))
```

This will create a set of files in your working directory:

```
-- pkg
   +- info.txt
   +- setup-pkg.R
   +- test-fun.R
   +- test-fun2.R
   +- test-etc.R
```

- `info.txt` file will contain the name of the package and a freetext description of what is tested,
- `setup-pkg.R` is for any necessary setup code (e.g. installation of the package),
- for each function in the `funs` argument, a `test-function.R` file is created, which will contain the actual testing code.

In the event that the package already has tests, the `test_skeleton` function will not overwrite the existing files, only adding any necessary `test-function.R` files.

Add the relevant tests to the `test-function.R` files and check that they work as expected (run `test("package", download = FALSE)`).

4 Writing tests

Testing is performed via the testthat framework. All tests for a given function should be placed in a dedicated `test-function.R` file.

Each test is comprised of one or more expectations and a descriptive name.

E.g.

```
test_that("some meaningful message about the tests", {
  expect_equal(1 + 1, 2)
  expect_true(is.numeric(1))
  expect_false(is.character(1))
})
```

Where multiple tests are to be made on what could be a single object, it is often useful to create the object outside of the `test_that` function. This is particularly useful when different descriptive texts should be shown for the tests (e.g. perhaps the coefficients and standard errors from a model):

```
obj <- some_function(params)
test_that("test 1 on obj", {
  expect_equal(obj$value_to_test, expected_value)
})
test_that("test 2 on obj", {
  expect_equal(obj$another_value_to_test, expected_value)
})
```

If objects are only useful to the test, they can be created within the `test_that` function.

```
test_that("tests on obj", {
  obj <- some_function(params)
  expect_equal(obj$value_to_test, expected_value)
  expect_equal(obj$another_value_to_test, expected_value)
})
```

Making the description of the test meaningful is important, as it will help the user diagnose where the problem is.

`testthat` supports a large number of expectations, which are documented in the testthat documentation. We demonstrate a few examples below.

4.1 Compare computation to a reference value

To test the computation of the function, the following code must be added to the testing file, for as many test cases as considered appropriate:

```
test_that("function f works", {
  expect_equal(f(x), y)
})
```

Where f is the function to be tested, x are the input parameters for the function and y is the expected returned value.

Note that is/may be necessary/desirable to set a tolerance for floating point comparisons. This can be done with the `tolerance` argument.

4.2 Testing for errors, warnings and other messages

To test whether, under certain conditions, the function returns an error, a warning or a message, the following corresponding code can be adapted, for as many test cases as considered appropriate:

```
test_that("function f returns an error", {
  expect_error(f(x))
})

test_that("function f returns a warning", {
  expect_warning(f(x))
})

test_that("function f returns a message", {
  expect_message(f(x))
})
```

Where f is the function to be tested, x are the arguments that define the conditions. Use the `regexp` argument to check for a particular error, warning or message.

```
test_that("function f returns an error", {
  expect_error(f(x), regexp = "some error message")
})
```

In contrast, to test whether the function runs without returning an error, a warning or a message, the following corresponding code can be adapted, for as many test cases as considered appropriate:

```
test_that("function f runs without error", {
  expect_no_error(f(x))
})
```

```
test_that("function f runs without a warning", {
  expect_warning(f(x))
})

test_that("function f runs without a message", {
  expect_message(f(x))
})
```

4.3 Testing booleans

To test whether, under certain conditions, the function returns TRUE or FALSE, adapt the following code as appropriate:

```
test_that("function f returns TRUE", {
  expect_true(f(x))
})

test_that("function f returns FALSE", {
  expect_false(f(x))
})
```

Where f is the function to be tested, x are the arguments that define the conditions.

4.4 Testing for NULL

To test whether, under certain conditions, the function returns NULL, the following code can be adapted, for as many test cases as considered appropriate:

```
test_that("function f returns NULL", {
  expect_null(f(x))
})
```

4.5 Testing the type of object returned (base R)

To test whether, the function returns an object of a certain type, the following code can be adapted, for as many test cases as considered appropriate:

```
test_that("function f returns object of type XXX", {
  expect_type(f(x), type)
})
```

Where f is the function to be tested, x are the arguments that define the conditions and type is any of the following: “integer”, “character”, “factor”, “logical”, “double”.

4.6 Testing the class (s3) of an object

To test whether, the function returns an object of class s3, the following code can be adapted, for as many cases as considered appropriate:

```
test_that("function f returns object of class s3", {
  expect_s3_class(f(x), class)
})
```

Where f is the function to be tested, x are the arguments that define the conditions and class is, among others, any of the following: “data.frame”, “factor”, “Date”, “POSIXct”, etc.

4.7 Running tests under certain conditions

On occasion, it may be desirable to restrict the tests to specific package versions. This can be done by using the `skip_if` functions in `testthat`.

For example, the pivot functions we introduced to tidyr in version 1.0.0. If we have tests on those functions, we can restrict them to versions 1.0.0 and above with the following code:

```
skip_if(packageVersion("tidyr") < "1.0.0")
```

This line can be placed at the top of the test file, before any tests are run. The equivalent can be done for versions below a certain version, which might be useful for deprecated functions.

It might be suitable to stop tests from being run if the internet is not available:

```
skip_if_offline()
```

Or if a package can only be run on a specific operating system:

```
skip_on_os("mac")
```

5 Submitting tests to the platform

Once you have added the necessary tests, add the files to the validation_tests repository. The easiest way is to create a fork of the repository, then navigate to the tests folder, click on “Add file” and then “Upload files”. Now you can simply drag the `pkg` folder into the browser window and commit the change. You can now create a pull request to the original repository to incorporate your code. It is also possible to fork the repository, clone it to your computer and make the commit there, but this is not strictly necessary.

At this stage, the four-eye principle will be applied to the pull request to check the adequacy and quality of the tests and code. If the reviewer agrees with your tests, they will be merged into the package. If they note any issues, which you will see as comments in the GitHub pull request, you will need to address them before the tests can be merged.

Once merged, the tests can be run via the validation package `validation::test("packagename")` and documented in the repository at https://github.com/SwissClinicalTrialOrganisation/pkg_validation.

6 Worked example

Theory is all well and good, but it’s always useful to see how that would be in practice.

Assume that we want to check that the `lm` function from the `stats` package works as expected. We can write a test file that checks that the function returns the expected coefficients and standard errors.

The following assumes that we have cloned the validation_tests repository to our computer and we are within that project.

To begin with, we construct the testing files, specifically the `test_skeleton` function. We only want to test the `lm` function, so we only pass that to the second argument. We also specify the `dir` argument to tell R where to create the new files:

This will have created a `stats` folder within tests. Within that folder, there will be files called `info.txt`, `setup-stats.R`, and `test-lm.R`.

6.1 `test-lm.R`

First we will write the actual tests that we want to run. Tests are entered into the `test-lm.R` file. Opening that file, we see that there are just a few comments at the top of the file, with some reminders.

```
# Write relevent tests for the function in here
# Consider the type of function:
#   - is it deterministic or statistic?
#   - is it worth checking for errors/warnings under particular conditions?
```

We decide that we will use the mtcars dataset as our basis for testing `lm`, so we can load the dataset. We want to test both the linear effect of the number of cylinders on the miles per gallon, and the effect of the number of cylinders (`cyl`), as well as when `cyl` is treated as a factor.

```
# Write relevent tests for the function in here
# Consider the type of function:
#   - is it deterministic or statistic?
#   - is it worth checking for errors/warnings under particular conditions?

data(mtcars)
mtcars$cyl_f <- factor(mtcars$cyl)
```

Note that if there are multiple functions being tested (each in their own test-function.R file) that require the same data, we can load and prepare the data in the `setup-stats.R` file.

We can also define the models that we want to test:

```
cmod <- lm(mpg ~ cyl, data = mtcars)
fmod <- lm(mpg ~ cyl_f, data = mtcars)
```

We do not include the model definitions within a test_that call because we will use the same models in multiple tests. Again, if we needed to use those models for testing multiple functions, we could define them in the setup file.

6.1.1 Testing coefficients

Suppose that we know that the coefficient for `mpg ~ cyl` is known (-2.88 for the linear effect). We can write a test that checks that expectation:

```
test_that("lm returns the expected coefficients", {
  expect_equal(coef(cmod)[2], -2.88)
})
```

Due to floating point precision, this is probably insufficient - R will not return exactly −2.88. We can use the `tolerance` argument to check that the coefficient is within a certain range (we could also round the coefficient). We also need to tell `expect_equal` to ignore the names attribute of the vector, otherwise it compares the whole object, attributes and all:

```
test_that("lm returns the expected coefficients", {
  expect_equal(coef(cmod)[2], -2.88, tolerance = 0.01, check.attributes = FALSE)
})
```

We can do the same for the coefficients from the model with `cyl_f`. This time, we can derive the values from the `tapply` function as, in this case, the coefficients are just the means:

```
test_that("lm returns the expected coefficients", {
  means <- tapply(mtcars$mpg, mtcars$cyl, mean)
```

```
  coefs <- coef(fmod)
  expect_equal(coefs[1], means[1], check.attributes = FALSE)
  expect_equal(coefs[2], means[2] - means[1], check.attributes = FALSE)
  expect_equal(coefs[3], means[3] - means[1], check.attributes = FALSE)
})
```

We have now performed 4 tests (the expectations) in two `test_that` calls. We can also combine them together:

```
test_that("lm returns the expected coefficients", {
  expect_equal(coef(cmod)[2], -2.88, tolerance = 0.01, check.attributes = FALSE)

  means <- tapply(mtcars$mpg, mtcars$cyl, mean)
  coefs <- coef(fmod)
  expect_equal(coefs[1], means[1], check.attributes = FALSE)
  expect_equal(coefs[2], means[2] - means[1], check.attributes = FALSE)
  expect_equal(coefs[3], means[3] - means[1], check.attributes = FALSE)
})
```

Whether to put them in one or two calls is up to the author. Distributing them across more calls helps identify which tests fail, but it also makes the file longer.

6.1.1.1 A note on selecting tolerances

The tolerance is a tricky thing to select. It is a balance between being too strict and too lenient. If the tolerance is too strict, then the test will fail when the function is working as expected. If the tolerance is too lenient, then the test will pass when the function is not working as expected.

Consider the example above. We compared −2.88 with the coefficient which R reports to (at least) 5 decimal places. In this case, it does not make sense to use a tolerance of less than 0.01 because we only know the coefficient to two decimal places (even though we would have access to a far greater precision had we worked for it).

Generally speaking, values that are easy to calculate should probably have a lower tolerance. Values that are very dependent on specifics of the implementation (e.g. maximisation algorithm, etc) should probably have a higher tolerance. This is especially the case when using external software as a reference (e.g. Stata uses different defaults settings to `lme4`, causing differences in SEs). Simulation results, may also require a more lenient tolerance.

6.1.2 Testing the standard errors against Stata

Suppose that we have used Stata as a reference software for the standard errors. We include the commands used in the reference software in comments in the script, including the output and the information of the version of the reference software.

```
# write.csv(mtcars, "mtcars.csv", row.names = FALSE)
# reference software: Stata 17.0 (revision 2024-02-13)
# import delimited "mtcars.csv"
# regress mpg cyl
# [output truncated for brevity]
# ------------------------------------------------------------------------------
#          mpg | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
# -------------+----------------------------------------------------------------
#          cyl |   -2.87579   .3224089    -8.92   0.000    -3.534237   -2.217343
#        _cons |   37.88458   2.073844    18.27   0.000     33.64922    42.11993
# ------------------------------------------------------------------------------
# [output truncated for brevity]
# regress mpg i.cyl
# [output truncated for brevity]
```

We can then use the SE values from Stata in the tests, specifying a suitable tolerance (it's pretty simple to calculate, so we can be quite stringent):

```
test_that("Standard errors from LM are correct", {
  expect_equal(summary(cmod)$coefficients[2, 2], 0.322408,
               tolerance = 0.00001)
  expect_equal(summary(fmod)$coefficients[2, 2], 1.558348,
               tolerance = 0.00001)
  expect_equal(summary(fmod)$coefficients[3, 2], 1.298623,
               tolerance = 0.0001)
})
```

6.1.3 The completed test file

The test file including the tests above is then:

```
# Write relevent tests for the function in here
# Consider the type of function:
#   - is it deterministic or statistic?
#   - is it worth checking for errors/warnings under particular conditions?

data(mtcars)
mtcars$cyl_f <- factor(mtcars$cyl)

cmod <- lm(mpg ~ cyl, data = mtcars)
fmod <- lm(mpg ~ cyl_f, data = mtcars)

test_that("lm returns the expected coefficients", {
  expect_equal(coef(cmod)[2], -2.88, tolerance = 0.01, check.attributes = FALSE)

  means <- tapply(mtcars$mpg, mtcars$cyl, mean)
  coefs <- coef(fmod)
  expect_equal(coefs[1], means[1], check.attributes = FALSE)
  expect_equal(coefs[2], means[2] - means[1], check.attributes = FALSE)
  expect_equal(coefs[3], means[3] - means[1], check.attributes = FALSE)
})

# write.csv(mtcars, "mtcars.csv", row.names = FALSE)
# reference software: Stata 17.0 (revision 2024-02-13)
# import delimited "mtcars.csv"
# regress mpg cyl
#
#       Source |       SS           df       MS      Number of obs   =        32
# -------------+----------------------------------   F(1, 30)        =     79.56
#        Model |  817.712952         1  817.712952   Prob > F        =    0.0000
#     Residual |  308.334235        30  10.2778078   R-squared       =    0.7262
# -------------+----------------------------------   Adj R-squared   =    0.7171
#        Total |  1126.04719        31  36.3241028   Root MSE        =    3.2059
#
# ------------------------------------------------------------------------------
#          mpg | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
# -------------+----------------------------------------------------------------
#          cyl |   -2.87579   .3224089    -8.92   0.000    -3.534237   -2.217343
#        _cons |   37.88458   2.073844    18.27   0.000     33.64922    42.11993
# ------------------------------------------------------------------------------
# regress mpg i.cyl
#
#       Source |       SS           df       MS      Number of obs   =        32
# -------------+----------------------------------   F(2, 29)        =     39.70
#        Model |   824.78459         2  412.392295   Prob > F        =    0.0000
#     Residual |  301.262597        29  10.3883654   R-squared       =    0.7325
# -------------+----------------------------------   Adj R-squared   =    0.7140
```

```
#       Total |  1126.04719        31  36.3241028   Root MSE        =    3.2231
#
# ------------------------------------------------------------------------------
#          mpg | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
# -------------+----------------------------------------------------------------
#          cyl |
#           6  |  -6.920779   1.558348    -4.44   0.000    -10.10796   -3.733599
#           8  |  -11.56364   1.298623    -8.90   0.000    -14.21962   -8.907653
#              |
#        _cons |   26.66364   .9718008    27.44   0.000     24.67608    28.65119
# ------------------------------------------------------------------------------

test_that("Standard errors from lm are correct", {
  expect_equal(summary(cmod)$coefficients[2, 2], 0.322408,
               tolerance = 0.0001)
  expect_equal(summary(fmod)$coefficients[2, 2], 1.558348,
               tolerance = 0.0001)
  expect_equal(summary(fmod)$coefficients[3, 2], 1.298623,
               tolerance = 0.0001)
})
```

For `lm`, other things that might be tested include the R-squared, the F-statistic, and the p-values. Generally speaking, we might also want to test that the model is of the appropriate class (also `lm` in this case), or that the model has the expected number of coefficients, that the function issues warnings and/or errors at appropriate times.

6.2 `info.txt`

It is easiest to write the `info.txt` file once all tests have been written. It provides a listing of what has been tested in prose form and serves as a quick overview of the tests.

The default text the file contains a single line:

```
Tests for package stats
```

Extra details on the tests that we have performed should be added. In this case, we might modify it to:

```
Tests for package stats
- coefficients and SEs from a unvariate model with continuous and factor predictors. SEs were
checked against Stata.
```

Where tests are for/from a specific version of the package, as might be the case for newly added or deprecated functions, this should also be noted.

6.3 `setup-stats.R`

This file should contain the code necessary to load the package and any other packages that are required for the tests. In this case, we need the `stats` package and the `testthat` package.

The testthat package is always needed. In general, loading the package is necessary. As `stats` is a standard R package, it's no necessary in this case.

We also try to leave the environment as we found it, so we detach the packages via the `withr::defer` function. Again, in this case, we don't want to detach it, as it is a standard package.

```
if(!require(stats)) install.packages("stats")
library(stats)
library(testthat)
```

```
withr::defer({
  # most of the time, we would want to detach packages, in this case we don't
  # detach(package:stats)
}, teardown_env())
```

6.4 Testing that the tests work

Assuming the tests are in a folder called `stats`, which is within our current working directory, we can run the tests with:

```
validation::test("stats", download = FALSE)
```

We specify `download = FALSE` because `validation::test` will download the files from GitHub and run those tests by default. `download = FALSE` tells it to use the local copy of the files instead.

The output should return various information on our tests and system. The first part comes from testthat itself while it runs the tests, the remainder (after "Copy and paste the following…") provides summary information that should be copied into a github issue :

```
✔ | F W  S  OK | Context
✔ |          7 | lm

══ Results ═══════════════════════════════════════════════════════════════════
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 7 ]
## Copy and paste the following output into the indicated sections of a new issue

ISSUE NAME:
[Package test]: stats version 4.3.1

### Name of the package you have validated:
stats

### What version of the package have you validated?
4.3.1

### When was this package tested?
2024-03-18

### What was tested?
Tests for package stats
 - coefficients and SEs from a unvariate model with continuous and factor predictors. SEs were
checked against Stata.


### Test results
PASS

### Test output:


|file      |context |test                                  | nb| passed|skipped |error | warning|
|:---------|:-------|:-------------------------------------|--:|------:|:-------|:-----|-------:|
|test-lm.R |lm      |lm returns the expected coefficients |  8|      4|FALSE   |FALSE |       4|
|test-lm.R |lm      |Standard errors from lm are correct  |  3|      3|FALSE   |FALSE |       0|

### SessionInfo:
R version 4.3.1 (2023-06-16 ucrt)
Platform: x86_64-w64-mingw32/x64 (64-bit)
Running under: Windows 10 x64 (build 19045)
```

```
Matrix products: default


locale:
[1] LC_COLLATE=German_Switzerland.utf8  LC_CTYPE=German_Switzerland.utf8
[3] LC_MONETARY=German_Switzerland.utf8 LC_NUMERIC=C
[5] LC_TIME=German_Switzerland.utf8

time zone: Europe/Zurich
tzcode source: internal

attached base packages:
[1] graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] gh_1.4.0         validation_0.1.0 testthat_3.2.0

loaded via a namespace (and not attached):
 [1] xfun_0.40         httr2_0.2.3       htmlwidgets_1.6.2 devtools_2.4.5
 [5] remotes_2.4.2.1   processx_3.8.2    callr_3.7.3       vctrs_0.6.5
 [9] tools_4.3.1       ps_1.7.5          generics_0.1.3    curl_5.1.0
[13] tibble_3.2.1      fansi_1.0.6       pkgconfig_2.0.3   desc_1.4.2
[17] lifecycle_1.0.4   compiler_4.3.1    stringr_1.5.1     brio_1.1.3
[21] httpuv_1.6.12     htmltools_0.5.6.1 usethis_2.2.2     yaml_2.3.8
[25] pkgdown_2.0.7     tidyr_1.3.0       later_1.3.1       pillar_1.9.0
[29] crayon_1.5.2      urlchecker_1.0.1  ellipsis_0.3.2    cranlogs_2.1.1
[33] rsconnect_1.1.1   cachem_1.0.8      sessioninfo_1.2.2 mime_0.12
[37] tidyselect_1.2.0  digest_0.6.33     stringi_1.8.3     dplyr_1.1.4
[41] purrr_1.0.2       rprojroot_2.0.3   fastmap_1.1.1     cli_3.6.2
[45] magrittr_2.0.3    pkgbuild_1.4.2    utf8_1.2.4        withr_3.0.0
[49] waldo_0.5.1       prettyunits_1.2.0 promises_1.2.1    rappdirs_0.3.3
[53] roxygen2_7.3.0    rmarkdown_2.25    httr_1.4.7        gitcreds_0.1.2
[57] stats_4.3.1       memoise_2.0.1     shiny_1.8.0       evaluate_0.22
[61] knitr_1.45        miniUI_0.1.1.1    profvis_0.3.8     rlang_1.1.3
[65] Rcpp_1.0.11       xtable_1.8-4      glue_1.7.0        xml2_1.3.5
[69] pkgload_1.3.3     rstudioapi_0.15.0 jsonlite_1.8.7    R6_2.5.1
[73] fs_1.6.3


### Where is the test code located for these tests?
please enter manually

### Where the test code is located in a git repository, add the git commit SHA
please enter manually, if relevant
```

7 Hints for working with GitHub

RStudio has a built-in git interface, which is a good way to manage your git repositories if you use RStudio.

The Happy Git and GitHub for the useR book is a comprehensive guide to working with git and GitHub. Of particular use are chapters 9 to 12 on connecting your computer with GitHub.

The GitHub desktop app is a good way to manage your git repositories if you are not comfortable with the command line. This is also an easy way to connect your computer with your GitHub account. There are many other GUIs for working with git repositories. See here for a listing of some of them.