Professional R development: being a good boy/girl

Introduction

You want to be good. You want to (learn to) write code that follows all good practices. You are open to being corrected by any professional tool and learn from it. You want to program like the pros. You should read this article.

What you will learn...

You'll learn to add automatic testing for coding standard, code coverage and good practices upon a push to your package its GitHub.

We'll use an example package as a testcase.

In the end, you'll have a script that forces you to work like a pro.

What you should know...

It is assumed you know how to

- read a trivial function with basic R code
- create a package
- use the 'testthat' testing framework most basic functionality
- how to let that package be hosted on GitHub

To be able to read an R function, read any beginner book about R or use the 'swirl' package. The other points are covered in [Hadley, 2015], a book that recommend beginners to read as soon as possible.

About the author

The author enjoys to teach programming following the industry's highest standards. His students, aged 7-77 years, are all confronted with quotes from the literature, especially from 'The Pragmatic Programmer' by Andrew Hunt and David Thomas. Within R, he like to quote all works from Hadley Wickham.

Advantages

You are a novice R programmer. You have a brilliant idea. You follow that avenue. All your packages will benefit!

A year later, you regret that descision. You could have known you should not have done that, would you have read those books and articles earlier.

You can prevent this detour. When in Rome, do like the romans do. Likewise, when programming in R, do like the experts do.

Use in practice

In this article, I will show how to let yourself be helped. First, I will introduce a testcase, that has possible improvements. After setting up a Travis CI and Codecov account, a Travis CI script is pushed to the GitHub. Then I'll discuss the Travis CI build log.

Testcase

You are reading Hadley Wickham's book 'R packages' and are building a package following that structure.

Figure 1. Hadley Wickham

You've written a brilliant function, called 'do_magic' like this (probably stored in a file called *R/do_magic.R*):

```
#' Multiples all values by two,
#'   except 42, which stays 42
#' @param x input, must be numeric
#' @return magicified output
#' @export
do_magic <- function(x)
{
  if (!is.numeric(x)) {
    stop("x must be numeric");
  }
  out = x * 2;
  out = replace(out, out == 84, 42);
  out;
}
```

Listing 1. The do_magic function

You are proud of yourself: you've nicely documentated the function (using the 'Roxygen2' package). Next to this, your function checks its inputs, and fails fast if it cannot process these. You even wrote some tests, using 'testthat' (probably stored in a file calles *tests/testthat/test-do_magic.R*):

```
context("do_magic")

test_that("do_magic: use", {
  expect_equal(do_magic(42), 42)
  expect_equal(do_magic(1), 2)
})
```

Listing 2. The do_magic tests

You assume you've did a great job, as no errors are found when you check the build in RStudio or use *devtools::check()*. You can submit your package to CRAN tonight without any problem (except convincing that the package has relevance)!

That night, before going to sleep, you start to ponder. You wonder if your code can be improved. As a novice R programmer (there are hints in your R code you come from a C or C++ background), you may have no ideas.

Rest assured there will be packages that will aid you in becoming a professional R developer!

Activate Travis CI

Figure 2. The Travis CI logo

First step is to activate Travis CI.

Travis CI is a continuous integration (hence, the 'CI' in the name) service, that is free to use when developing FLOSS software and works smoothly with GitHub.

Continuous integration means that the effect of code, after pushing it to GitHub, is shown automatically after a short amount of time. In other words: if you broke it, you'll notice early. Or, if someone else breaks it, the team will notice early. Also, when someone submits a Pull Request, you will see if it will break the build already before accepting it.

There are many other continuous integration services that work just as well, like Jenkins, Codeship, CircleCI and Wercker. I just happened to learn Travis CI first and I am unaware of the quality of the other continuous integration services.

We need to activate Travis CI first, because only when activated, it will start running upon a GitHub push.

Go to the Travis CI website, *www.travis-ci.org*, and sign in with your GitHub account. Travis requests authorization for some GitHub information, like your name and email. After authorization, you see all GitHubs and their activation status

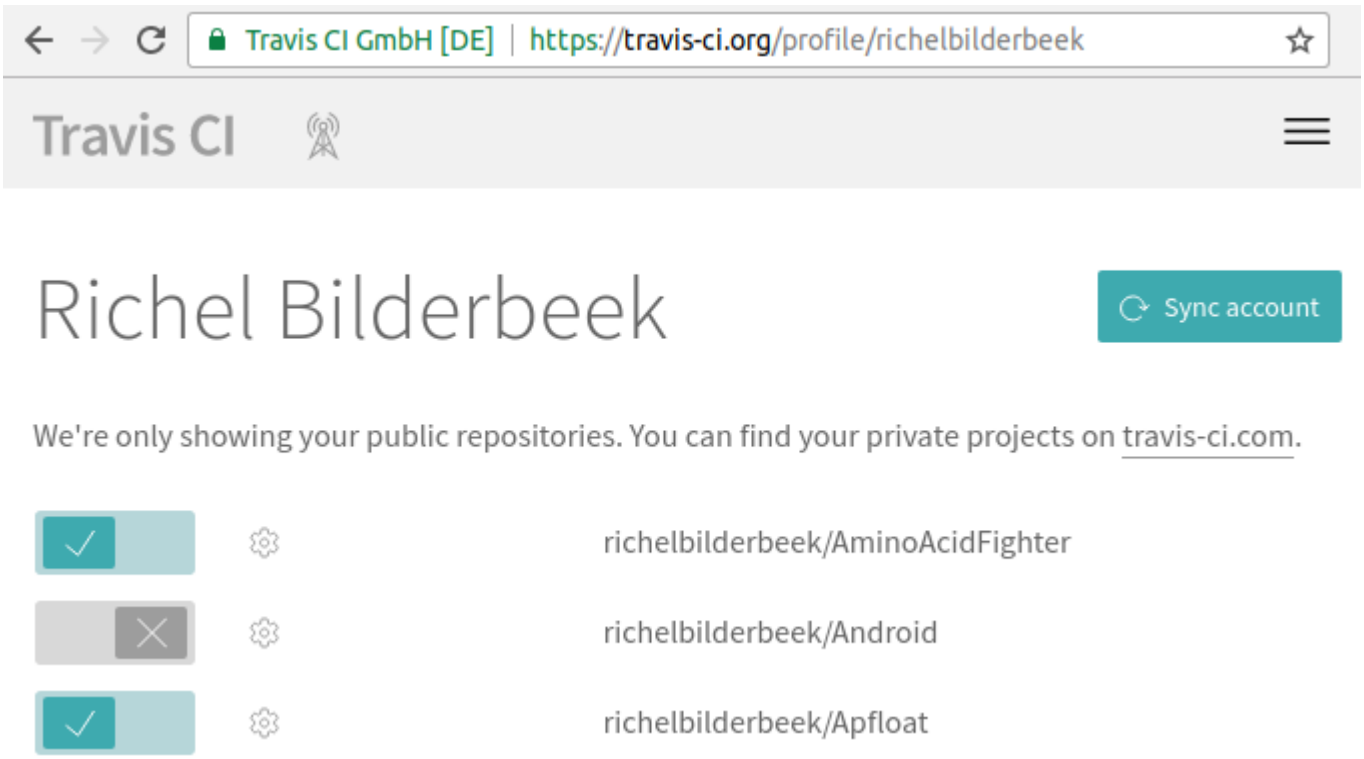

Figure 3. Overview of Travis CI accounts

The sliders indicate the Travis CI activation state. Go find your R package its GitHub and activate it.

Activate Codecov

Figure 4. The Codecov logo

Second step is to activate Codecov.

Codecov is a website that shows your code coverage in a user-friendly form. Codecov tracks a

project its code coverage in time and over multiple branches.

Code coverage is the percentage of lines of code covered by tests. If a line is untested, either you have detected dead code (that can be removed) or you should (be able to) write another test that does use that code. Code coverage correlates with code quality [Del Frate et al., 1995]

There are other services that track code coverage, like Code Climate, Codacity, Coveralls, QuantifiedCode and many more. It just happens to be that the package we'll use ('lintr') uses Codecov, but it could probably just as easy have used to other code coverage service.

We need to activate Codecov now, because only when you have an account, Codecov will receive and display your code coverage.

Go to the Codecov website, *https://codecov.io*, and sign in with your GitHub account. Codecov requests authorization for some GitHub information, like your name and email. After authorization, you see all the GitHubs that have sent their code coverages over.

Add build script

Third step is to create a Travis CI build script.

A Travis CI build script is a file that instructs Travis CI what to do. Basically, you can instruct Travis CI to do anything you can do on within the bash command language. A Travis CI build script is always named *.travis.yml*. The file starts with a dot, which makes it a hidden file on UNIX systems. The '.yml' extension is an abbreviation of 'Yet another Markup Language'.

In your project's root folder, create a file named *.travis.yml*, and put the following text in it:

```
language: r
cache: packages

r_github_packages:
  - jimhester/lintr
  - jimhester/covr
  - MangoTheCat/goodpractice

after_success:
  - Rscript -e "lintr::lint_package()"
  - Rscript -e "covr::codecov()"
  - Rscript -e "goodpractice::gp()"
```

Listing 3. The Travis CI script

You can see that this *.travis.yml* is straightforward. The first line states that the programming language used here is R. The second line tells Travis CI to keep the installed packages in a cache, to prevent needless reinstalls of these packages. The 'r_github_packages' section instructs Travis CI to install these GitHub-hosted packages. The 'after_success' section is run after the package passes a *devtools::check()*. In this section, it will run checks from the 'lintr', 'covr' and 'goodpractice' packages. More on those packages later.

After having creates this *.travis.yml* file, commit and push it to GitHub. I enjoy to name this commit 'Go Travis', but I am open to even better suggestions.

After pushing *.travis.yml* to your GitHub, it will be visible immediatly on GitHub:

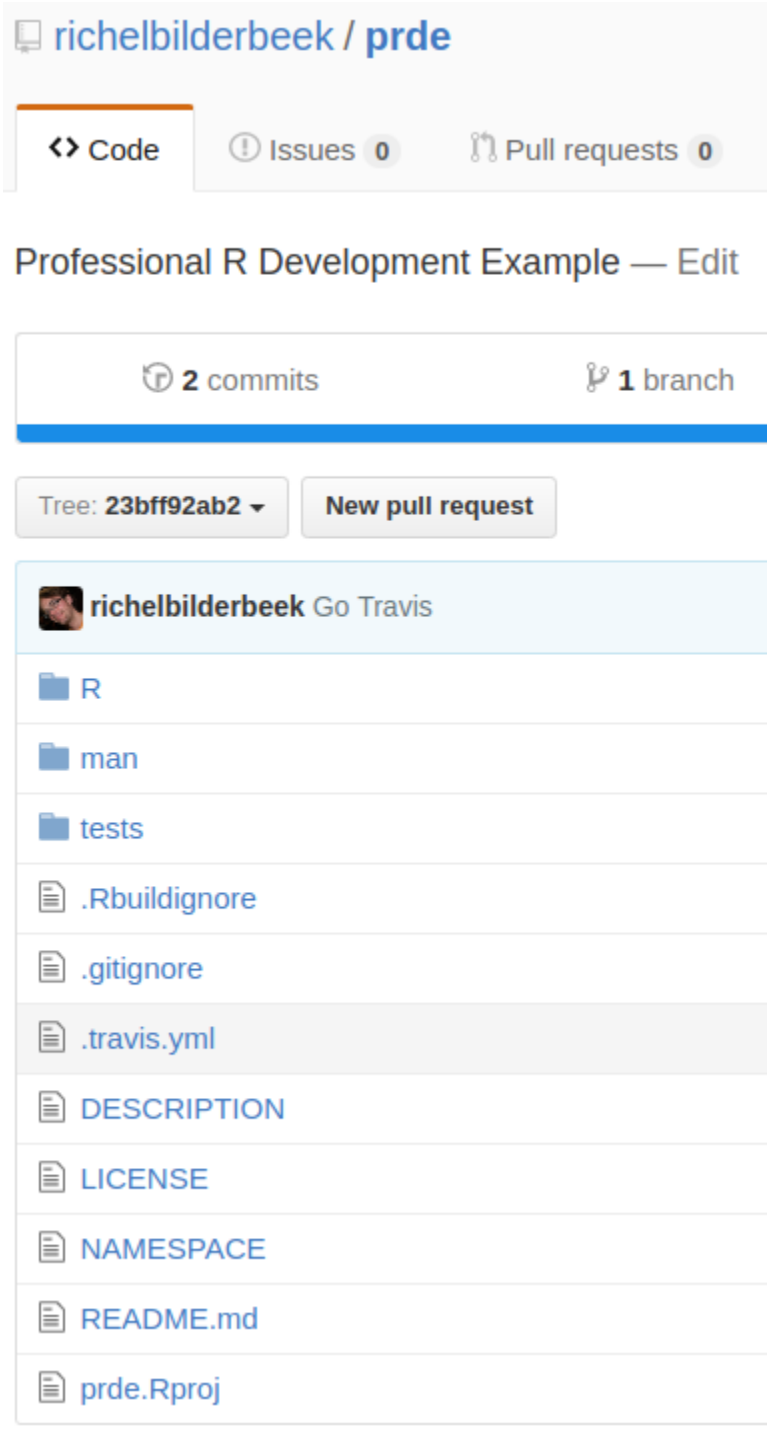

Figure 5. Your GitHub after adding the Travis CI build script

In the back, Travis CI will start doing its labour immediatly.

Read results

Travis CI needs some time to set up a virtual machine. Every time you push to GitHub, a virtual machine is created, so you have a (close to) clean slate that your tests run in. To see Travis CI do its work, go to the Travis CI website, *https://travis-ci.org*. After approximately one minute, you'll see Travis CI do its work. You will see it first installs all packages and their dependencies. The *.travis.yml* script caches all packages, making the second build finish faster.

Here is the header of your first build:

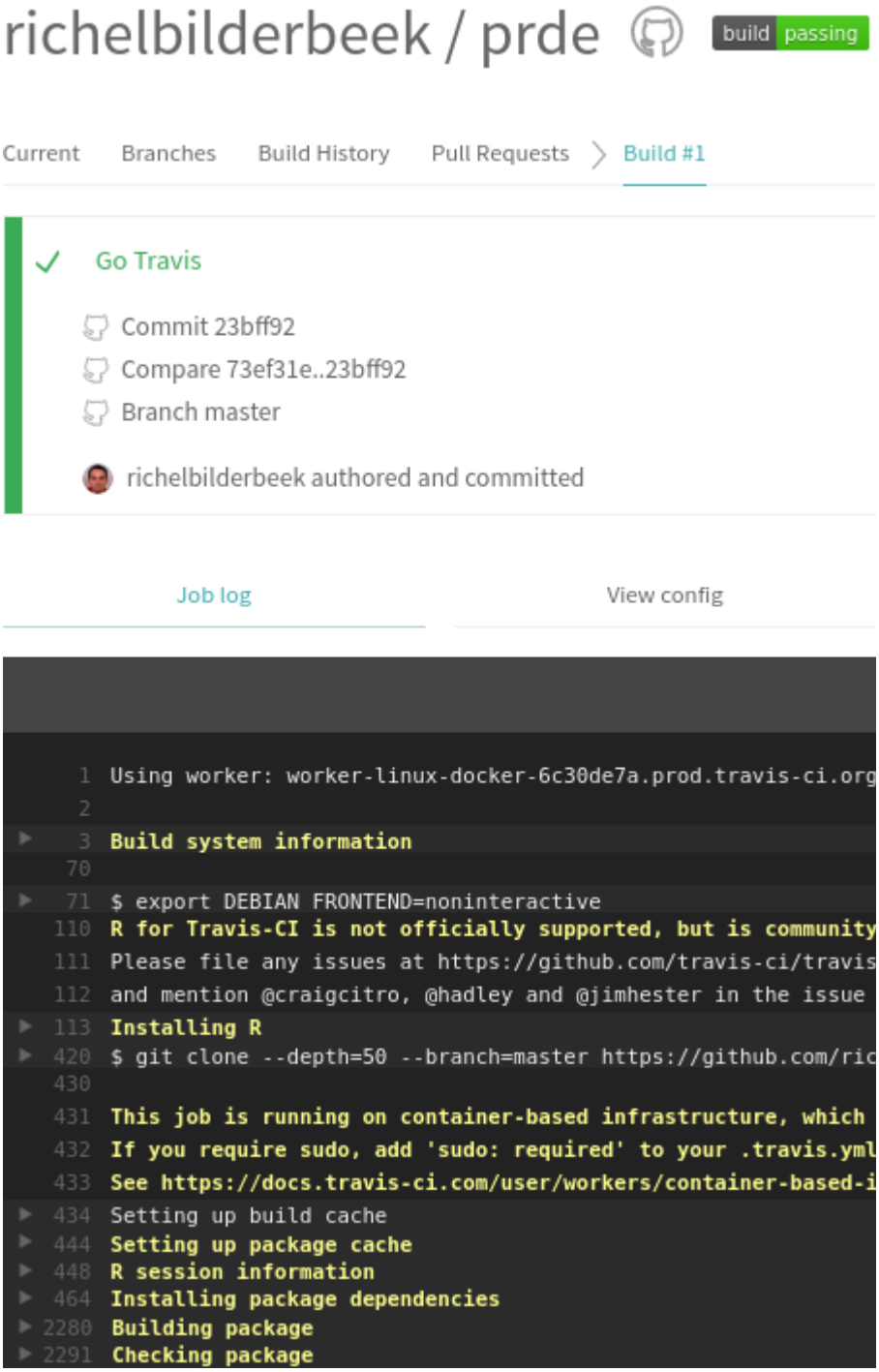

Figure 6. Header of your first build

Craig Citro, Hadley Wickham and Jim Hester are all mentioned for their contributions to make R packages easy to be checked by Travis.

Figure 7. Craig Citro

We already know your build will pass, as you've already checked the build in RStudio or used *devtools::check()*. Would the build not pass, you will see the same output as given by *devtools::check()* and nothing more. If the build passes, there will be some new information is at the bottom:

```
2374 The command "grep -q -R "WARNING" "${RCHECK_DIR}/00check.log"" exited with 0.
2375 store build cache
2381 $ Rscript -e 'lintr::lint_package()'
2392 $ Rscript -e 'library(covr); codecov()'
2415 $ Rscript -e 'library(goodpractice); gp()'
2471
2472 Done. Your build exited with 0.
```

Figure 8. Tail of your first build

Clicking on the triangles on the left reveals some extra information.

First, we'll expand the 'lintr' package (by Jim Hester) its feedback. It shows:

```
2381 $ Rscript -e 'lintr::lint_package()'
2382 R/do_magic.R:6:1: style: Opening curly braces should never go
     new line.
2383 {
2384 ^
2385 R/do_magic.R:10:7: style: Use <-, not =, for assignment.
2386   out = x * 2;
2387       ^
2388 R/do_magic.R:11:7: style: Use <-, not =, for assignment.
2389   out = replace(out, out == 84, 42);
2390       ^
```

Figure 9. The feedback given by the 'lintr' package

'lintr' is a package to check if your coding style follow the one used by, among others, [Wickham, 2014] and [Wickham, 2015]. You'll see that 'lintr' has some suggestions.

Figure 10. Jim Hester

Not only can you see this in your build logs, my good friend lintr-bot will comment on that commit himself, with exactly the same messages:

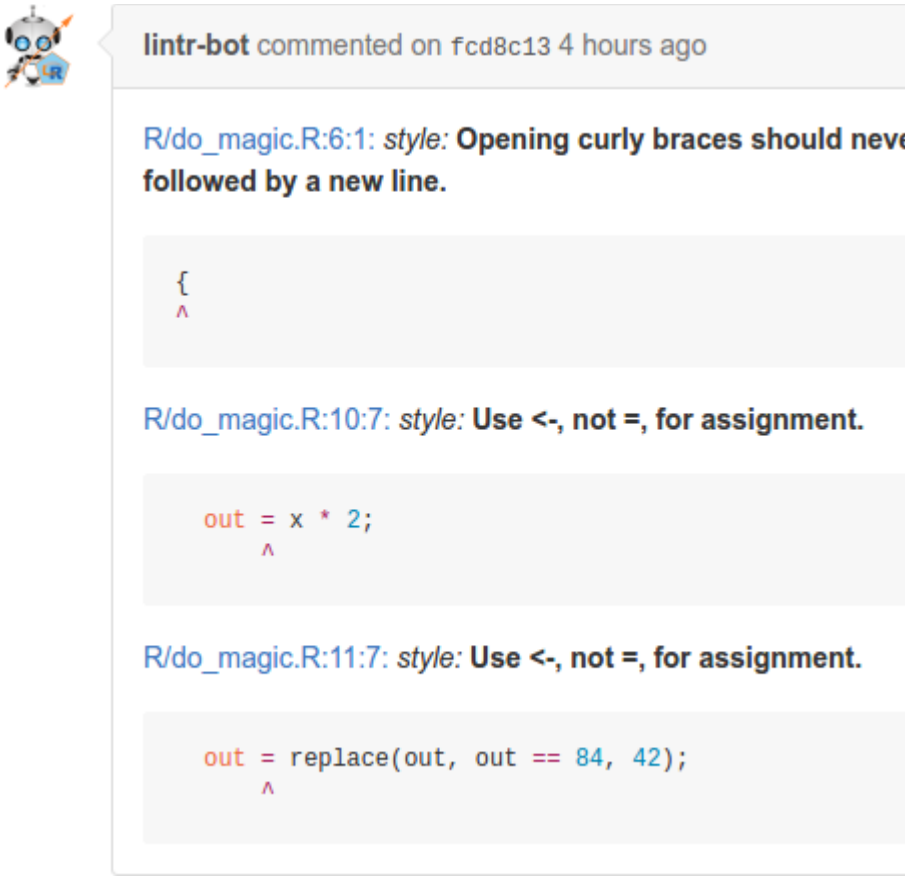

Figure 11. Comments by lintr-bot on your commit

lintr-bot is always right. In case you disagree with it, you can modify the checks done by 'lintr' and allow for other coding standards (but why would one want to go that avenue? Why would the experts have picked those standards, and wouldn't they also know the arguments favoring other standards?).

Figure 12. The MangoTheCat logo

Moving on from lintr-bots words of wisdom, we'll expand the 'goodpractice' package (by MangoTheCat) its feedback. This one shows:

```
── GP prde ──────────────────────────────────────────────────────────────

It is good practice to

  ✖ write unit tests for all functions, and all package code
    in general. 80% of code lines are covered by test cases.

    R/do_magic.R:8:NA

  ✖ add a "URL" field to DESCRIPTION. It helps users find
    information about your package online. If your package does not
    have a homepage, add an URL to GitHub, or the CRAN package package
    page.
  ✖ add a "BugReports" field to DESCRIPTION, and point it to
    a bug tracker. Many online code hosting services provide bug
    trackers for free, https://github.com, https://gitlab.com, etc.
  ✖ use '<-' for assignment instead of '='. '<-' is the
    standard, and R users and developers are used it and it is easier
    to read your code for them if you use '<-'.

    R/do_magic.R:10:7
    R/do_magic.R:11:7

  ✖ omit trailing semicolons from code lines. They are not
    needed and most R coding standards forbid them

    R/do_magic.R:8:30
    R/do_magic.R:10:14
    R/do_magic.R:11:36
    R/do_magic.R:12:6

  ✖ fix this R CMD check NOTE: Malformed Description field:
    should contain one or more complete sentences.
  ✖ fix this R CMD check NOTE: File LICENSE is not mentioned
    in the DESCRIPTION file.
  ✖ fix this R CMD check NOTE: Found the following hidden
    files and directories: .travis.yml These were most likely included
    in error. See section 'Package structure' in the 'Writing R
    Extensions' manual.
```

Figure 13. Feedback given by the 'goodpractice' package

'goodpractice' extends 'lintr' by adding good practices. For example, it may suggest not to use a function, but use a better alternative instead.

There is a third triangle, about the call to the 'covr' package, in the Travis build log that can be extended. Feel free to take a look there, but it won't be too helpful. Instead, go to the Codecov website, https://codecov.io, to get your code coverage displayed in a prettier way:

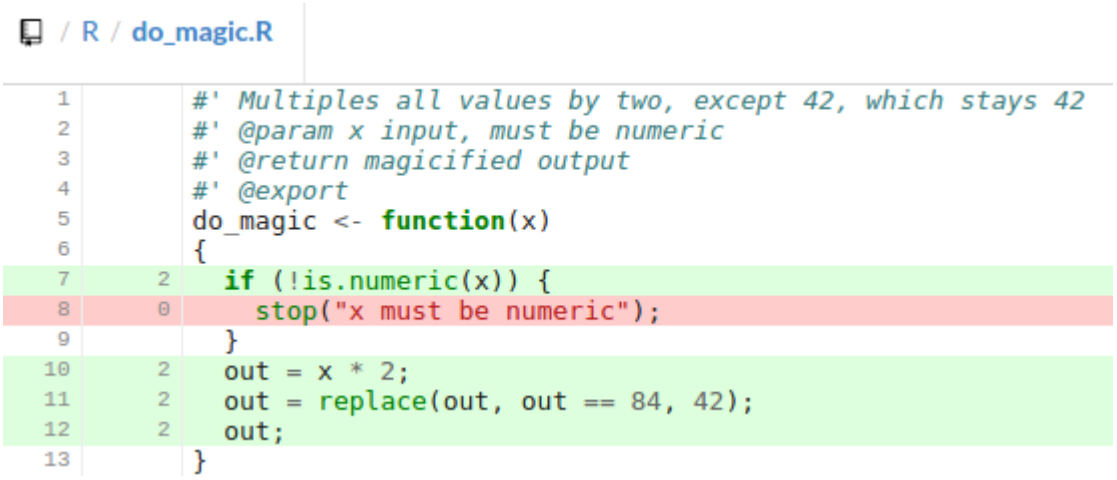

Figure 14. Feedback given by the 'covr' package, as displayed by Codecov

For your code coverage, you can see that you have forgotten to test if your function does throw an exception when the input is not numerical.

You can thank the tools (and people having written those) for helping you become a better R programmer. You may listen to these advices and fix these. Again, you may have your own ideas that you may think are better. But why would the experts recommend against those? And wouldn't they also be aware of you arguments agains those rules?

For my students, I enforce a clean 'oclint' and 'goodpractice' log and a code coverage of at least 95%. It is open to discussion what should be the minimal code coverage limit. Personally, I favor a 100% code coverage. The argumentation for a 10% code coverage is, that if you cannot test for a 100% code coverage, you probably should not keep those rarely triggered if-statements in. For my students, I allow them some leeway and test for things like file corruption and data races. They will have an incomplete code coverage for a nice reason.

Who can use it?

Already from the beginner level, one can use these techniques. For FLOSS development, all tools are free. For closed-source development, there is a fee on using GitHub, Travis CI and Codecov.

What if I use it?

`Code better. Sleep better. [Langr, 2013]`

- As an developer, you can rest assured you've followed all best practices.
- As a potential collaborator, it will be easier to read your code.
- Within a team, there will be no need to write a low-level coding standard
- As a package maintainer, let Pull Requests be checked for these same high standards

What else you can do

When having all tests cleared and high code coverage, you may want to show the world.

This can be done by adding build badges to your *README.md*, in your GitHub's main folder. Such badges look like this:

Figure 15. The badges displayed on your GitHub

To display these badges, add the following code to the *README.md* in your GitHub's main folder:

```
[![Build Status](https://travis-ci.org/[yourname]/[package name].svg?
branch=master)](https://travis-ci.org/[yourname]/[package name])
[![codecov.io](https://codecov.io/github/[yourname]/[package
name]/coverage.svg?branch=master)](https://codecov.io/github/[yourname]/
[package name]?branch=master)
```

I hope it will inspire other people to do the same. I know that it did so for me.

Summary

In this article, you have learned how to let yourself be corrected when deviating from the industry standard.

Go forth and develop like a pro.

On the Web

- https://github.com/richelbilderbeek/sdj_prde: the text and pictures used in this article
- https://github.com/richelbilderbeek/prde: the GitHub developed in this article
- https://github.com/richelbilderbeek/PresentationsAboutR: slides and videos of my presentations about R
- https://travis-ci.org: the Travis CI website
- https://codecov.io: the Codecov website

Glossary

- Code coverage: the percentage of statements executed in your tests
- Continuous integration: integrate development branches continuously, monitoring their effects continously
- git: version control system
- GitHub: online git repository host
- Travis CI: online continuous integration service

References

- [Del Frate et al., 1995] Del Frate, Fabio, et al. "On the correlation between code coverage and software reliability." Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on. IEEE, 1995.

- [Hunt & Thomas, 2000] Hunt, Andrew, and David Thomas. The pragmatic programmer: from journeyman to master. Addison-Wesley Professional, 2000.
- [Langr, 2013] Langr, Jeff. Modern C++ Programming with Test-driven Development: Code Better, Sleep Better. Pragmatic Bookshelf, 2013.
- [Wickham, 2014] Wickham, Hadley. Advanced R. CRC Press, 2014.
- [Wickham, 2015] Wickham, Hadley. R packages. " O'Reilly Media, Inc.", 2015.