# Functional programming and unit testing for data munging with R

*Bruno Rodrigues*

*2016-11-15*

# Contents

# Chapter 1

# Why this book?

This short book serves to show how functional programming and unit testing can be useful for the task of data munging. This book is not an in-depth guide to functional programming, nor unit testing with R.

If you want to have an in-depth understanding of the concepts presented in these books, I can't but recommend Wickham (2014) and Wickham (2015) enough. Here, I will only briefly present functional programming, unit testing and building your own R packages. Just enough to get you (hopefully) interested and going.

This book is not an introduction to R either. I will assume that you have intermediate knowledge of R.

## 1.1 Motivation

Functional programming has very nice features that make working on data sets much more pleasant. It is common that you have to repeat the same instructions over and over again for different data sets that look very similar (for example, same, or similar column names). Of course, it is possible to loop over these data sets and repeat a set of instructions that change these data sets. However, we will see why a functional programming approach is to be preferred.

Unit testing then allows you to make sure that the functions you want to apply to your data sets actually do what you really want them to do. Knowing and applying these two concepts together will make you hopefully a better data analyst.

## 1.2 Who am I?

I use R daily at my current job, and discovered R some years ago while I was at the University of Strasbourg. I'm not an R developer, and don't have a CS background. Most, if not everything, that I know about R is self-taught. I hope however that you will find this book useful. You can follow me on twitter or check my blog.

## 1.3 Thanks

I'd like to thank Ross Ihaka and Robert Gentleman for developing the R programming language. Many thanks to Hadley Wickham for all the wonderful packages he developed that make R much more pleasant to use. Thanks to Yihui Yie for `bookdown` without which this book would not exist (at least not in this very nice format).

Thanks to Hans-Martin von Gaudecker for introducing me to unit testing and writing elegant code. The PEP 8 style guidelines will forever remain etched in my brain.

Finally I have to thank my wife for putting up with my endless rants against people not using functional programming nor testing their code (or worse, using proprietary software!).

## 1.4 License

This book is licensed under the GNU Free Documentation License, version 1.3. A copy of the license is available on the repo, or you can read it online.

# Chapter 2

# Introduction

## 2.1 Getting R

Since I'm assuming you have an intermediate level in R, you already should have R and Rstudio installed on your machine. However, you may lack some of the following packages that are needed to follow the examples in this book:

- covr
- dplyr
- lazyeval
- lubridate
- memoise
- purrr
- readr
- roxygen2
- stringr
- testthat
- tibble
- tidyr

If you're missing some or all of these packages, install them. You'll notice that most, if not all, of these packages were authored or co-authored by Hadley Wickham, currently chief scientist at Rstudio, so you can install most of these packages by installing a single package called `tidyverse`:

```
install.packages("tidyverse")
```

The `tidyverse` package installs some other useful packages that we will not use, but you should check them out anyways!

## 2.2 A short overview of functional programming

What is functional programming? Wikipedia tells us the following:

> In computer science, functional programming is a programming paradigm —a style of building the structure and elements of computer programs— that treats computation as the evaluation

> of mathematical functions and avoids changing state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions or declarations instead of statements. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function f twice with the same value for an argument x will produce the same result f(x) each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

That's the first paragraph of the Wikipedia page and it's quite heavy already!

So let's try to decrypt what is said in this paragraph. Functional programming is a programming paradigm. You may have heard of object oriented programming, or imperative programming before. You actually probably program in an imperative way without knowing it. Imperative programming is usually how programming is taught at universities, and most people then keep on programming in this way. Usually, people that write code in an imperative way tend to write very long scripts that change the state of the program gradually. In the case of a statistician (I will use the word 'statistician' to mean any person that works with datasets. Be it an economist, biologist, data scientist, etc.) this usually means loading a dataset, doing whatever you have to do by writing each step on a file, then running everything. Sometimes you have to save temporary datasets, and then write other scripts that do a series of computations on these temporary datasets and then not forget to delete said temporary datasets. Functional programming is different, in that you write functions that do one single task and then call these functions successively on your data set. These functions can be used for any other project, can be documented and tested. It is also easier to automate tasks and navigate through the code. Since one function does one single task, if you're looking for the line of code that creates variable `X`, just look in the function called `create_var_X()`, instead of scrolling aimlessly through a 1000 lines long script. You can also be sure that your functions do not do anything else (basically, this is what is meant by "eliminating side effects") than the single task you gave them. You can *trust your functions*.

## 2.3 A short overview of unit testing

At the end of the last section I wrote that you can *trust your functions*. Is that true though? Functional programming can make your life easier, but it does not prevent you from introducing bugs in your code. However, what functional programming makes easily possible, is to very easily and effectively test your code thanks to unit testing. You probably already test your code, by hand. You write some loop that is supposed to sum the first 10 integers and then you try it out and check if, indeed, your loop returns 55. Because this is the correct result, you save your work and continue programming something else, and so on. Unit testing is this, but in an automated way. Instead of just trying things out in the interpreter, you write unit tests. You write code that actually checks your functions. You save this unit tests somewhere, and then re-run them whenever you make changes to your code. Even if you don't change some parts of your code, you re-run every unit test. Because you actually never know what may happen. Maybe changing a single line in one of your functions introduced some unforeseen consequences that breaks functionality some place else. When you change code, and *all* your unit tests still pass, then you can be confident that your code is correct (actually, don't be too confident, because maybe you didn't write enough unit tests to cover every case. But we will see how we can be sure there is enough *coverage*).

# Chapter 3

# Functional Programming

## 3.1 Introduction

### 3.1.1 Function definitions

As mentioned in the functional programming overview functional programming is one of the numerous ways to write code. In functional programming, you write functions that do the computations and then as the user, you call these functions to work for you.

You should be familiar with function definitions in R. For example, suppose you want to compute the square root of a number and want to do so using Newton's algorithm:

```
sqrt_newton <- function(a, init, eps = 0.01){
    while(abs(init**2 - a) > eps){
        init <- 1/2 *(init + a/init)
    }
    return(init)
}
```

You can then use this function to get the square root of a number:

```
sqrt_newton(16, 2)
```

```
## [1] 4.00122
```

We are using a `while` loop inside the body[1] of the function. In *pure* functional programming languages, like Haskell, you don't have loops. How can you program without loops, you may ask? In functional programming, loops are replaced by recursion. Let's rewrite our little example above with recursion:

```
sqrt_newton_recur <- function(a, init, eps = 0.01){
    if(abs(init**2 - a) < eps){
        result <- init
    } else {
        init <- 1/2 * (init + a/init)
        result <- sqrt_newton_recur(a, init, eps)
```

[1] The *body* of a function are the instructions that define the function. You can get the body of a function with `body(some_func)`

```
    }
    return(result)
}
```

```
sqrt_newton_recur(16, 2)
```

```
## [1] 4.00122
```

R is not a pure functional programming language though, so we can still use loops (be it `while` or `for` loops) in the bodies of our functions. Actually, for R specifically, it is better, performance-wise, to use loops instead of recursion, because R is not tail-call optimized. I won't got into the details of what tail-call optimization is but just remember that if performance is important a loop will be faster. However, sometimes, it is easier to write a function using recursion. I personally tend to avoid loops if performance is not important, because I find that code that avoids loops is easier to read and debug. However, knowing that you have can use loops is reassuring. In the coming sections I will show you some built-in function that make it possible to avoid writing loops and that don't rely on recursion, so performance won't be penalized.

### 3.1.2 Properties of functions

Mathematical functions have a nice property: we always get the same output for a given input. This is called referential transparency and we should aim to write our R functions in such a way.

For example, the following function:

```
increment <- function(x){
    return(x + 1)
}
```

Is a referential transparent function. We always get the same result for any `x` that we give to this function.

This:

```
increment(10)
```

```
## [1] 11
```

will always produce `11`.

However, this one:

```
increment_opaque <- function(x){
    return(x + spam)
}
```

is not a referential transparent function, because its value depends on the global variable `spam`.

```
spam <- 1

increment_opaque(10)
```

```
## [1] 11
```

will only produce `11` if `spam = 1`. But what if `spam = 19`?

```
spam <- 19

increment_opaque(10)
```

```
## [1] 29
```

To make `increment_opaque()` a referential transparent function, it is enough to make `spam` an argument:

```
increment_not_opaque <- function(x, spam){
    return(x + spam)
}
```

Now even if there is a global variable called `spam`, this will not influence our function:

```
spam <- 19

increment_not_opaque(10, 34)
```

```
## [1] 44
```

This is because the variable `spam` defined in the body of the function is a local variable. It could have been called anything else, really. Avoiding opaque functions makes our life easier.

Another property that adepts of functional programming value is that functions should have no, or very limited, side-effects. This means that functions should not change the state of your program.

For example this function (which is not a referential transparent function):

```
count_iter <- 0

sqrt_newton_side_effect <- function(a, init, eps = 0.01){
    while(abs(init**2 - a) > eps){
        init <- 1/2 *(init + a/init)
        count_iter <<- count_iter + 1 # The "<<-" symbol means that we assign the
    }                                 # RHS value in a variable in the global environment
    return(init)
}
```

If you look in the environment pane, you will see that `count_iter` equals 0. Now call this function with the following arguments:

```
sqrt_newton_side_effect(16000, 2)
```

```
## [1] 126.4911
```

```
print(count_iter)
```

```
## [1] 9
```

If you check the value of `count_iter` now, you will see that it increased! This is a side effect, because the function changed something outside its scope. It changed a value in the global environment. In general, it is good practice to avoid side-effects. For example, we could make the above function not have any side effects like this:

```
sqrt_newton_count <- function(a, init, count_iter = 0, eps = 0.01){
    while(abs(init**2 - a) > eps){
        init <- 1/2 *(init + a/init)
        count_iter <- count_iter + 1
    }
    return(c(init, count_iter))
}
```

Now, this function returns a list with two elements, the result, and the number of iterations it took to get the result:

```
sqrt_newton_count(16000, 2)
```

```
## [1] 126.4911   9.0000
```

Writing to disk is also considered a side effect, because the function changes something (a file) outside its scope. But this cannot be avoided (and it's actually a good thing to have, functions that can write to disk) so just remember: try to avoid having functions changing variables in the global environment unless you have a very good reason of doing so.

Finally, another property of mathematical functions, is that they do one single thing. Functional programming purists also program their functions to do one single task. This has benefits, but can complicate things. The function we wrote previously does two things: it computes the square root of a number and also returns the number of iterations it took to compute the result. However, this is not a bad thing; the function is doing two tasks, but these tasks are related to each other and it makes sense to have them together. My piece of advice: avoid having functions that do too many *unrelated* things. This makes debugging harder.

In conclusion: you should strive for referential transparency, try to avoid side effects unless you have a good reason to have them and try to keep your functions short and do as little tasks as possible. This makes testing and debugging easier, as you will see.

## 3.2 Mapping and Reducing: the *base* way

No introduction to functional programming would be complete without some discussion about the functions `Map()` (and the associated `*apply()` family of functions) and `Reduce()`. `Map()` allows you to map your function to every element of a list of arguments and is easy to understand, while `Reduce()` (sometimes called `fold()` in other programming languages) *reduces* a list of values to a single value by successively applying a function. It's a bit harder to understand, but with some examples it will become clear soon enough. In this section we will focus on how to do things using `base` functions. In the next section we will take a look at the `purrr` package which extends R's functional programming capabilities tremendously.

### 3.2.1 Mapping with `Map()` and the `*apply()` family of functions

Now that we have our nice function that computes square roots using Newton's algorithm, we would like to compute the square root of every element in the following list:

```
numbers <- c(16, 25, 36, 49, 64, 81)

sqrt_newton(numbers, init = rep(1, 6), eps = rep(0.001, 6))
```

```
## Warning in while (abs(init^2 - a) > eps) {: the condition has length > 1
## and only the first element will be used

## Warning in while (abs(init^2 - a) > eps) {: the condition has length > 1
## and only the first element will be used

## Warning in while (abs(init^2 - a) > eps) {: the condition has length > 1
## and only the first element will be used

## Warning in while (abs(init^2 - a) > eps) {: the condition has length > 1
## and only the first element will be used

## Warning in while (abs(init^2 - a) > eps) {: the condition has length > 1
## and only the first element will be used

## Warning in while (abs(init^2 - a) > eps) {: the condition has length > 1
## and only the first element will be used
```

```
## [1] 4.000001 5.000023 6.000253 7.001406 8.005148 9.014272
```

We get a whole bunch of nasty warning messages, but we do get the expected result. But you should not leave it like this. Who knows what may happen some time down the road, when you try to compose this function with another? Maybe you'll get an error and you won't understand why! Let's rewrite the function properly.

We get these warnings because the condition `(init^2 - a) > eps` does not make sense for vectors. Here, R tells the user that it only uses the first element and then does the computation anyways. I would prefer if R would stop the execution and print an error message. This would force the user to have to rewrite the function to explicitly take vectors into account. And there is a very simple way of doing it, by using the function `Map()`:

```
Map(sqrt_newton, numbers, init = 1)
```

```
## [[1]]
## [1] 4.000001
##
## [[2]]
## [1] 5.000023
##
## [[3]]
## [1] 6.000253
##
## [[4]]
## [1] 7
##
## [[5]]
## [1] 8.000002
##
## [[6]]
## [1] 9.000011
```

`Map()` applies a function to every element of a list and returns a list.

We could then write a wrapper around `Map()`:

```
sqrt_newton_vec <- function(numbers, init, eps = 0.01){
    return(Map(sqrt_newton, numbers, init, eps))
}

sqrt_newton_vec(numbers, 1)
```

```
## [[1]]
## [1] 4.000001
##
## [[2]]
## [1] 5.000023
##
## [[3]]
## [1] 6.000253
##
## [[4]]
## [1] 7
##
## [[5]]
## [1] 8.000002
##
## [[6]]
## [1] 9.000011
```

As you can see, we can give a function as an argument to another function. This makes `Map()` a *higher-order function*. Higher-order functions are functions that take other functions as arguments and return either another function, or a value. This is another important concept in functional programming and encourages modularity. It makes your code easily reusable!

R has other higher-order functions that work like `Map()`, such as `apply()`, `lapply()`, `mapply()`, `sapply()`, `vapply()` and `tapply()`. Depending on what you want to do, you will have to use one or the other. `apply()` and 'tapply()' are different from the other `*apply()` functions, because they work on arrays. You can apply a function on the rows or columns of an array, for example if you want a row-wise sum:

```
a <- cbind(c(1, 2, 3), c(4, 5, 6), c(7, 8, 9))
apply(a, 1, sum)
```

```
## [1] 12 15 18
```

We could use `lapply()` instead of `Map()`:

```
lapply(numbers, sqrt_newton, init = 1)
```

```
## [[1]]
## [1] 4.000001
##
## [[2]]
## [1] 5.000023
##
## [[3]]
## [1] 6.000253
##
```

```
## [[4]]
## [1] 7
##
## [[5]]
## [1] 8.000002
##
## [[6]]
## [1] 9.000011
```

or `sapply()`:

```r
sapply(numbers, sqrt_newton, init = 1)
```

```
## [1] 4.000001 5.000023 6.000253 7.000000 8.000002 9.000011
```

We could rewrite `sqrt_newton_vec()` with `sapply()` which would return a better looking result (a list of numbers instead of a list of lists):

```r
sqrt_newton_vec <- function(numbers, init, eps = 0.01){
    return(sapply(numbers, sqrt_newton, init, eps))
}

sqrt_newton_vec(numbers, 1)
```

```
## [1] 4.000001 5.000023 6.000253 7.000000 8.000002 9.000011
```

`mapply()` is different from these two:

```r
inits <- c(100, 20, 3212, 487, 5, 9888)
mapply(sqrt_newton, numbers, init = inits)
```

```
## [1] 4.000284 5.000001 6.000003 7.000006 8.000129 9.000006
```

What happens here is that `sqrt_newton()` gets called with following arguments:

```r
sqrt_newton(numbers[1], inits[1])
```

```
## [1] 4.000284
```

```r
sqrt_newton(numbers[2], inits[2])
```

```
## [1] 5.000001
```

```r
sqrt_newton(numbers[3], inits[3])
```

```
## [1] 6.000003
```

```
sqrt_newton(numbers[4], inits[4])
```

```
## [1] 7.000006
```

```
sqrt_newton(numbers[5], inits[5])
```

```
## [1] 8.000129
```

```
sqrt_newton(numbers[6], inits[6])
```

```
## [1] 9.000006
```

From the `Map()`'s documentation, we learn that:

```
`Map()` is wrapper to `mapply()` which does not attempt to simplify the result...
```

All this behaviour can be replicated using loops, but once you get the gist of these functions, you can write code that is shorter and easier to read and unlike in the case of recursion, without any loss in performance (but without any gains either).

### 3.2.2 `Reduce()`

`Reduce()` is another very useful higher-order function, especially if you want to avoid loops to make your code easier to read. In some programming languages, `Reduce()` is called `fold()`.

I think that the following example illustrates the power of `Reduce()` well:

```
Reduce(`+`, numbers, init = 0)
```

```
## [1] 271
```

Can you guess what happens? `Reduce()` takes a function as an argument, here the function `+`[2] and then does the following computation:

```
0 + numbers[1] + numbers[2] + numbers[3]...
```

It applies the user supplied function successively but has to start with something, so we give it the argument `init` also. This argument is actually optional, but I show it here because in some cases it might be useful to start the computations at another value than `0`. This function generalizes functions that only take two arguments. If you were to write a function that returns the minimum between two numbers:

```
my_min <- function(a, b){
    if(a < b){
        return(a)
    } else {
        return(b)
    }
}
```

[2]This is simply the `+` operator you're used to. Try this out: `` `+`(1, 5) `` and you'll see `+` is a function like any other. You just have to write backticks around the plus symbol to make it work.

You could use `Reduce()` to get the minimum of a list of numbers:

```
print(numbers)
```

```
## [1] 16 25 36 49 64 81
```

```
Reduce(my_min, numbers)
```

```
## [1] 16
```

Here we don't supply an `init` because there is no need for it. Of course R's built-in `min()` function works on a list of values. But `Reduce()` is a very powerful function that can make our life much easier and most importantly avoid writing clumsy loops.

This is the end of the introduction to functional programming. Entire books have been written on the subject, such as the upcoming book by Khan (2017) or Lipovaca (2011). If you're curious about functional programming, you should read these books. For our purposes though, knowing how to write functions, and trying to make them referentially transparent as well as knowing about `Map()` and `Reduce()` is enough to get us going.

## 3.3 Mapping and Reducing: the purrr way

Hadley Wickham developed a package called `purrr` which contains a lot of very useful functions. I will show some of them, but will only scratch the surface. Take the time to read `purrr`'s documentation! You can read more about `purrr` in Wickham and Grolemund (2016).

### 3.3.1 The `map*()` family of functions

In the previous section we saw how to map a function to each element of a list. Each version of an `*apply()` function has a different purpose, but it is not very easy to remember which one returns a list, which other one returns an atomic vector and so on. If you're working on data frames you can use `apply()` to sum (for example) over columns or rows, because you can specify which `MARGIN` you want to sum over. But you do not get a data frame back. In the `purrr` package, each of the functions that do mapping have a similar name. The first part of these functions' names all start with `map_` and the second part tells you what this function is going to output. For example, if you want `doubles` out, you would use `map_dbl()`. If you are working on data frames want a data frame back, you would use `map_df()`. These are much more intuitive and easier to remember. There are also other interesting variants, such as `map_if()`:

```
library("purrr")
a <- seq(1,10)

is_multiple_of_two <- function(x){
    ifelse(x %% 2 == 0, TRUE, FALSE)
}

map_if(a, is_multiple_of_two, sqrt)
```

```
## [[1]]
## [1] 1
##
```

```
## [[2]]
## [1] 1.414214
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] 2
##
## [[5]]
## [1] 5
##
## [[6]]
## [1] 2.44949
##
## [[7]]
## [1] 7
##
## [[8]]
## [1] 2.828427
##
## [[9]]
## [1] 9
##
## [[10]]
## [1] 3.162278
```

What happened in this snippet of code? First I wrote a function that returns `TRUE` if a number is a multiple of 2, and `FALSE` otherwise. Then, I used `map_if()` to take the square root of only those numbers in vector `a` that are divisble by 2.

`map2()` is the equivalent of `mapply()` and `pmap()` is the generalisation of `map2()` for more than 2 arguments:

```
map2(numbers, inits, sqrt_newton)
```

```
## [[1]]
## [1] 4.000284
##
## [[2]]
## [1] 5.000001
##
## [[3]]
## [1] 6.000003
##
## [[4]]
## [1] 7.000006
##
## [[5]]
## [1] 8.000129
##
## [[6]]
## [1] 9.000006
```

### 3.3.2 Reducing with purrr

In the `purrr` package, you can find two more functions for folding: `reduce()` and `reduce_right()`. The difference between `reduce()` and `reduce_right()` is pretty obvious: `reduce_right()` starts from the right!

```r
a <- seq(1, 10)

reduce(a, `-`)
```

```
## [1] -53
```

```r
reduce_right(a, `-`)
```

```
## [1] -35
```

For operations that are not commutative, this makes a difference. Other interesting folding functions are `accumulate()` and `accumulate_right()`:

```r
a <- seq(1, 10)

accumulate(a, `-`)
```

```
##  [1]   1  -1  -4  -8 -13 -19 -26 -34 -43 -53
```

```r
accumulate_right(a, `-`)
```

```
##  [1] -35 -34 -32 -29 -25 -20 -14  -7   1  10
```

These two functions keep the intermediary results.

### 3.3.3 Other useful functions from purrr

There are a lot of other useful functions in `purrr`. For example `safely()` and `possibly()` are great:

```r
a <- list("a", 4, 5)

sqrt(a)
```

```
Error in sqrt(a) : non-numeric argument to mathematical function
```

Using `map()` or `Map()` will result in a similar error. However, using `safely()` will work for the numbers contained in `a` and show an error for the first element of `a` which is a character:

```r
a <- list("a", 4, 5)

safe_sqrt <- safely(sqrt)

map(a, safe_sqrt)
```

```
## [[1]]
## [[1]]$result
## NULL
##
## [[1]]$error
## <simpleError in .f(...): non-numeric argument to mathematical function>
##
##
## [[2]]
## [[2]]$result
## [1] 2
##
## [[2]]$error
## NULL
##
##
## [[3]]
## [[3]]$result
## [1] 2.236068
##
## [[3]]$error
## NULL
```

And `possibly()` allows you to specify a return value in case of an error:

```
possible_sqrt <- possibly(sqrt, otherwise = NA_real_)

map(a, possible_sqrt)
```

```
## [[1]]
## [1] NA
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 2.236068
```

Of course, in this particular example, the same effect could be obtained way more easily:

```
sqrt(as.numeric(a))
```

```
## Warning: NAs introduced by coercion
```

```
## [1]       NA 2.000000 2.236068
```

However, in some situations, this trick does not work as intended (or at all), so `possibly()` and `safely()` are the way to go.

Another interesting function is `transpose()`. It is not an alternative to the function `t()` from `base` but, has a similar effect. `transpose()` works on lists. Let's take a look at the example from before:

```
safe_sqrt <- safely(sqrt, otherwise = NA_real_)

map(a, safe_sqrt)
```

```
## [[1]]
## [[1]]$result
## [1] NA
##
## [[1]]$error
## <simpleError in .f(...): non-numeric argument to mathematical function>
##
##
## [[2]]
## [[2]]$result
## [1] 2
##
## [[2]]$error
## NULL
##
##
## [[3]]
## [[3]]$result
## [1] 2.236068
##
## [[3]]$error
## NULL
```

The output is a list with the first element being a list with a result and an error message. One might want to have all the results in a single list, and all the error messages in another list. This is safe with transpose:

```
transpose(map(a, safe_sqrt))
```

```
## $result
## $result[[1]]
## [1] NA
##
## $result[[2]]
## [1] 2
##
## $result[[3]]
## [1] 2.236068
##
##
## $error
## $error[[1]]
## <simpleError in .f(...): non-numeric argument to mathematical function>
##
## $error[[2]]
## NULL
##
## $error[[3]]
## NULL
```

## 3.4 Wrap-up

- Make your functions referentially transparent.
- Avoid side effects (if possible).
- Make your functions do one thing (if possible).
- A function that takes another function as an argument is called an higher-order function. You can write your own higher-order functions and this is a way of having short and easily testable functions. Making these functions then work together is trivial and is what makes functional programming very powerful.

## 3.5 Exercises

For the following exercises, you will have to use any of the functions that we saw in this chapter. `Reduce()`, `Map()` or any function from the `*apply()` family of functions. Do not use loops! If you don't know how to solve these exercises wait for the next section, where we'll learn how to write unit tests. Writing unit tests before the functions they're supposed to test is called test-driven development and can help you write your functions.

1. Create a function that returns the factorial of a number using `Reduce()`. Remember: no recursion nor loops allowed!

```
my_fact(5)
```

```
[1] 120
```

2. Suppose you have a list of data set names. Create a function that removes ".csv" from each of these names. Start by creating a function that does so using `stri_split()` from the package `stringi` (you can also use `strsplit()` from base R). Below is an illustration of how it's supposed to work:

```
dataset_names <- c("dataset1.csv", "dataset2.csv", "dataset3.csv")

remove_csv(dataset_names)
```

```
[1] "dataset1" "dataset2" dataset3"
```

3. Create a function that takes a number `a`, and then returns either the sum of the numbers from 1 to this number that are divisible by another number `b` or the product of the numbers from 1 to this number that are divisible by `b`. Your function should be a higher-order function with the following arguments: `a` the number, `divisible_func` the function that checks whether a number is divisible by some number `b` and `reduce_op` the function that either sums or multiplies the numbers from 1 to `a` that are divisible by `b`.

```
reduce_some_numbers(a = 10, divisible_func = divisible, b = 2, reduce_op = `*`)
```

```
[1] 3840
```

# Chapter 4

# Unit testing

## 4.1 Introduction

Let's take a look at Wikipedia's definition of unit testing:

> In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use. Intuitively, one can view a unit as the smallest testable part of an application. In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure. In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method. Unit tests are short code fragments created by programmers or occasionally by white box testers during the development process. It forms the basis for component testing.

So unit tests are small pieces of code that test your code. They're called *unit* tests, because they test the smallest unit composing your code, in the case of functional programming, the smallest units are functions. You've probably been testing your code *manually* since you've started programming. For example, you would simply do something like this:

```
sqrt_newton(4, 1)
```

```
## [1] 2.00061
```

and check if the result is equal to 2 and stop there. Usually you would probably write this in the console and then forget about it. If you need to check again, you would write this small test again in the console. But what if some of your functions have to work together with other functions? Maybe changing something in these other functions will indirectly break in other functions. You would have to retest everything together again! In this chapter you will learn the basics of unit testing, which is simply writing these tests in a file, and running this file each time you change your code. If all your unit tests still pass, you can be more confident that your code works as intended.

Unit tests can also be useful to guide you as you program. Some programmers do test-driven development. These programmers start by writing the unit tests first, and then the code to make them pass. This can be useful sometimes, if you don't really know where you should start but know what you want.

## 4.2 Unit testing with the testthat package

We are going to test the function we wrote in the previous chapter, `sqrt_newton()`. The basic steps are:

1. Write a file containing your tests
2. Run the tests

It's very simple! You only need to install the `testthat` package for this. In this section I'll only show you how to write tests and try to illustrate their usefulness. In the next section, we'll see how we can run the tests.

Below is the code that we are going to put in the file `test_my_functions.R`:

```
library("testthat")

test_that("Test sqrt_newton: positive numeric",{
              expected <- 2
              actual <- sqrt_newton(4, 1)
              expect_equal(expected, actual)
})
```

The syntax of the test is pretty straightforward. We start with a short description of what the test is about, and then we define two variables: the result we expect, and the actual result that is returned by the function we wish to test. When we run this test (we'll discuss running tests in the next section), this is what we get:

```
Error: Test failed: 'Test sqrt_newton: positive numeric'
* `expected` not equal to `actual`.
1/1 mismatches
[1] 2 - 2 == -0.00061
```

This is because the value that `sqrt_newton()` returns is not exactly equal to `2`. How to solve this? We could simply check if the difference of the value expected and the value returned is smaller than `eps` (which is actually how the function works):

```
library("testthat")
```

```
##
## Attaching package: 'testthat'
```

```
## The following object is masked from 'package:purrr':
##
##     is_null
```

```
test_that("Test sqrt_newton: positive numeric",{
              eps <- 0.001
              expected <- 2
              actual <- sqrt_newton(4, 1, eps = eps)
              expect_lt(abs(expected - actual), eps)
})
```

There's no visible output, meaning that the test passes. Don't worry, we'll see how to run these tests in the next section, and we'll get a nice output confirming that tests did, indeed, pass.

I didn't talk about the functions `expect_equal()` and `expect_lt()`, but now is the moment. These functions are part of the `testthat` package and these are what allow you to test your functions. There's a number of them that allow you to test for a variety of situations. Check the documentation of `testthat` for more info. Let's continue to write more tests!

```
library("testthat")

test_that("Test sqrt_newton: negative numeric",{
              expect_error(sqrt_newton(-4, 1))
})
```

We would like our function to return an error message if the user tries to get the square root of a negative number (let's say we don't want to generalize our function to complex numbers). But what happens here is that the function runs forever! This is because we are using a while loop whose condition is never fulfilled. This test basically allowed us to find two problems with our function:

- it doesn't deal with negative numbers
- the while loop may run forever if the condition is never fulfilled (for example if `eps` is too small

Let's rewrite our function to take care of this, one problem at a time:

```
sqrt_newton <- function(a, init, eps = 0.01){
    stopifnot(a >= 0)
    while(abs(init**2 - a) > eps){
        init <- 1/2 *(init + a/init)
    }
    return(init)
}
```

Now let's run our test again:

```
library("testthat")

test_that("Test sqrt_newton: negative numeric",{
              expect_error(sqrt_newton(-4, 1))
})
```

Again no output, so things are good. Now to the next issue: we need to write a safeguard in the function to avoid having the while loop running for too long. For example if you try to run this:

```
sqrt_newton(49, 1E100000, 1E-100000)
```

You will see that it takes an awful lot of time! Let's limit the number of iterations to 100.

```
sqrt_newton <- function(a, init, eps = 0.01){
    stopifnot(a >= 0)
    i <- 1
    while(abs(init**2 - a) > eps){
        init <- 1/2 *(init + a/init)
        i <- i + 1
        if(i > 100) stop("Maximum number of iterations reached")
    }
    return(init)
}
```

Now when we try to run the following expression we get an error message:

```
sqrt_newton(49, 1E100, 1E-100)
```

```
Error in sqrt_newton(49, 1e+100, 1e-100) :
  Maximum number of iterations reached
```

But wouldn't it be better if the user could change the number of iterations himself?

```
sqrt_newton <- function(a, init, eps = 0.01, iter = 100){
    stopifnot(a >= 0)
    i <- 1
    while(abs(init**2 - a) > eps){
        init <- 1/2 *(init + a/init)
        i <- i + 1
        if(i > iter) stop("Maximum number of iterations reached")
    }
    return(init)
}
```

We can now write some more tests:

```
library("testthat")

test_that("Test sqrt_newton: not enough iterations",{
              expect_error(sqrt_newton(4, 1E100, 1E-100, iter = 100))
})
```

## 4.3 Actually running your tests

One of the easiest ways to run your tests is when your developing a package. We are going to see this in the next chapter, but for now, let's suppose that we have a folder called `my_project` with the code inside of it. There's a file called `my_functions.R` and another file called `test_my_functions.R` which contain the functions you programmed and the unit tests that go with it respectively.

The file `test_my_functions.R` contains the following source code:

```
library("testthat")

test_that("Test sqrt_newton: positive numeric",{
    eps <- 0.001
    expected <- 2
    actual <- sqrt_newton(4, 1, eps = eps)
    expect_lt(abs(expected - actual), eps)
})

test_that("Test sqrt_newton: negative numeric",{
    expect_error(sqrt_newton(-4, 1))
})

test_that("Test sqrt_newton: not enough iterations",{
    expect_error(sqrt_newton(4, 1E100, 1E-100, iter = 100))
})
```

Then you simply run the following in the console:

```
test_file("test_my_functions.R")
```

of course you have to make sure that you are in the correct working directory. This can be tricky, and is one of the reasons why it's easier to run your tests when you're developing a package.

This is the output we get:

```
...
DONE ===============================================================================
```

See the three dots on the first line? Each dot represents a test that passed successfully. Let's add a test that will not pass on purpose, just to see what happens:

```
test_that("Test sqrt_newton: wrong on purpose",{
    eps <- 0.001
    expected <- 12
    actual <- sqrt_newton(4, 1, eps = eps)
    expect_lt(abs(expected - actual), eps)
})
```

This is the output we get now:

```
...1
Failed -----------------------------------------------------------------------------
1. Failure: Test sqrt_newton: wrong on purpose (@test_my_functions.R#22) ------------
abs(expected - actual) is not strictly less than `eps`. Difference: 10


DONE ===============================================================================
```

You can then go back to the file that contains the tests and correct them. If all your tests are in a separate folder, you can use the function `test_dir()` to test all the functions in a given folder. The files containing your tests should all start with the string `test`. You could have a file called `run_tests.R` on the root of the directory and this file could contain the following:

```
library("testthat")

test_dir("tests")
```

You could then run your tests by running this file. You might also be tempted to write a bash script on GNU/Linux distributions or on macOS:

```
#!/bin/sh

Rscript -e "testthat::test_that('/whole/path/to/your/tests')"
```

but you'll probably only get burned because when you run this script, a new R session is started which does not know anything about your functions in your file `my_functions.R`. Managing the working directory is quite a pain. This is why in the next chapter we are going to start learning about packages and why writing our own packages to clean datasets is the best possible way to write your code.

## 4.4 Wrap-up

- Unit tests are a way of testing your code, and more specifically your functions.
- The basic workflow is to write your code, write tests, and check if your tests pass.
- You can also start with the tests and then write or modify your code to make them pass.
- We didn't talk about *coverage* yet. Are you sure that you test every line of your function? No you're not. In the next chapter I'll show you how can be sure to test each line of your function with the `covr` package.

## 4.5 Exercises

1. Write unit tests for the functions you wrote in the previous chapter. Just play around a little bit, and get a feeling for unit tests.

# Chapter 5

# Packages

## 5.1 Why you need your own packages in your life

One of the reasons you might have tried R in the first place is the abundance of packages. As I'm writing these lines (in August 2016), 8922 packages are available on CRAN. That's almost over 9000. This is an absolutely crazy amount of packages! Chances are that if you want to do something, there's a package for that (I'll stop it here with the lame references, promise!).

So why the heck should you write your own packages? After all, with 8922 packages you're sure to find something that suits your needs, right? No. Simply because the data sets that you're working with are probably unique to your workplace or maybe what you want to do with them is unique to your needs. You won't find a package that will take care of cleaning *your* data for you.

Ok, but is it necessary to write a package? Why not just write functions inside some scripts and then simply run these scripts? This seems like a valid solution at first. However, it quickly becomes tedious, especially if you have multiple scripts scattered around your computer or inside different subfolders. You'll also have to write the documentation on separate files and these can easily get lost or become outdated.

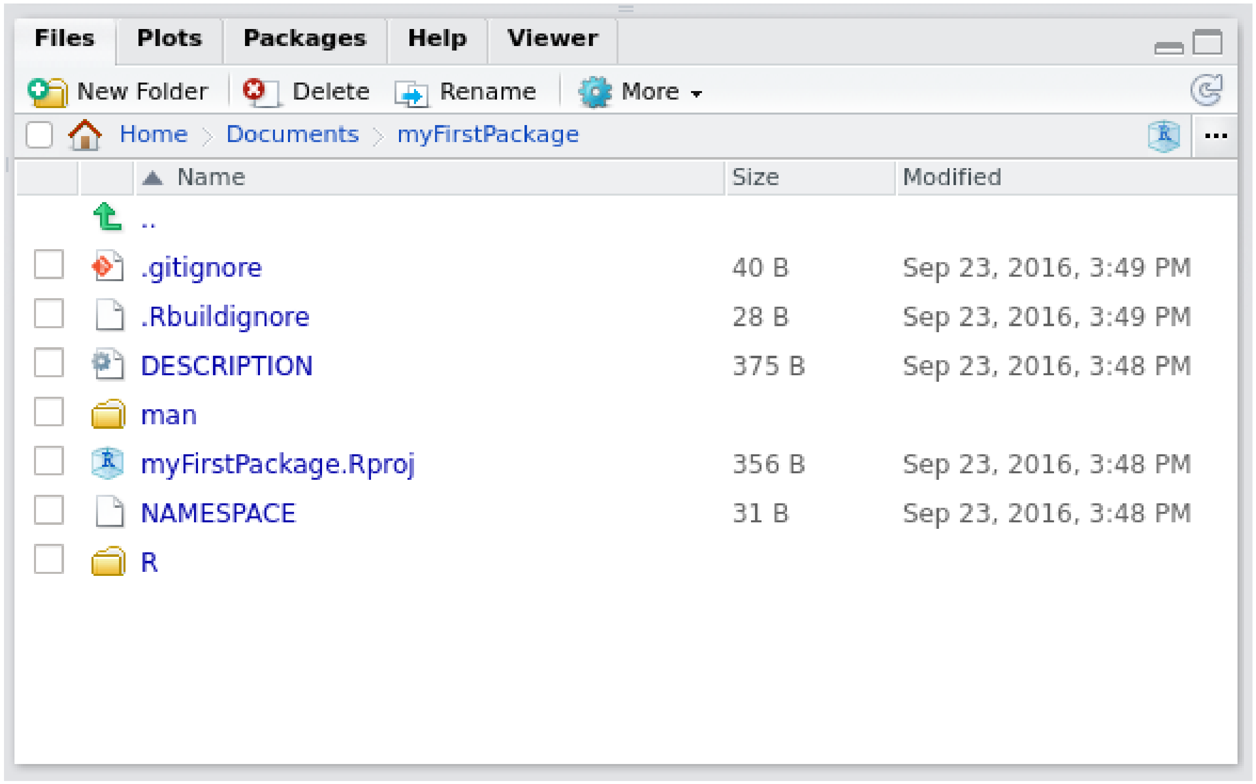
Having everything inside a package takes care of these headaches for you. And code that is inside packages is very easy to test, especially if you're using Rstudio. It also makes it possible to use the wonderful `covr` package, which tells you which lines in which functions are called by your tests. If some lines are missing, write tests that invoke them and increase the coverage of your tests!

As I mentioned in the introduction, if you want to learn much more than I'll show about packages read Wickham (2014). I will only show you the basics, but it should be enough to get you productive.

One last thing: if you don't know git, you really should learn git. I won't talk about it here, because there's a ton of books on git, such as Silverman (2013). I learned by reading it and googling whenever I had a problem. Learning git is really worth it, especially if you're collaborating with some colleagues on your packages.

## 5.2 R packages: the basics

To start writing a package, the easiest way is to load up Rstudio and start a new project, under the *File* menu. If you're starting from scratch, just choose the first option, *New Directory* and then *R package*. Give a new to your package, for example `myFirstPackage` and you can also choose to use git for version control. Now if you check the folder where you chose to save your package, you will see a folder with the same name as your package, and inside this folder a lot of new files and other folders. The most important folder for now is the `R` folder. This is the folder that will hold your `.R` source code files. You can also see these files and folders inside the *Files* panel from within Rstudio. Rstudio will also have `hello.R` opened, which is a single demo source file inside the `R` folder. You can get rid of this file.



| Name | Size | Modified |
|---|---|---|
| .. | | |
| .gitignore | 40 B | Sep 23, 2016, 3:49 PM |
| .Rbuildignore | 28 B | Sep 23, 2016, 3:49 PM |
| DESCRIPTION | 375 B | Sep 23, 2016, 3:48 PM |
| man | | |
| myFirstPackage.Rproj | 356 B | Sep 23, 2016, 3:48 PM |
| NAMESPACE | 31 B | Sep 23, 2016, 3:48 PM |
| R | | |

The picture above shows the basic structure of your package. As a first step, create a script called `square_root_loop.R` and put the following code in it:

```r
sqrt_newton <- function(a, init, eps = 0.01, iter = 100){
    stopifnot(a >= 0)
    i <- 1
    while(abs(init**2 - a) > eps){
        init <- 1/2 *(init + a/init)
        i <- i + 1
        if(i > iter) stop("Maximum number of iterations reached")
    }
    return(init)
}
```

Then save this script. You can now test your package by building your package, either by clicking on the button named *Build and Reload* button which you can find inside the *Build* pane or by using the following keyboard shortcut: `CTRL-SHIFT-B`. You will use *Build and Reload* quite often, so I advise you remember this shortcut! In the next section we will see how we can add documentation to our functions.

## 5.3 Writing documentation for your functions

Writing documentation for your functions is very streamlined, thanks to the `roxygen2` package. Suppose we want to write documentation for our square root function:

```
sqrt_newton <- function(a, init, eps = 0.01, iter = 100){
    stopifnot(a >= 0)
    i <- 1
    while(abs(init**2 - a) > eps){
        init <- 1/2 *(init + a/init)
        i <- i + 1
        if(i > iter) stop("Maximum number of iterations reached")
    }
    return(init)
}
```

Usually, you would write comments to describe what your function does, what are its inputs and outputs. 'roxygen2' is a package that turns these comments into documentation. Here is what our function would look like with `roxygen2` type comments:

```
#' Function to compute the square root of a number
#' @param a the number whose square root is computed
#' @param init an initial guess
#' @param eps *optional* the precision. Default value: 0.01
#' @param iter *optional* the number of iteration. Default value: 100
#' @description This function computes the square root of a number using a loop.
#' @export
sqrt_newton <- function(a, init, eps = 0.01, iter = 100){
    stopifnot(a >= 0)
    i <- 1
    while(abs(init**2 - a) > eps){
        init <- 1/2 *(init + a/init)
        i <- i + 1
        if(i > iter) stop("Maximum number of iterations reached")
    }
    return(init)
}
```

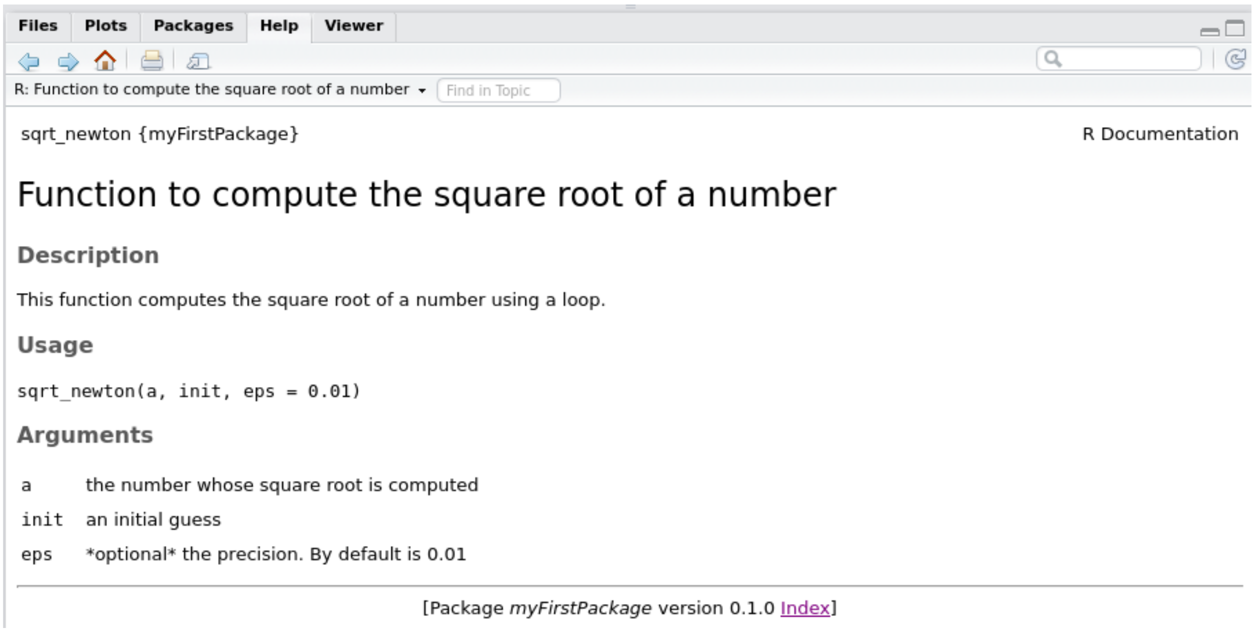
The first difference with standard comments is that `roxygen2` type comments start with the `#'` symbol instead of simply the `#` symbol. Then, after `#'` you can supply different keywords such as `@param`, `@description`, `@export`. These keywords are then used by the `roxygenise()` function from the `roxygen` package to create the documentation files inside your package. Before `roxygen`, these documentation files were written in the `.Rd` format by hand. Now these files get created automagically by simply formatting your comments with this specific syntax and then running

```
roxygen2::roxygenise()
```

in the command prompt. Try it, you should see the following in the command prompt:

```
Writing sqrt_newton.Rd
```

then you can *Build and Reload* your package again using `CTRL-SHIFT-B`. If you go check the documentation of your function inside your package, this is what you should see:



sqrt_newton {myFirstPackage} R Documentation

Function to compute the square root of a number

Description

This function computes the square root of a number using a loop.

Usage

```
sqrt_newton(a, init, eps = 0.01)
```

Arguments

| | |
|---|---|
| a | the number whose square root is computed |
| init | an initial guess |
| eps | *optional* the precision. By default is 0.01 |

[Package *myFirstPackage* version 0.1.0 Index]

There is still a keyword that I did not mention: the `@export` keyword. This keyword is needed if you want your function to be accessible by the user without prepending the package name, like this:

```
my_package::my_function
```

Not using `@export` can be useful though, if you want to have helper functions that are used by your other functions inside your package, and if you wish to not make these functions accessible to the users.

## 5.4 Unit test your package

Now that we know the basics of creating a package, we move on to unit testing your package. Unit testing is very useful, but require some work, especially because you have to run them often to make them truly worth your time. However running them often can be painful because you have to be careful with the current working directory. The simplest way to do unit testing is to put your functions inside a package and write unit tests for these functions and use Rstudio's keyboard shortcuts to run your tests. First of all, create a folder called `inst` in the root of your package and inside this `inst` folder create another folder, called `tests`. The `tests` folder will hold your unit tests and everything that is inside the `inst` folder gets installed along with the package, so the unit tests are also available to the users, along the source code of the package. Inside the `tests` folder, create a script called `test_sqrt_newton.R` and put the following code in it:

```r
library("testthat")
library("myFirstPackage")

test_that("Test sqrt_newton: positive numeric",{
    eps <- 0.001
    expected <- 2
    actual <- sqrt_newton(4, 1, eps = eps)
    expect_lt(abs(expected - actual), eps)
})
```

Save this file and use the following keyboard shortcut: `CTRL-SHIFT-T` to run your unit test. You will see the following output:

```
==> devtools::test()

Loading myFirstPackage
Loading required package: testthat
Testing myFirstPackage
.
DONE ===========================================================================
```

You can of course add more unit tests inside the same file. Add the following code to `test_sqrt_newton.R`:

```
test_that("Test sqrt_newton: negative numeric",{
    expect_error(sqrt_newton(-4, 1))
})
```

You will now see the following output:

```
==> devtools::test()

Loading myFirstPackage
Loading required package: testthat
Testing myFirstPackage
..
DONE ===========================================================================
```

Notice the two `.` above `DONE`. This means that two unit tests passed. If a unit test does not pass, you will of course get notified. For example, add the following test to `test_sqrt_newton.R`:

```
test_that("Test sqrt_newton: with a string!",{
    expect_equal(4, sqrt_newton("WontWork", 1))
})
```

and if you try running your tests this is what you will see:

```
==> devtools::test()

Loading myFirstPackage
Loading required package: testthat
Testing myFirstPackage
..1
Failed -------------------------------------------------------------------------
1. Error: Test sqrt_newton: with a string! (@test_sqrt_newton.R#15) ------------
non-numeric argument to binary operator
1: expect_equal(4, sqrt_newton("WontWork", 1)) at /home/bro/Documents/myFirstPackage/inst/tests/test_sqrt
2: compare(object, expected, ...)
3: compare.numeric(object, expected, ...)
4: all.equal(x, y, tolerance = tolerance, ...)
5: all.equal.numeric(x, y, tolerance = tolerance, ...)
6: attr.all.equal(target, current, tolerance = tolerance, scale = scale, ...)
7: mode(current)
8: sqrt_newton("WontWork", 1)

DONE ===========================================================================
```

You can then either modify the test if you made a mistake writing the test, or amend your function if your test is correct and needs to pass, but does not because there is an error in your function. For now, simply remove these lines for your `test_sqrt_newton.R` script.

Another interesting feature you should use once in a while, is the *Check Package* command using `CTRL-SHIFT-E`. This command will find errors and other mistakes and warns you. For example, when I ran this command I got the following report:

```
checking DESCRIPTION meta-information ... WARNING
Non-standard license specification:
  What license is it under?
Standardizable: FALSE

checking for code/documentation mismatches ... WARNING
Codoc mismatches from documentation object 'sqrt_newton':
sqrt_newton
  Code: function(a, init, eps = 0.01, iter = 100)
  Docs: function(a, init, eps = 0.01)
  Argument names in code not in docs:
    iter
```

*Check Package* is telling me that I did not specify a license for my package, and that I did not document the `iter` parameter. This command takes some time to run, so do not run it as often as your unit tests, but do not forget about it either!

## 5.5 Checking the coverage of your unit tests with `covr`

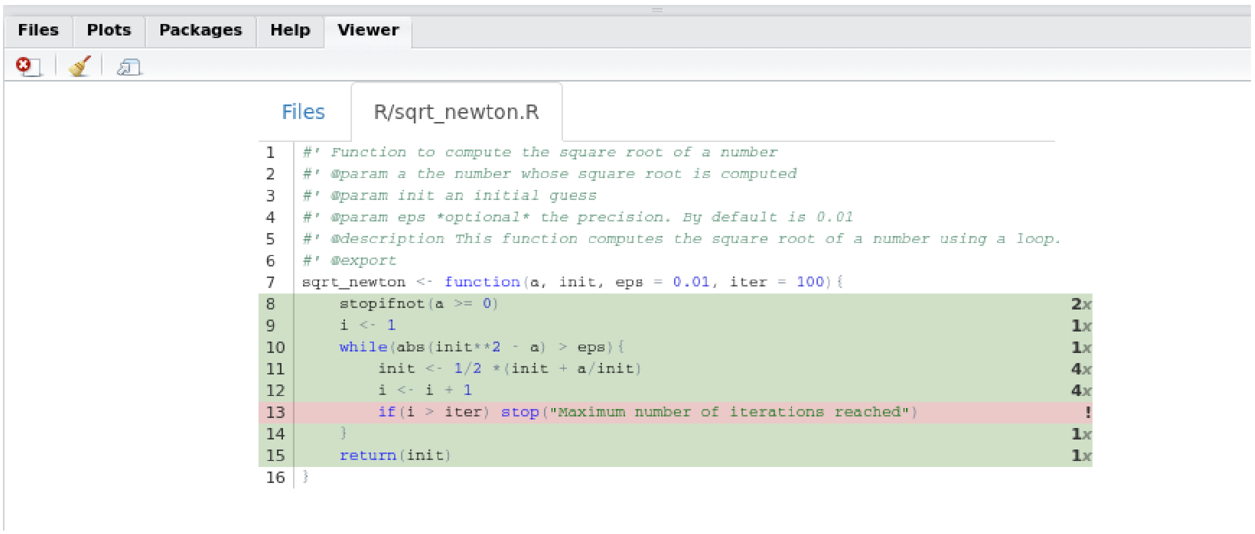
To check the coverage of your package run the following code:

```r
library("covr")

cov <- package_coverage()

shine(cov)
```

The line `shine(cov)` launches an interactive shiny app inside your viewer pane with the following:

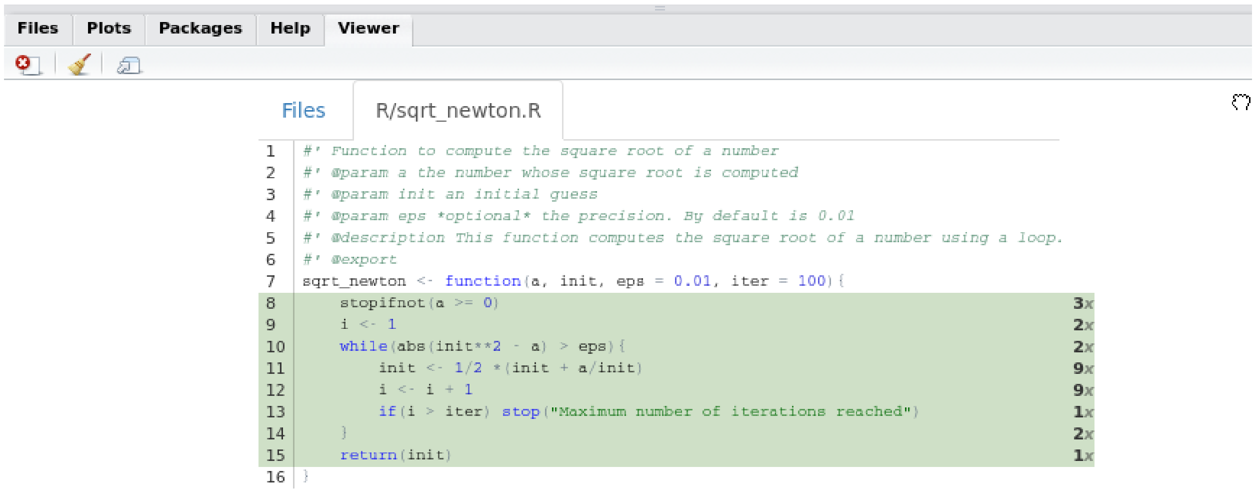


```r
#' Function to compute the square root of a number
#' @param a the number whose square root is computed
#' @param init an initial guess
#' @param eps *optional* the precision. By default is 0.01
#' @description This function computes the square root of a number using a loop.
#' @export
sqrt_newton <- function(a, init, eps = 0.01, iter = 100){
    stopifnot(a >= 0)                                                 2x
    i <- 1                                                            1x
    while(abs(init**2 - a) > eps){                                    1x
        init <- 1/2 *(init + a/init)                                  4x
        i <- i + 1                                                    4x
        if(i > iter) stop("Maximum number of iterations reached")      !
    }                                                                 1x
    return(init)                                                      1x
}
```

We see that no unit test executes the highlighted line. So let's write a unit test to test this line and increase the coverage of our package! Add the following test to `test_sqrt_newton.R`:

```r
test_that("Test maximum number of iterations",{
    expect_error(sqrt_newton(10, 1E10, eps=1E-10, 5))
})
```

Now if you look at the coverage of the package:



```r
#' Function to compute the square root of a number
#' @param a the number whose square root is computed
#' @param init an initial guess
#' @param eps *optional* the precision. By default is 0.01
#' @description This function computes the square root of a number using a loop.
#' @export
sqrt_newton <- function(a, init, eps = 0.01, iter = 100){
    stopifnot(a >= 0)                                               3x
    i <- 1                                                          2x
    while(abs(init**2 - a) > eps){                                  2x
        init <- 1/2 *(init + a/init)                                9x
        i <- i + 1                                                  9x
        if(i > iter) stop("Maximum number of iterations reached")   1x
    }                                                               2x
    return(init)                                                    1x
}
```

In this example, we used `package_coverage()`, but if you are interested in the coverage of a single function you can use `function_coverage()`, or even `file_coverage()` to get the coverage of a single file. However, I suggest to always run `package_coverage()` since we are working inside a package. There are other functions in the `covr` package that might be useful depending on your needs, so do not hesitate to explore `covr` documentation!

## 5.6 Wrap-up

- Packages are the easiest way to organize, document and test your code.
- You do not need to take care of paths anymore.
- You do not need to write documentation "by hand".
- If you use Rstudio, the workflow is very streamlined and you can use version control to keep track of your changes.
- Developing a package is also the easiest way to share your code with colleagues at your company or online.

# Chapter 6

# Putting it all together: writing a package to work on data

Everything we have seen until now allows us to develop our own packages with the goal of *working* on data. By *working* on data I mean any operation that involves cleaning, transforming, analyzing or plotting data. I will summarize why everything we have seen until now helps us in this task:

1. Functional programming makes our code easier to test
2. Unit tests make sure our code is correct
3. Packages allows us to forget about paths, so unit tests are easier to run, makes writing documentation easier and makes sharing our code easier

For the rest of this chapter we are going to work with mock datasets that I created. The data is completely random but for our purposes it does not matter. In this chapter, we are going to write a number of functions with the goal of going from these awful, badly formatted datasets to a nice longitudinal data set.

## 6.1 Getting the data

You can download the data from the github repository of the book. There are 5 `.csv` files that comprise the data sets we are going to work with:

- `data2000.csv`
- `data2001.csv`
- `data2002.csv`
- `data2003.csv`
- `data2004.csv`

The first step, of course, is to load these datasets into R. For 5 datasets, I suppos that you would simply write the following into Rstudio:

```r
data2000 <- read.csv("/path/to/data/data2000.csv", header = T)
data2001 <- read.csv("/path/to/data/data2001.csv", header = T)
data2002 <- read.csv("/path/to/data/data2002.csv", header = T)
data2003 <- read.csv("/path/to/data/data2003.csv", header = T)
data2004 <- read.csv("/path/to/data/data2004.csv", header = T)
```

This might be ok for 5 datasets which are named very similarily, especially since you can do block editing in Rstudio. However, imagine that you have hundreds, thousands, of datasets? And image that their names are not so well formatted as here? We will start our package by writing a function that reads a lot of datasets at once.

## 6.2 Your first data munging package: `prepareMyData`

Using Rstudio, create a new project like shown in the previous chapter, and select *R package*. Give it a name, for example `prepareMyData`. If you are working with datasets that have a name, for example the *Penn World Tables*, you could call your package `preparePWT`, or something similar. The first function I will show you is actually very general and could work with any datasets. This means that I created a package called `brunoUtils` (very inspired, I know) that contains all the little functions that I use daily. But for illustration purposes, we will put this function inside `prepareMyData`, even if it does not have anything directly to do with it. I have called this function `read_list()` and here is the source code:

```r
#' Reads a list of datasets
#' @param list_of_datasets A list of datasets (names of datasets are strings)
#' @param read_func A function, the read function to use to read the data
#' @return Returns a list of the datasets
#' @export
#' @examples
#' \dontrun{
#' setwd("path/to/datasets/")
#' list_of_datasets <- list.files(pattern = "*.csv")
#' list_of_loaded_datasets <- read_list(list_of_datasets, read_func = read.csv)
#' }
read_list <- function(list_of_datasets, read_func, ...){

    stopifnot(length(list_of_datasets)>0)
    read_and_assign <- function(dataset, read_func){
        dataset_name <- as.name(dataset)
        dataset_name <- read_func(dataset, ...)
}

    # invisible is used to suppress the unneeded output
    output <- invisible(
        sapply(list_of_datasets,
               read_and_assign, read_func = read_func, simplify = FALSE, USE.NAMES = TRUE))

    # Remove the ".csv" at the end of the data set names
    names_of_datasets <- c(unlist(strsplit(list_of_datasets, "[.]"))[c(T, F)])
    names(output) <- names_of_datasets
    return(output)
}
```

The basic idea of `read_list()` is that it takes a list of datasets as the first argument, then a functon to read in the datasets as a second argument and as a third argument the famous `...`, which allows the user to specify further options to other functions that are contained in the body of the function. In this case, further arguments are passed to the `read_func` function, for example if your data does not contains headers, you could pass the option `header = FALSE` to `read_list()` which would then get passed to `read_func`.

# Bibliography

Khan, A. (2017). *Grokking functional programming.* Manning Publications, 1st edition. ISBN 9781617291838.

Lipovaca, M. (2011). *Learn You a Haskell for Great Good!: A Beginner's Guide.* no starch press.

Silverman, R. E. (2013). *Git Pocket Guide.* " O'Reilly Media, Inc.".

Wickham, H. (2014). *Advanced R.* CRC Press. http://adv-r.had.co.nz/.

Wickham, H. (2015). *R packages.* O'Reilly, 1st edition. ISBN 978-1491910597.

Wickham, H. and Grolemund, G. (2016). *R for Data Science.* O'Reilly, 1st edition.