Functional programming and unit testing for data munging with R

Bruno Rodrigues

2017-07-31

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 (2014a), Wickham (2015) and Wickham and Grolemund (2016) 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. Then we will learn to develop our own packages; not with the goal of publishing them in CRAN, but with the goal of making programming more streamlined.

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

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: to check the coverage of your unit tests
- dplyr: to clean, transform, prepare data
- lazyeval: for lazy evaluation
- lubridate: makes working with dates easier
- memoise: makes your function remember intermediate results
- purrr: extends R's functional programming capabilities
- readr: provides alternative functions to `read.csv()` and such
- roxygen2: creates documentation files from comments
- stringr: makes working with characters easier
- testthat: the library we are going to use for unit testing
- tibble: provides a nice, cleaner alternative to `data.frame`
- tidyr: works hand in hand with `dplyr`

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, especially in applied sciences like applied econometrics. 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 has to be done by writing each instruction in a file, then running everything. Sometimes this statistician has 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 easily documented and tested (more on this below). Because each function performs a single task and is well documented, it is also easier to understand what the program is supposed to do. Comments in a thousand-lines file are actually not that much useful. The file is so long, that even when commented you simply cannot make any sense of what is going on. 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 `CTRL-F`ing around. 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*).

2.4 General recommendations to follow this book

You should follow the examples in this book as closely as possible. I advise you to use exactly the same names as I do, because I will sometimes refer to previous examples and if you use different names, you will then need to go back and waste time to change the names.

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

```
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):

```
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:

```
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:

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

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

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

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

```
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 `+`[1] 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)
    }
}
```

[1] 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.

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!

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

reduce(a, `-`)
```

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

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

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

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

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

accumulate(a, `-`)
```

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

```
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:

```
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:

```
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 Anonymous functions

One last very useful concept are anonymous functions. Suppose that you want to apply one of your own functions to a list of datasets. For instance, you want to have a histogram of a variable that is called the same accross a list of datasets. Maybe your datasets are yearly surveys and each year the survey was conducted is another `.csv` file. For illustration purposes, let us use the `mtcars` dataset with some minor changes:

```
data(mtcars)

mtcars2000 <- mtcars
mtcars2001 <- mtcars
mtcars2001$cyl <- mtcars2001$cyl+3
datasets <- list("mtcars2000" = mtcars2000,
"mtcars2001" = mtcars2001)
```

In the next chapters we will learn how to load a lot of datasets at once and store them in a list. So it is important to know how to work with datasets that are stored on lists. Now suppose you want to use `purrr::map()` to plot a histogram of variable `cyl` for each dataset that is contained in your list.

```
map(datasets, hist, cyl)
```

```
Error in hist.default(.x[[i]], ...) : 'x' must be numeric
```

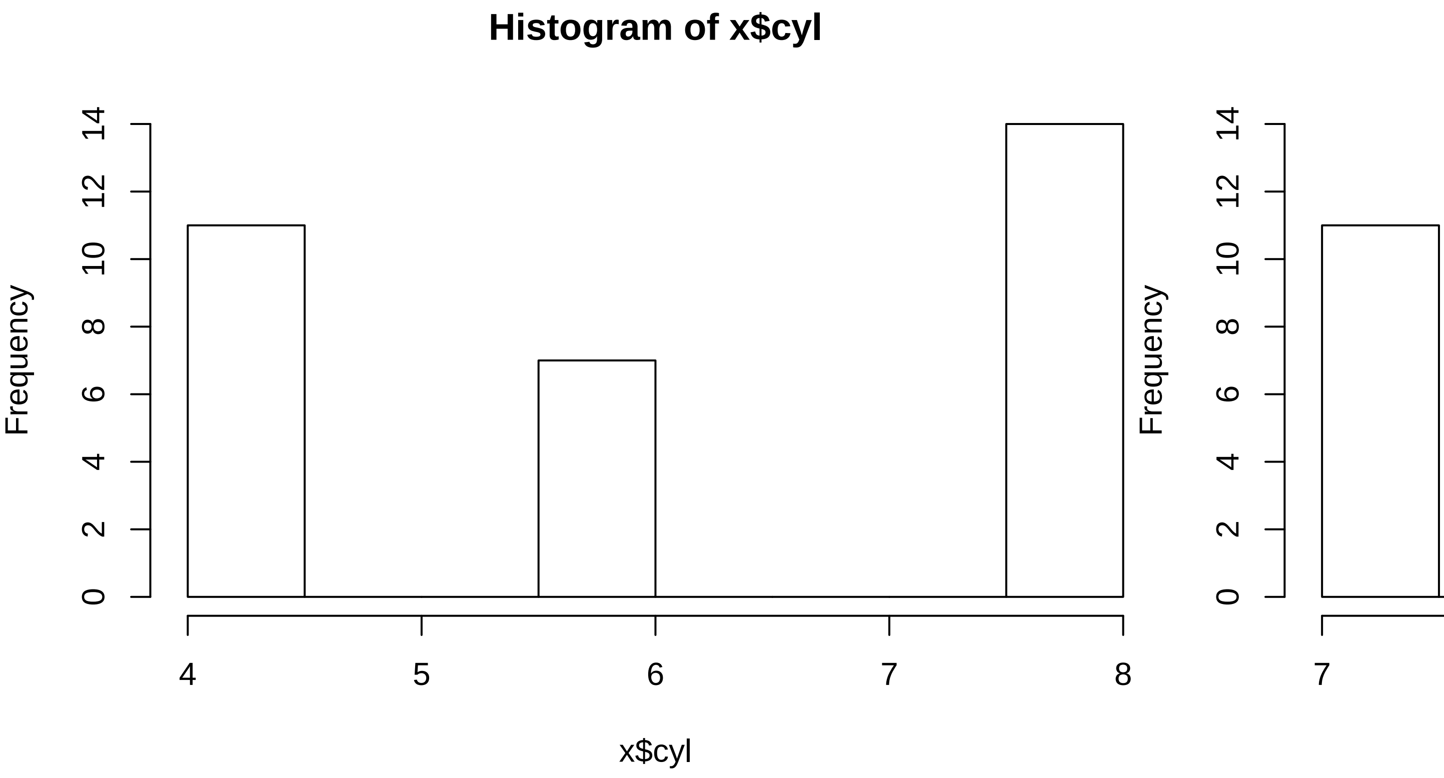
Maybe try this:

```
map(datasets, hist(cyl))
```

```
Error in hist(cyl) : object 'cyl' not found
```

So how can we solve this issue? One way is to use an anonymous function. Anonymous functions are functions that get declared on the fly and do not have names. These are especially useful inside higher order functions such as `purrr::map()`:

```
map(datasets, (function(x) hist(x$cyl)))
```

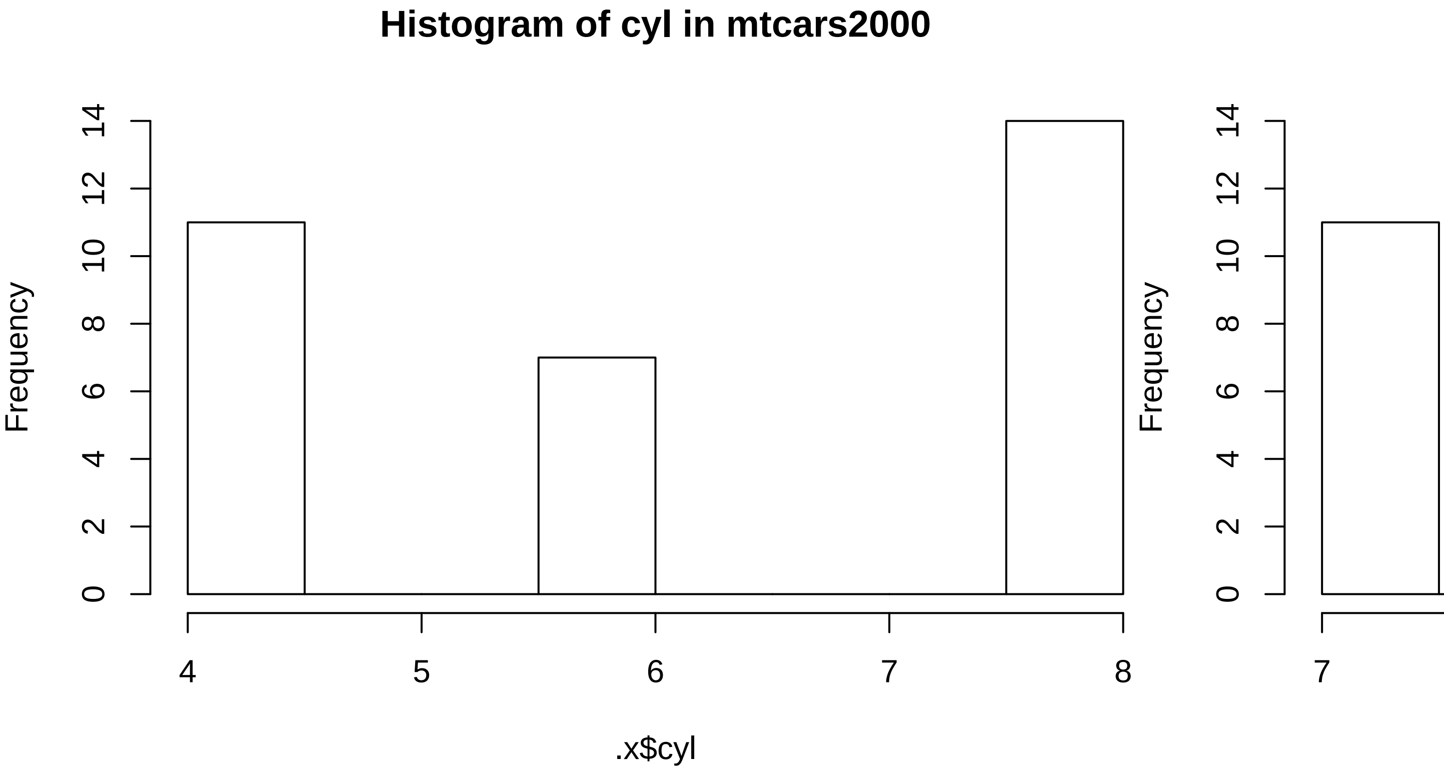


```
## $mtcars2000
## $breaks
## [1] 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0
##
## $counts
## [1] 11  0  0  7  0  0  0 14
##
## $density
## [1] 0.6875 0.0000 0.0000 0.4375 0.0000 0.0000 0.0000 0.8750
##
## $mids
## [1] 4.25 4.75 5.25 5.75 6.25 6.75 7.25 7.75
##
## $xname
## [1] "x$cyl"
##
## $equidist
## [1] TRUE
##
## attr(,"class")
## [1] "histogram"
##
## $mtcars2001
## $breaks
## [1]  7.0  7.5  8.0  8.5  9.0  9.5 10.0 10.5 11.0
##
## $counts
## [1] 11  0  0  7  0  0  0 14
##
## $density
## [1] 0.6875 0.0000 0.0000 0.4375 0.0000 0.0000 0.0000 0.8750
##
## $mids
## [1]  7.25  7.75  8.25  8.75  9.25  9.75 10.25 10.75
##
## $xname
## [1] "x$cyl"
##
## $equidist
## [1] TRUE
##
## attr(,"class")
## [1] "histogram"
```

Here the function is enclosed between `()` and is not named. The function has a single argument `x`, which is supposed to be a dataset. We then plot a histogram of the variable `cyl` from this dataset. Then this function is mapped to every dataset contained in the list `datasets` that we created above.

We can also write anonymous functions that are more complex:

```
map2(
.x = datasets, .y = names(datasets),
(function(.x, .y) hist(.x$cyl, main=paste("Histogram of cyl in", .y)))
)
```



```
## $mtcars2000
## $breaks
## [1] 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0
##
## $counts
## [1] 11  0  0  7  0  0  0 14
##
## $density
## [1] 0.6875 0.0000 0.0000 0.4375 0.0000 0.0000 0.0000 0.8750
##
## $mids
## [1] 4.25 4.75 5.25 5.75 6.25 6.75 7.25 7.75
##
## $xname
## [1] ".x$cyl"
##
## $equidist
## [1] TRUE
##
## attr(,"class")
## [1] "histogram"
##
## $mtcars2001
## $breaks
## [1]  7.0  7.5  8.0  8.5  9.0  9.5 10.0 10.5 11.0
##
## $counts
## [1] 11  0  0  7  0  0  0 14
##
```

```
## $density
## [1] 0.6875 0.0000 0.0000 0.4375 0.0000 0.0000 0.0000 0.8750
##
## $mids
## [1]  7.25  7.75  8.25  8.75  9.25  9.75 10.25 10.75
##
## $xname
## [1] ".x$cyl"
##
## $equidist
## [1] TRUE
##
## attr(,"class")
## [1] "histogram"
```

Of course you could have defined the anonymous function as a regular function before using `map()`. But sometimes it is faster to simply use an anonymous function as long as it does not hurt clarity.

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 mapping and reducing is enough to get us going.

3.5 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.

3.6 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

The tidyverse

The tidyverse is the name given to a certain number of packages, most of all (if not all?) developed by, or co-developed by, Hadley Wickham. There's a website that introduces them all: The tidyverse. In this chapter, we are going to learn about some functions of some of these packages. We already know a little bit about purrr; let's discover what these other packages have to offer!

4.1 Getting to know the tidyverse

Before reading everything that follows, I'd suggest you watch Hadley Wickham's talk Expressing yourself with R. R is a computer *language*, and as with any language we really are writing things that are supposed to be read and understood by others, not just the computer. Even if you're working alone, you owe it to your future self to write clean, easy to understand code. Using the tidyverse and adopting the principles presented in the talk will put you in the right mindset for everything that follows!

First of all, let's install the tidyverse packages. You can install them one by one, or you can install the tidyverse meta-package:

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

I suggest you do just that, as we're going to skim over all the packages. To start an analysis, we first have to import data into R.

4.1.1 Smoking is bad for you, but pipes are your friend

The title of this section might sound weird at first, but by the end of it, you'll get this (terrible) pun.

You probably know the following painting by René Magritte, *La trahison des images*:

```
knitr::include_graphics("assets/pas_une_pipe.png")
```

It turns out there's an R package from the `tidyverse` that is called `magrittr`. What does this package do? It brings *pipes* to R. Pipes are a concept from the Unix operating system; if you're using a GNU+Linux distribution or macOS, you're basically using a *modern* unix. (That's an oversimplification, but I'm an economist by training, and outrageously oversimplifying things is what we do, deal with it.)

The idea of pipes is to take the output of a command, and *feed* it as the input of another command. The `magrittr` package brings pipes to R, by using the weird looking `%>%`. Try the following:

```
library(magrittr)
```

```
16 %>% sqrt
```

```
## [1] 4
```

Super weird right? But you probably understand what happened; `16` got fed as the first argument of the function `sqrt()`. You can chain multiple functions:

```
16 %>% sqrt %>% `+`(18)
```

```
## [1] 22
```

The output of `16` (16) got fed to `sqrt()`, and the output of `sqrt(16)` (4) got fed to `+(18)` (22). Without `%>%` you'd write the line just above like this:

```
sqrt(16) + 18
```

```
## [1] 22
```

It might not be very clear right now why this is useful, but the `%>%` is probably one of the best things that R has, because when using packages from the `tidyverse`, you will naturally want to chain a lot of functions together. Without the `%>%` it would become messy very fast.

%>% is not the only pipe operator in magrittr. There's %T%, %<>% and %$%. All have their uses, but are basically shortcuts to some common tasks with %>% plus another function. Which means that you can live without them, and because of this, I will only discuss them briefly once we'll have learned about the other tidyverse packages.

4.1.2 Getting data into R with readr, readxl, haven and what are *tibbles*

You probably already know how to import data in R, but maybe you are not familiar with these packages. Using them is pretty straightforward, and I will only discuss haven a little bit more than readr or readxl. readr allows you to import *.csv files as well as other files in plain text. The functions included in readr are fairly straightforward, but there is an aspect that I really like about them: if they fail to read your data you can get a report of what went wrong with the reports() function. I suggest you read the Data import chapter of R for Data Science, to get to know readr better. But for our purposes, knowing the basic read_csv() function is enough.

readxl is very similar to readr but focuses on importing Excel sheets into R. Read more about it on the tidyverse website.

haven imports data from STATA, SAS and SPSS. I'm going into a bit more detail here, by showing an example with a STATA file. STATA files are usually labelled, and I'd like to show how to work with these labels using R. We're going to work with the mtcars dataset. I used STATA 14 to label the variables; so the dataset looks like one you could have to work with one day.

```
mtcars_stata <- haven::read_dta("assets/mtcars.dta")
head(mtcars_stata)
```

```
## # A tibble: 6 x 12
##                 car   mpg   cyl  disp    hp  drat    wt  qsec    vs    am
##               <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1         Mazda RX4  21.0     6   160   110  3.90 2.620 16.46     0     1
## 2     Mazda RX4 Wag  21.0     6   160   110  3.90 2.875 17.02     0     1
## 3        Datsun 710  22.8     4   108    93  3.85 2.320 18.61     1     1
## 4    Hornet 4 Drive  21.4     6   258   110  3.08 3.215 19.44     1     0
## 5 Hornet Sportabout  18.7     8   360   175  3.15 3.440 17.02     0     0
## 6           Valiant  18.1     6   225   105  2.76 3.460 20.22     1     0
## # ... with 2 more variables: gear <dbl>, carb <dbl>
```

You don't see it here, but the columns are labelled. Try the following:

```
str(mtcars_stata$car)
```

```
##  atomic [1:32] Mazda RX4 Mazda RX4 Wag Datsun 710 Hornet 4 Drive ...
##  - attr(*, "label")= chr "Make and model of the car"
##  - attr(*, "format.stata")= chr "%19s"
```

As you can see, the car column has the label attribute, which equals "Make and model of the car". The other columns are also labelled:

```
str(mtcars_stata$cyl)
```

```
##  atomic [1:32] 6 6 4 6 8 6 8 4 4 6 ...
##  - attr(*, "label")= chr "Number of cylinders"
##  - attr(*, "format.stata")= chr "%8.0g"
```

```
str(mtcars_stata$am)
```

```
##  atomic [1:32] 1 1 1 0 0 0 0 0 0 0 ...
##  - attr(*, "label")= chr "Transmission (0 = automatic, 1 = manual)"
##  - attr(*, "format.stata")= chr "%8.0g"
```

Another way to get the label is to use the `attr()` function:

```
attr(mtcars_stata$cyl, "label")
```

```
## [1] "Number of cylinders"
```

Let's use what we learned until now to get the labels of all the columns:

```
show_labels <- function(dataset){
  map(dataset, function(col)(attr(col, "label")))
}

show_labels(mtcars_stata)
```

```
## $car
## [1] "Make and model of the car"
##
## $mpg
## [1] "Miles/(US) gallon"
##
## $cyl
## [1] "Number of cylinders"
##
## $disp
## [1] "Displacement (cu.in.)"
##
## $hp
## [1] "Gross horsepower"
##
## $drat
## [1] "Rear axle ratio"
##
## $wt
## [1] "Weight (1000 lbs)"
##
## $qsec
## [1] "1/4 mile time"
##
## $vs
## [1] "V/S"
##
## $am
## [1] "Transmission (0 = automatic, 1 = manual)"
##
## $gear
## [1] "Number of forward gears"
##
## $carb
## [1] "Number of carburetors"
```

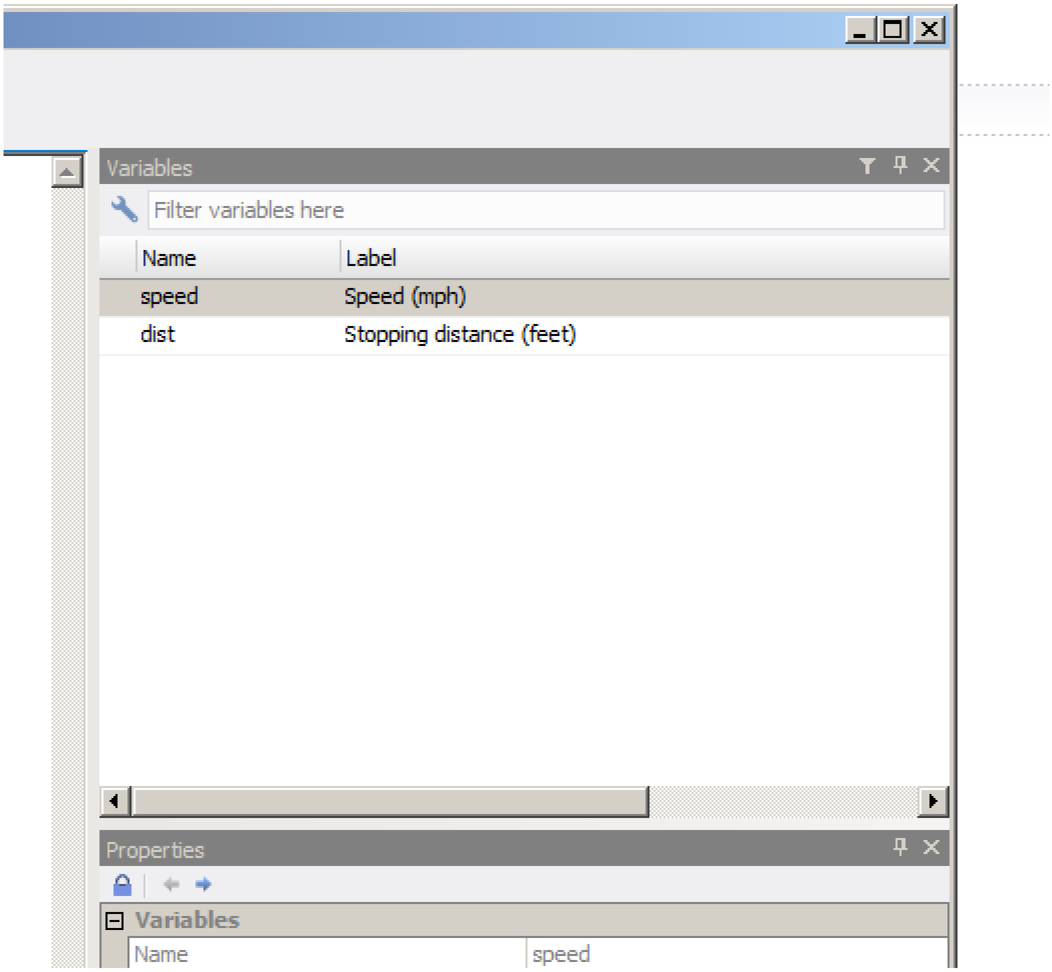
Could we label any dataset and then export it to a `.dta` file and have the labels in STATA? Let's find out with the `cars` dataset:

```
data(cars)

attr(cars$speed, "label") <- "Speed (mph)"
attr(cars$dist, "label") <- "Stopping distance (feet)"

haven::write_dta(cars, "assets/cars.dta")
```

Below you see that `cars.dta` file opened in STATA:



Name	Label
speed	Speed (mph)
dist	Stopping distance (feet)

Name	speed

When you use any of the discussed packages to import data, the resulting object is a `tibble`. `tibbles` are modern day 'data.frame's. The first thing you might have noticed is when you print a `tibble` vs a 'data.frame':

```r
data(mtcars)

print(mtcars)
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

```r
print(mtcars_stata)
```

```
## # A tibble: 32 x 12
##                   car   mpg   cyl  disp    hp  drat    wt  qsec    vs    am
##                 <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1          Mazda RX4  21.0     6 160.0   110  3.90 2.620 16.46     0     1
##  2      Mazda RX4 Wag  21.0     6 160.0   110  3.90 2.875 17.02     0     1
##  3         Datsun 710  22.8     4 108.0    93  3.85 2.320 18.61     1     1
##  4     Hornet 4 Drive  21.4     6 258.0   110  3.08 3.215 19.44     1     0
##  5  Hornet Sportabout  18.7     8 360.0   175  3.15 3.440 17.02     0     0
##  6            Valiant  18.1     6 225.0   105  2.76 3.460 20.22     1     0
##  7         Duster 360  14.3     8 360.0   245  3.21 3.570 15.84     0     0
##  8          Merc 240D  24.4     4 146.7    62  3.69 3.190 20.00     1     0
##  9           Merc 230  22.8     4 140.8    95  3.92 3.150 22.90     1     0
## 10          Merc 280  19.2     6 167.6   123  3.92 3.440 18.30      1     0
## # ... with 22 more rows, and 2 more variables: gear <dbl>, carb <dbl>
```

Only the first 10 lines of the `tibble` get printed, but the number of remaining lines and the names of the columns that didn't find are shown as well as the types of the columns.

You can easily create a `tibble` from vectors:

```
library(tibble)

set.seed(123)
example <- tibble(a = seq(1,5), b = rnorm(5), c = rpois(5, 3))

print(example)
```

```
## # A tibble: 5 x 3
##       a           b     c
##   <int>       <dbl> <int>
## 1     1 -0.56047565     6
## 2     2 -0.23017749     3
## 3     3  1.55870831     4
## 4     4  0.07050839     3
## 5     5  0.12928774     1
```

Even better than `print()`, there's `glimpse()`:

```
glimpse(mtcars_stata)
```

```
## Observations: 32
## Variables: 12
## $ car  <chr> "Mazda RX4", "Mazda RX4 Wag", "Datsun 710", "Hornet 4 Dri...
## $ mpg  <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19....
## $ cyl  <dbl> 6, 6, 4, 6, 8, 6, 8, 4, 4, 6, 6, 8, 8, 8, 8, 8, 8, 4, 4, ...
## $ disp <dbl> 160.0, 160.0, 108.0, 258.0, 360.0, 225.0, 360.0, 146.7, 1...
## $ hp   <dbl> 110, 110, 93, 110, 175, 105, 245, 62, 95, 123, 123, 180, ...
## $ drat <dbl> 3.90, 3.90, 3.85, 3.08, 3.15, 2.76, 3.21, 3.69, 3.92, 3.9...
## $ wt   <dbl> 2.620, 2.875, 2.320, 3.215, 3.440, 3.460, 3.570, 3.190, 3...
## $ qsec <dbl> 16.46, 17.02, 18.61, 19.44, 17.02, 20.22, 15.84, 20.00, 2...
## $ vs   <dbl> 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, ...
## $ am   <dbl> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, ...
## $ gear <dbl> 4, 4, 4, 3, 3, 3, 3, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 4, 4, ...
## $ carb <dbl> 4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4, 1, 2, ...
```

`tibble`s are lazy, which means that something like this is valid:

```
set.seed(123)
example <- tibble(a = seq(1,5), b = rnorm(5), c = 10 * b)

glimpse(example)
```

```
## Observations: 5
## Variables: 3
## $ a <int> 1, 2, 3, 4, 5
## $ b <dbl> -0.56047565, -0.23017749, 1.55870831, 0.07050839, 0.12928774
## $ c <dbl> -5.6047565, -2.3017749, 15.5870831, 0.7050839, 1.2928774
```

The `tibble` package contains some other useful functions, such as `tribble()`, which allows you to create a `tibble` row by row:

```
set.seed(123)

example <- tribble(
  ~a, ~b, ~c,
  1, 2, "spam",
  3, 4, "eggs",
  5, 6, "bacon"
)

glimpse(example)
```

```
## Observations: 3
## Variables: 3
## $ a <dbl> 1, 3, 5
## $ b <dbl> 2, 4, 6
## $ c <chr> "spam", "eggs", "bacon"
```

Another thing I find very useful is the following:

```
mtcars$m
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

```
mtcars_stata$m
```

```
## Warning: Unknown or uninitialised column: 'm'.
```

```
## NULL
```

`mtcars$m` shows the `mpg` column… for some reason. There might be a good reason for this, but I prefer `tibbles`' behaviour of notifying the user that this column does not exist.

It is possible to convert a lot of objects into `tibbles`:

```
example <- matrix(rnorm(36), nrow = 6)

as_tibble(example)
```

```
## # A tibble: 6 x 6
##            V1        V2        V3        V4         V5        V6
##         <dbl>     <dbl>     <dbl>     <dbl>      <dbl>     <dbl>
## 1 -0.56047565  0.4609162  0.4007715  0.7013559 -0.6250393  0.4264642
## 2 -0.23017749 -1.2650612  0.1106827 -0.4727914 -1.6866933 -0.2950715
## 3  1.55870831 -0.6868529 -0.5558411 -1.0678237  0.8377870  0.8951257
## 4  0.07050839 -0.4456620  1.7869131 -0.2179749  0.1533731  0.8781335
## 5  0.12928774  1.2240818  0.4978505 -1.0260044 -1.1381369  0.8215811
## 6  1.71506499  0.3598138 -1.9666172 -0.7288912  1.2538149  0.6886403
```

```
example_df <- as.data.frame(example)

as_tibble(example_df)
```

```
## # A tibble: 6 x 6
##            V1         V2         V3         V4         V5         V6
##         <dbl>      <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
## 1 -0.56047565  0.4609162  0.4007715  0.7013559 -0.6250393  0.4264642
## 2 -0.23017749 -1.2650612  0.1106827 -0.4727914 -1.6866933 -0.2950715
## 3  1.55870831 -0.6868529 -0.5558411 -1.0678237  0.8377870  0.8951257
## 4  0.07050839 -0.4456620  1.7869131 -0.2179749  0.1533731  0.8781335
## 5  0.12928774  1.2240818  0.4978505 -1.0260044 -1.1381369  0.8215811
## 6  1.71506499  0.3598138 -1.9666172 -0.7288912  1.2538149  0.6886403
```

```
example_list <- list(a = seq(1,5), b = seq(6, 10))

as_tibble(example_list)
```

```
## # A tibble: 5 x 2
##       a     b
##   <int> <int>
## 1     1     6
## 2     2     7
## 3     3     8
## 4     4     9
## 5     5    10
```

You can also convert named vectors to `tibble`s with enframe:

```
recipe <- c("spam" = 1, "eggs" = 3, "bacon" = 10)

enframe(recipe, "ingredients", "quantity")
```

```
## # A tibble: 3 x 2
##   ingredients quantity
##         <chr>    <dbl>
## 1        spam        1
## 2        eggs        3
## 3       bacon       10
```

Contrast this to `as_tibble()` or as `as.data.frame()`:

```
as.data.frame(recipe)
```

```
##       recipe
## spam       1
## eggs       3
## bacon     10
```

```
as_tibble(recipe)
```

```
## # A tibble: 3 x 1
##   value
## * <dbl>
## 1     1
## 2     3
## 3    10
```

There are a lot of other functions in the `tibble` package that you might find useful. I suggest you take a look at all of them and see what you can integrate in your workflow!

4.1.3 Transforming your data with `dplyr` and `tidyr`

You may have never heard of the `tidyverse`, but you most certainly heard about `dplyr` and `tidyr`. Both these packages are probably the most popular packages of the tidyverse. Even if you know these packages already, you might not be using some more advanced functions, I'm talking about the *scoped* version of the usual `dplyr` *verbs* (`dplyr` *verbs* is how Hadley Wickham refers to the functions included in the package: `group_by()`, `select()`, etc).

This is going to be long, so prepare some coffee, lock the door to your study, turn off your phone and buckle up.

4.1.3.1 `filter()` and friends

We're going to use the `Gasoline` dataset from the `plm` package, so install that first:

```
install.packages("plm")
```

Then load the required data:

```
data(Gasoline, package = "plm")
```

and load dplyr:

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:purrr':
##
##     contains, order_by
## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

This dataset gives the consumption of gasoline for 18 countries from 1960 to 1978. When you load the data like this, it is a standard `data.frame`. `dplyr` functions can be used on standard `data.frame` objects, but just because we learned about `tibble`'s, let's convert the data to a `tibble` and change its name:

```
gasoline <- as_tibble(Gasoline)
```

`filter()` is pretty straightforward. What if you would like to subset the data to focus on the year 1969? Simple:

```
filter(gasoline, year == 1969)
```

```
## # A tibble: 18 x 6
##     country  year lgaspcar  lincomep      lrpmg   lcarpcap
##      <fctr> <int>    <dbl>     <dbl>      <dbl>      <dbl>
##  1  AUSTRIA  1969 4.046355 -6.153140 -0.5591105  -8.788686
##  2  BELGIUM  1969 3.854601 -5.857532 -0.3548085  -8.521453
##  3   CANADA  1969 4.864433 -5.560853 -1.0368639  -8.095113
##  4  DENMARK  1969 4.173561 -5.722769 -0.4068792  -8.470459
##  5   FRANCE  1969 3.773460 -5.840774 -0.3151909  -8.369136
##  6  GERMANY  1969 3.899185 -5.829641 -0.5892314  -8.438061
##  7   GREECE  1969 4.894773 -6.591104 -0.1798700 -10.713848
##  8  IRELAND  1969 4.208613 -6.379743 -0.2716284  -8.947265
##  9    ITALY  1969 3.737389 -6.282857 -0.2475668  -8.666004
## 10    JAPAN  1969 4.518290 -6.159308 -0.4168502  -9.607600
## 11 NETHERLA  1969 3.987689 -5.880556 -0.4169496  -8.634102
## 12   NORWAY  1969 4.086823 -5.735319 -0.3382305  -8.694593
## 13    SPAIN  1969 3.994103 -5.601046  0.6694895  -9.720425
## 14   SWEDEN  1969 3.991715 -7.771081 -2.7319041  -8.197462
## 15 SWITZERL  1969 4.211290 -5.912172 -0.9181216  -8.473379
## 16   TURKEY  1969 5.720705 -7.388646 -0.2984542 -12.518545
## 17     U.K.  1969 3.948058 -6.031953 -0.3833246  -8.468119
## 18   U.S.A.  1969 4.841383 -5.414374 -1.2231427  -7.792706
```

Remember the pipe operator, `%>%` from the start of this chapter? Here's how this would work with it:

```
gasoline %>% filter(year == 1969)
```

```
## # A tibble: 18 x 6
##     country  year lgaspcar  lincomep      lrpmg   lcarpcap
##      <fctr> <int>    <dbl>     <dbl>      <dbl>      <dbl>
##  1  AUSTRIA  1969 4.046355 -6.153140 -0.5591105  -8.788686
##  2  BELGIUM  1969 3.854601 -5.857532 -0.3548085  -8.521453
##  3   CANADA  1969 4.864433 -5.560853 -1.0368639  -8.095113
##  4  DENMARK  1969 4.173561 -5.722769 -0.4068792  -8.470459
##  5   FRANCE  1969 3.773460 -5.840774 -0.3151909  -8.369136
##  6  GERMANY  1969 3.899185 -5.829641 -0.5892314  -8.438061
##  7   GREECE  1969 4.894773 -6.591104 -0.1798700 -10.713848
##  8  IRELAND  1969 4.208613 -6.379743 -0.2716284  -8.947265
##  9    ITALY  1969 3.737389 -6.282857 -0.2475668  -8.666004
## 10    JAPAN  1969 4.518290 -6.159308 -0.4168502  -9.607600
## 11 NETHERLA  1969 3.987689 -5.880556 -0.4169496  -8.634102
## 12   NORWAY  1969 4.086823 -5.735319 -0.3382305  -8.694593
## 13    SPAIN  1969 3.994103 -5.601046  0.6694895  -9.720425
## 14   SWEDEN  1969 3.991715 -7.771081 -2.7319041  -8.197462
## 15 SWITZERL  1969 4.211290 -5.912172 -0.9181216  -8.473379
## 16   TURKEY  1969 5.720705 -7.388646 -0.2984542 -12.518545
## 17     U.K.  1969 3.948058 -6.031953 -0.3833246  -8.468119
## 18   U.S.A.  1969 4.841383 -5.414374 -1.2231427  -7.792706
```

So `gasoline`, which is a `tibble` object, is passed as the first argument of the `filter()` function. Starting now, we're only going to use these pipes. You will see why soon enough, so bear with me.

You can also filter more than just one year, by using the `%in%` operator:

```
gasoline %>% filter(year %in% seq(1969, 1973))
```

```
## # A tibble: 90 x 6
##    country  year lgaspcar  lincomep      lrpmg  lcarpcap
##     <fctr> <int>    <dbl>     <dbl>      <dbl>     <dbl>
##  1 AUSTRIA  1969 4.046355 -6.153140 -0.5591105 -8.788686
##  2 AUSTRIA  1970 4.080888 -6.081712 -0.5965612 -8.728200
##  3 AUSTRIA  1971 4.106720 -6.043626 -0.6544591 -8.635898
##  4 AUSTRIA  1972 4.128018 -5.981052 -0.5963318 -8.538338
##  5 AUSTRIA  1973 4.199381 -5.895153 -0.5944468 -8.487289
##  6 BELGIUM  1969 3.854601 -5.857532 -0.3548085 -8.521453
##  7 BELGIUM  1970 3.870392 -5.797201 -0.3779404 -8.453043
##  8 BELGIUM  1971 3.872245 -5.761050 -0.3992299 -8.409457
##  9 BELGIUM  1972 3.905402 -5.710230 -0.3106458 -8.362588
## 10 BELGIUM  1973 3.895996 -5.644145 -0.3730919 -8.314447
## # ... with 80 more rows
```

or even non-consecutive years:

```
gasoline %>% filter(year %in% c(1969, 1973, 1977))
```

```
## # A tibble: 54 x 6
##    country  year lgaspcar  lincomep      lrpmg  lcarpcap
##     <fctr> <int>    <dbl>     <dbl>      <dbl>     <dbl>
##  1 AUSTRIA  1969 4.046355 -6.153140 -0.5591105 -8.788686
##  2 AUSTRIA  1973 4.199381 -5.895153 -0.5944468 -8.487289
##  3 AUSTRIA  1977 3.931676 -5.833288 -0.4219156 -8.249563
##  4 BELGIUM  1969 3.854601 -5.857532 -0.3548085 -8.521453
##  5 BELGIUM  1973 3.895996 -5.644145 -0.3730919 -8.314447
##  6 BELGIUM  1977 3.854311 -5.556697 -0.4316413 -8.138534
##  7  CANADA  1969 4.864433 -5.560853 -1.0368639 -8.095113
##  8  CANADA  1973 4.899694 -5.414753 -1.1331614 -7.942140
##  9  CANADA  1977 4.810992 -5.336967 -1.0708445 -7.768793
## 10 DENMARK  1969 4.173561 -5.722769 -0.4068792 -8.470459
## # ... with 44 more rows
```

`%in%` tests if an object is part of a set.

`filter()` is not the only *filtering* verb there is. Suppose that we have a condition that we want to use to filter out a lot of columns at once. For example, for every column that is of type `numeric`, keep only the lines where the condition *value > -8* is satisfied. The next line does that:

```
gasoline %>% filter_if( ~all(is.numeric(.)), all_vars(. > -8))
```

```
## # A tibble: 30 x 6
##    country  year lgaspcar  lincomep      lrpmg  lcarpcap
##     <fctr> <int>    <dbl>     <dbl>      <dbl>     <dbl>
##  1  CANADA  1972 4.889302 -5.436603 -1.0996670 -7.989531
##  2  CANADA  1973 4.899694 -5.414753 -1.1331614 -7.942140
##  3  CANADA  1974 4.891591 -5.418456 -1.1238000 -7.900758
##  4  CANADA  1975 4.888471 -5.379097 -1.1856843 -7.873313
##  5  CANADA  1976 4.837359 -5.361285 -1.0617966 -7.808425
##  6  CANADA  1977 4.810992 -5.336967 -1.0708445 -7.768793
##  7  CANADA  1978 4.855846 -5.311272 -1.0749507 -7.788061
##  8 GERMANY  1978 3.883879 -5.561733 -0.6281728 -7.950079
##  9  SWEDEN  1975 3.973840 -7.679557 -2.7673146 -7.994217
## 10  SWEDEN  1976 3.983997 -7.672043 -2.8229448 -7.956066
## # ... with 20 more rows
```

It's a bit more complicated than before. `filter_if()` needs 3 arguments to work; the data, a predicate function (a function that returns `TRUE`, or `FALSE`) which will select the columns we want to work on, and then the condition. The condition can be applied to *all* the columns that were selected by the predicate function (hence the `all_vars()`) or only to at least one (you'd use `any_vars()` then). Try to change the condition, or the predicate function, to figure out how `filter_if()` works. The dot is a placeholder that stands for whatever columns where selected.

`filter_at()` works differently; it allows the user to filter columns by position:

```
gasoline %>% filter_at(vars(ends_with("p")), all_vars(. > -8))
```

```
## # A tibble: 30 x 6
##    country  year lgaspcar  lincomep      lrpmg  lcarpcap
##     <fctr> <int>    <dbl>     <dbl>      <dbl>     <dbl>
##  1  CANADA  1972 4.889302 -5.436603 -1.0996670 -7.989531
##  2  CANADA  1973 4.899694 -5.414753 -1.1331614 -7.942140
##  3  CANADA  1974 4.891591 -5.418456 -1.1238000 -7.900758
##  4  CANADA  1975 4.888471 -5.379097 -1.1856843 -7.873313
##  5  CANADA  1976 4.837359 -5.361285 -1.0617966 -7.808425
##  6  CANADA  1977 4.810992 -5.336967 -1.0708445 -7.768793
##  7  CANADA  1978 4.855846 -5.311272 -1.0749507 -7.788061
##  8 GERMANY  1978 3.883879 -5.561733 -0.6281728 -7.950079
##  9  SWEDEN  1975 3.973840 -7.679557 -2.7673146 -7.994217
## 10  SWEDEN  1976 3.983997 -7.672043 -2.8229448 -7.956066
## # ... with 20 more rows
```

`end_with()` is a helper function that we are going to use a lot (as well as `starts_with()` and some others, you'll see..). So the above line means "for the columns whose name end with a 'p' only keep the lines where, for all the selected columns, the values are strictly superior to `-8`". Again, this is not very easy the first time you deal with that, so play around with it for a bit.

`filter_all()`, as the name implies, considers all variables for the filtering step.

`filter_if()` and `filter_at()` are very useful when you have very large datasets with a lot of variables and you want to apply a filtering function only to a subset of them. `filter_all()` is useful if, for example, you only want to keep the positive values for all the columns.

4.1.3.2 `select()` and its helpers

While `filter()` and its scoped versions allow you to keep or discard rows of data, `select()` (and its scoped versions) allow you to keep or discard entire columns. To keep columns:

```
gasoline %>% select(country, year, lrpmg)
```

```
## # A tibble: 342 x 3
##    country  year      lrpmg
##  *  <fctr> <int>      <dbl>
##  1 AUSTRIA  1960 -0.3345476
##  2 AUSTRIA  1961 -0.3513276
##  3 AUSTRIA  1962 -0.3795177
##  4 AUSTRIA  1963 -0.4142514
##  5 AUSTRIA  1964 -0.4453354
##  6 AUSTRIA  1965 -0.4970607
##  7 AUSTRIA  1966 -0.4668377
##  8 AUSTRIA  1967 -0.5058834
##  9 AUSTRIA  1968 -0.5224125
## 10 AUSTRIA  1969 -0.5591105
## # ... with 332 more rows
```

To discard them:

```
gasoline %>% select(-country, -year, -lrpmg)
```

```
## # A tibble: 342 x 3
##    lgaspcar  lincomep  lcarpcap
##  *    <dbl>     <dbl>     <dbl>
##  1 4.173244 -6.474277 -9.766840
##  2 4.100989 -6.426006 -9.608622
##  3 4.073177 -6.407308 -9.457257
##  4 4.059509 -6.370679 -9.343155
##  5 4.037689 -6.322247 -9.237739
##  6 4.033983 -6.294668 -9.123903
##  7 4.047537 -6.252545 -9.019822
##  8 4.052911 -6.234581 -8.934403
##  9 4.045507 -6.206894 -8.847967
## 10 4.046355 -6.153140 -8.788686
## # ... with 332 more rows
```

To rename them:

```
gasoline %>% select(country, date = year, lrpmg)
```

```
## # A tibble: 342 x 3
##    country  date      lrpmg
##  *  <fctr> <int>      <dbl>
##  1 AUSTRIA  1960 -0.3345476
##  2 AUSTRIA  1961 -0.3513276
##  3 AUSTRIA  1962 -0.3795177
##  4 AUSTRIA  1963 -0.4142514
##  5 AUSTRIA  1964 -0.4453354
##  6 AUSTRIA  1965 -0.4970607
##  7 AUSTRIA  1966 -0.4668377
##  8 AUSTRIA  1967 -0.5058834
##  9 AUSTRIA  1968 -0.5224125
## 10 AUSTRIA  1969 -0.5591105
## # ... with 332 more rows
```

There's also `rename()`, but it works a bit differently:

```
gasoline %>% rename(date = year)
```

```
## # A tibble: 342 x 6
##    country  date lgaspcar  lincomep      lrpmg  lcarpcap
##  *  <fctr> <int>    <dbl>     <dbl>      <dbl>     <dbl>
##  1 AUSTRIA  1960 4.173244 -6.474277 -0.3345476 -9.766840
##  2 AUSTRIA  1961 4.100989 -6.426006 -0.3513276 -9.608622
##  3 AUSTRIA  1962 4.073177 -6.407308 -0.3795177 -9.457257
##  4 AUSTRIA  1963 4.059509 -6.370679 -0.4142514 -9.343155
##  5 AUSTRIA  1964 4.037689 -6.322247 -0.4453354 -9.237739
##  6 AUSTRIA  1965 4.033983 -6.294668 -0.4970607 -9.123903
##  7 AUSTRIA  1966 4.047537 -6.252545 -0.4668377 -9.019822
##  8 AUSTRIA  1967 4.052911 -6.234581 -0.5058834 -8.934403
##  9 AUSTRIA  1968 4.045507 -6.206894 -0.5224125 -8.847967
## 10 AUSTRIA  1969 4.046355 -6.153140 -0.5591105 -8.788686
## # ... with 332 more rows
```

`rename()` does not do any kind of selection, but just renames.

To re-order them:

```
gasoline %>% select(year, country, lrpmg, everything())
```

```
## # A tibble: 342 x 6
##     year country      lrpmg lgaspcar  lincomep  lcarpcap
##  * <int>  <fctr>      <dbl>    <dbl>     <dbl>     <dbl>
##  1  1960 AUSTRIA -0.3345476 4.173244 -6.474277 -9.766840
##  2  1961 AUSTRIA -0.3513276 4.100989 -6.426006 -9.608622
##  3  1962 AUSTRIA -0.3795177 4.073177 -6.407308 -9.457257
##  4  1963 AUSTRIA -0.4142514 4.059509 -6.370679 -9.343155
##  5  1964 AUSTRIA -0.4453354 4.037689 -6.322247 -9.237739
##  6  1965 AUSTRIA -0.4970607 4.033983 -6.294668 -9.123903
##  7  1966 AUSTRIA -0.4668377 4.047537 -6.252545 -9.019822
## 8  1967 AUSTRIA -0.5058834 4.052911 -6.234581 -8.934403
## 9  1968 AUSTRIA -0.5224125 4.045507 -6.206894 -8.847967
## 10  1969 AUSTRIA -0.5591105 4.046355 -6.153140 -8.788686
## # ... with 332 more rows
```

`everything()` is another of those helper functions (like `starts_with()`, and `ends_with()`). What if we are only interested in columns whose name start with "l"?

```
gasoline %>% select(starts_with("l"))
```

```
## # A tibble: 342 x 4
##    lgaspcar  lincomep      lrpmg  lcarpcap
##  *    <dbl>     <dbl>      <dbl>     <dbl>
##  1 4.173244 -6.474277 -0.3345476 -9.766840
##  2 4.100989 -6.426006 -0.3513276 -9.608622
##  3 4.073177 -6.407308 -0.3795177 -9.457257
##  4 4.059509 -6.370679 -0.4142514 -9.343155
##  5 4.037689 -6.322247 -0.4453354 -9.237739
##  6 4.033983 -6.294668 -0.4970607 -9.123903
##  7 4.047537 -6.252545 -0.4668377 -9.019822
##  8 4.052911 -6.234581 -0.5058834 -8.934403
##  9 4.045507 -6.206894 -0.5224125 -8.847967
## 10 4.046355 -6.153140 -0.5591105 -8.788686
## # ... with 332 more rows
```

The same can be achieved with `select_at()`:

```
gasoline %>% select_at(vars(starts_with("l")))
```

```
## # A tibble: 342 x 4
##    lgaspcar  lincomep      lrpmg  lcarpcap
##  *    <dbl>     <dbl>      <dbl>     <dbl>
##  1 4.173244 -6.474277 -0.3345476 -9.766840
##  2 4.100989 -6.426006 -0.3513276 -9.608622
##  3 4.073177 -6.407308 -0.3795177 -9.457257
##  4 4.059509 -6.370679 -0.4142514 -9.343155
##  5 4.037689 -6.322247 -0.4453354 -9.237739
##  6 4.033983 -6.294668 -0.4970607 -9.123903
##  7 4.047537 -6.252545 -0.4668377 -9.019822
##  8 4.052911 -6.234581 -0.5058834 -8.934403
##  9 4.045507 -6.206894 -0.5224125 -8.847967
## 10 4.046355 -6.153140 -0.5591105 -8.788686
## # ... with 332 more rows
```

`select_at()` can be quite useful if you know the position of the columns you're interested in:

```
gasoline %>% select_at(vars(c(1,2,5)))
```

```
## # A tibble: 342 x 3
##    country  year      lrpmg
##  *  <fctr> <int>      <dbl>
##  1 AUSTRIA  1960 -0.3345476
##  2 AUSTRIA  1961 -0.3513276
##  3 AUSTRIA  1962 -0.3795177
##  4 AUSTRIA  1963 -0.4142514
##  5 AUSTRIA  1964 -0.4453354
##  6 AUSTRIA  1965 -0.4970607
##  7 AUSTRIA  1966 -0.4668377
##  8 AUSTRIA  1967 -0.5058834
##  9 AUSTRIA  1968 -0.5224125
## 10 AUSTRIA  1969 -0.5591105
## # ... with 332 more rows
```

This also works with `filter_at()` by the way.

`select_if()` makes it easy to select columns that satisfy a criterium:

```
gasoline %>% select_if(is.numeric)
```

```
## # A tibble: 342 x 5
##     year lgaspcar  lincomep      lrpmg  lcarpcap
##  * <int>    <dbl>     <dbl>      <dbl>     <dbl>
##  1  1960 4.173244 -6.474277 -0.3345476 -9.766840
##  2  1961 4.100989 -6.426006 -0.3513276 -9.608622
##  3  1962 4.073177 -6.407308 -0.3795177 -9.457257
##  4  1963 4.059509 -6.370679 -0.4142514 -9.343155
##  5  1964 4.037689 -6.322247 -0.4453354 -9.237739
##  6  1965 4.033983 -6.294668 -0.4970607 -9.123903
##  7  1966 4.047537 -6.252545 -0.4668377 -9.019822
##  8  1967 4.052911 -6.234581 -0.5058834 -8.934403
##  9  1968 4.045507 -6.206894 -0.5224125 -8.847967
## 10  1969 4.046355 -6.153140 -0.5591105 -8.788686
## # ... with 332 more rows
```

You can even pass a further function to `select_if()` that will be applied to the selected columns:

```
gasoline %>% select_if(is.numeric, toupper)
```

```
## # A tibble: 342 x 5
##     YEAR LGASPCAR  LINCOMEP      LRPMG  LCARPCAP
##  * <int>    <dbl>     <dbl>      <dbl>     <dbl>
##  1  1960 4.173244 -6.474277 -0.3345476 -9.766840
##  2  1961 4.100989 -6.426006 -0.3513276 -9.608622
##  3  1962 4.073177 -6.407308 -0.3795177 -9.457257
##  4  1963 4.059509 -6.370679 -0.4142514 -9.343155
##  5  1964 4.037689 -6.322247 -0.4453354 -9.237739
##  6  1965 4.033983 -6.294668 -0.4970607 -9.123903
##  7  1966 4.047537 -6.252545 -0.4668377 -9.019822
##  8  1967 4.052911 -6.234581 -0.5058834 -8.934403
##  9  1968 4.045507 -6.206894 -0.5224125 -8.847967
## 10  1969 4.046355 -6.153140 -0.5591105 -8.788686
## # ... with 332 more rows
```

4.1.3.3 `group_by()`

`group_by()` is a very useful verb; as the name implies, it allows you to create groups and then, for example, compute descriptive statistics by groups. For example, let's group our data by country:

```
gasoline %>% group_by(country)
```

```
## # A tibble: 342 x 6
## # Groups:   country [18]
##    country  year lgaspcar  lincomep      lrpmg  lcarpcap
##  *  <fctr> <int>    <dbl>     <dbl>      <dbl>     <dbl>
##  1 AUSTRIA  1960 4.173244 -6.474277 -0.3345476 -9.766840
##  2 AUSTRIA  1961 4.100989 -6.426006 -0.3513276 -9.608622
##  3 AUSTRIA  1962 4.073177 -6.407308 -0.3795177 -9.457257
##  4 AUSTRIA  1963 4.059509 -6.370679 -0.4142514 -9.343155
##  5 AUSTRIA  1964 4.037689 -6.322247 -0.4453354 -9.237739
##  6 AUSTRIA  1965 4.033983 -6.294668 -0.4970607 -9.123903
##  7 AUSTRIA  1966 4.047537 -6.252545 -0.4668377 -9.019822
##  8 AUSTRIA  1967 4.052911 -6.234581 -0.5058834 -8.934403
##  9 AUSTRIA  1968 4.045507 -6.206894 -0.5224125 -8.847967
## 10 AUSTRIA  1969 4.046355 -6.153140 -0.5591105 -8.788686
## # ... with 332 more rows
```

It looks like nothing much happened, but if you look at the second line of the output you can read the followi`{r} ## # Groups:   country [18]`

this means that the data is grouped, and every computation you will do now will take these groups into account. This will be clearer in the next subsection.

It is also possible to group according to various variables:

```
gasoline %>% group_by(country, year)
```

```
## # A tibble: 342 x 6
## # Groups:   country, year [342]
##    country  year lgaspcar  lincomep      lrpmg  lcarpcap
##  *  <fctr> <int>    <dbl>     <dbl>      <dbl>     <dbl>
##  1 AUSTRIA  1960 4.173244 -6.474277 -0.3345476 -9.766840
##  2 AUSTRIA  1961 4.100989 -6.426006 -0.3513276 -9.608622
##  3 AUSTRIA  1962 4.073177 -6.407308 -0.3795177 -9.457257
##  4 AUSTRIA  1963 4.059509 -6.370679 -0.4142514 -9.343155
##  5 AUSTRIA  1964 4.037689 -6.322247 -0.4453354 -9.237739
##  6 AUSTRIA  1965 4.033983 -6.294668 -0.4970607 -9.123903
##  7 AUSTRIA  1966 4.047537 -6.252545 -0.4668377 -9.019822
##  8 AUSTRIA  1967 4.052911 -6.234581 -0.5058834 -8.934403
##  9 AUSTRIA  1968 4.045507 -6.206894 -0.5224125 -8.847967
## 10 AUSTRIA  1969 4.046355 -6.153140 -0.5591105 -8.788686
## # ... with 332 more rows
```

and so on. You can then also ungroup:

```
gasoline %>% group_by(country, year) %>% ungroup()
```

```
## # A tibble: 342 x 6
##    country  year lgaspcar  lincomep      lrpmg  lcarpcap
##  *  <fctr> <int>    <dbl>     <dbl>      <dbl>     <dbl>
##  1 AUSTRIA  1960 4.173244 -6.474277 -0.3345476 -9.766840
##  2 AUSTRIA  1961 4.100989 -6.426006 -0.3513276 -9.608622
##  3 AUSTRIA  1962 4.073177 -6.407308 -0.3795177 -9.457257
##  4 AUSTRIA  1963 4.059509 -6.370679 -0.4142514 -9.343155
##  5 AUSTRIA  1964 4.037689 -6.322247 -0.4453354 -9.237739
##  6 AUSTRIA  1965 4.033983 -6.294668 -0.4970607 -9.123903
##  7 AUSTRIA  1966 4.047537 -6.252545 -0.4668377 -9.019822
##  8 AUSTRIA  1967 4.052911 -6.234581 -0.5058834 -8.934403
##  9 AUSTRIA  1968 4.045507 -6.206894 -0.5224125 -8.847967
## 10 AUSTRIA  1969 4.046355 -6.153140 -0.5591105 -8.788686
## # ... with 332 more rows
```

4.1.3.4 summarise()

Ok, now that we have learned the basic verbs, we can start to do more interesting stuff. For example, one might want to compute the average gasoline consumption per car in each country, for the whole period:

```
gasoline %>%
  group_by(country) %>%
  summarise(mean(lgaspcar))
```

```
## # A tibble: 18 x 2
##     country `mean(lgaspcar)`
##      <fctr>            <dbl>
##  1  AUSTRIA         4.056487
##  2  BELGIUM         3.922286
##  3   CANADA         4.862402
##  4  DENMARK         4.189886
##  5   FRANCE         3.815198
##  6  GERMANY         3.893389
##  7   GREECE         4.878679
##  8  IRELAND         4.225560
##  9    ITALY         3.729646
## 10    JAPAN         4.699642
## 11 NETHERLA         4.080338
## 12   NORWAY         4.109773
## 13    SPAIN         4.055314
## 14   SWEDEN         4.006055
## 15 SWITZERL         4.237586
## 16   TURKEY         5.766355
## 17     U.K.         3.984685
## 18   U.S.A.         4.819075
```

`mean()` was given as an argument to `summarise()`, which is a `dplyr` verb. What we get is another tibble, that contains the variable we used to group, as well as the average per country. We can also rename this column:

```
gasoline %>%
  group_by(country) %>%
  summarise(mean_gaspcar = mean(lgaspcar))
```

```
## # A tibble: 18 x 2
##     country mean_gaspcar
##      <fctr>        <dbl>
##  1  AUSTRIA     4.056487
##  2  BELGIUM     3.922286
##  3   CANADA     4.862402
##  4  DENMARK     4.189886
##  5   FRANCE     3.815198
##  6  GERMANY     3.893389
##  7   GREECE     4.878679
##  8  IRELAND     4.225560
##  9    ITALY     3.729646
## 10    JAPAN     4.699642
## 11 NETHERLA     4.080338
## 12   NORWAY     4.109773
## 13    SPAIN     4.055314
## 14   SWEDEN     4.006055
## 15 SWITZERL     4.237586
## 16   TURKEY     5.766355
## 17     U.K.     3.984685
## 18   U.S.A.     4.819075
```

and because the output is a `tibble`, we can contain to use `dplyr` verbs on it:

```
gasoline %>%
  group_by(country) %>%
  summarise(mean_gaspcar = mean(lgaspcar)) %>%
  filter(country == "FRANCE")
```

```
## # A tibble: 1 x 2
##   country mean_gaspcar
##    <fctr>        <dbl>
## 1  FRANCE     3.815198
```

Ok, let's pause here. See what I did in the last example? I chained 3 `dplyr` verbs together with `%>%`. Without using `%>%` I would have written:

```
filter(
  summarise(
    group_by(gasoline, country),
    mean_gaspcar = mean(lgaspcar)),
  country == "FRANCE")
```

```
## # A tibble: 1 x 2
##   country mean_gaspcar
##    <fctr>        <dbl>
## 1  FRANCE     3.815198
```

I don't know about you, but this is much more difficult to read than the version with %>%. It is possible to work like that, of course, but personally, I would advise you bite the bullet and learn to love the pipe. It won't give you cancer.

Ok, back to summarise(). We can really do a lot of stuff with this verb. For example, we can compute several descriptive statistics at once:

```
gasoline %>%
  group_by(country) %>%
  summarise(mean_gaspcar = mean(lgaspcar), sd_gaspcar = sd(lgaspcar), max_gaspcar = max(lgaspcar), min_g
```

```
## # A tibble: 18 x 5
##     country mean_gaspcar sd_gaspcar max_gaspcar min_gaspcar
##      <fctr>        <dbl>      <dbl>       <dbl>       <dbl>
##  1  AUSTRIA     4.056487 0.06929942    4.199381    3.922750
##  2  BELGIUM     3.922286 0.10339189    4.164016    3.818230
##  3   CANADA     4.862402 0.02618377    4.899694    4.810992
##  4  DENMARK     4.189886 0.15819728    4.501986    4.000461
##  5   FRANCE     3.815198 0.04986425    3.908116    3.749535
##  6  GERMANY     3.893389 0.02389849    3.932402    3.848782
##  7   GREECE     4.878679 0.25467445    5.381495    4.479956
##  8  IRELAND     4.225560 0.04369894    4.325585    4.164896
##  9    ITALY     3.729646 0.22001527    4.050728    3.380209
## 10    JAPAN     4.699642 0.68411717    5.995287    3.948746
## 11 NETHERLA     4.080338 0.28642682    4.646268    3.711384
## 12   NORWAY     4.109773 0.12306866    4.435041    3.960331
## 13    SPAIN     4.055314 0.31696784    4.749409    3.620444
## 14   SWEDEN     4.006055 0.03639626    4.067373    3.913159
## 15 SWITZERL     4.237586 0.10178743    4.441330    4.050048
## 16   TURKEY     5.766355 0.32901391    6.156644    5.141255
## 17     U.K.     3.984685 0.04787887    4.100244    3.912584
## 18   U.S.A.     4.819075 0.02189802    4.860286    4.787895
```

Because the output is a tibble, you can save it in a variable of course:

```
desc_gasoline <- gasoline %>%
  group_by(country) %>%
  summarise(mean_gaspcar = mean(lgaspcar), sd_gaspcar = sd(lgaspcar), max_gaspcar = max(lgaspcar), min_
```

And then you can answer questions such as, *which country has the maximum average gasoline consumption per car?*:

```
desc_gasoline %>%
  filter(max(mean_gaspcar) == mean_gaspcar)
```

```
## # A tibble: 1 x 5
##   country mean_gaspcar sd_gaspcar max_gaspcar min_gaspcar
##    <fctr>        <dbl>      <dbl>       <dbl>       <dbl>
## 1  TURKEY     5.766355  0.3290139    6.156644    5.141255
```

Turns out it's Turkey. What about the minimum consumption?

```
desc_gasoline %>%
  filter(min(mean_gaspcar) == mean_gaspcar)
```

```
## # A tibble: 1 x 5
##   country mean_gaspcar sd_gaspcar max_gaspcar min_gaspcar
##    <fctr>        <dbl>      <dbl>       <dbl>       <dbl>
## 1   ITALY     3.729646  0.2200153    4.050728    3.380209
```

Just like for `filter()` and `select()`, `summarise()` comes with scoped versions:

```
gasoline %>%
  group_by(country) %>%
  summarise_at(vars(starts_with("l")), mean)
```

```
## # A tibble: 18 x 5
##     country lgaspcar  lincomep       lrpmg   lcarpcap
##      <fctr>    <dbl>     <dbl>       <dbl>      <dbl>
##  1  AUSTRIA 4.056487 -6.119613 -0.48578185  -8.848114
##  2  BELGIUM 3.922286 -5.852297 -0.32575856  -8.630392
##  3   CANADA 4.862402 -5.576948 -1.04918735  -8.081975
##  4  DENMARK 4.189886 -5.756725 -0.35761243  -8.583795
##  5   FRANCE 3.815198 -5.866167 -0.25277821  -8.452957
##  6  GERMANY 3.893389 -5.845314 -0.51718417  -8.506392
##  7   GREECE 4.878679 -6.606373 -0.03391043 -10.782066
##  8  IRELAND 4.225560 -6.441571 -0.34754288  -9.035927
##  9    ITALY 3.729646 -6.350354 -0.15219273  -8.827179
## 10    JAPAN 4.699642 -6.248629 -0.28662755  -9.945087
## 11 NETHERLA 4.080338 -5.920132 -0.36977928  -8.817087
## 12   NORWAY 4.109773 -5.753316 -0.27767861  -8.765066
## 13    SPAIN 4.055314 -5.627756  0.73937888  -9.896247
## 14   SWEDEN 4.006055 -7.816214 -2.70917074  -8.250729
## 15 SWITZERL 4.237586 -5.927320 -0.90165998  -8.541029
## 16   TURKEY 5.766355 -7.336992 -0.42151399 -12.458858
## 17     U.K. 3.984685 -6.015377 -0.45929339  -8.548493
## 18   U.S.A. 4.819075 -5.448560 -1.20756453  -7.781090
```

See how I managed to summarise every variable in one simple call to `summarise_at()`? Simply by using `vars()` and specifying that I was interested in the ones that started with "l" and then I specified the function I wanted. But what if I wanted to use more than one function to summarise the data? Very easy:

```
gasoline %>%
  group_by(country) %>%
  summarise_at(vars(starts_with("l")), funs(mean, sd, max, min))
```

```
## # A tibble: 18 x 17
##     country lgaspcar_mean lincomep_mean  lrpmg_mean lcarpcap_mean
##      <fctr>         <dbl>         <dbl>       <dbl>         <dbl>
##  1  AUSTRIA      4.056487     -6.119613 -0.48578185     -8.848114
##  2  BELGIUM      3.922286     -5.852297 -0.32575856     -8.630392
##  3   CANADA      4.862402     -5.576948 -1.04918735     -8.081975
##  4  DENMARK      4.189886     -5.756725 -0.35761243     -8.583795
##  5   FRANCE      3.815198     -5.866167 -0.25277821     -8.452957
##  6  GERMANY      3.893389     -5.845314 -0.51718417     -8.506392
##  7   GREECE      4.878679     -6.606373 -0.03391043    -10.782066
##  8  IRELAND      4.225560     -6.441571 -0.34754288     -9.035927
##  9    ITALY      3.729646     -6.350354 -0.15219273     -8.827179
## 10    JAPAN      4.699642     -6.248629 -0.28662755     -9.945087
## 11 NETHERLA      4.080338     -5.920132 -0.36977928     -8.817087
## 12   NORWAY      4.109773     -5.753316 -0.27767861     -8.765066
## 13    SPAIN      4.055314     -5.627756  0.73937888     -9.896247
## 14   SWEDEN      4.006055     -7.816214 -2.70917074     -8.250729
## 15 SWITZERL      4.237586     -5.927320 -0.90165998     -8.541029
## 16   TURKEY      5.766355     -7.336992 -0.42151399    -12.458858
## 17     U.K.      3.984685     -6.015377 -0.45929339     -8.548493
## 18   U.S.A.      4.819075     -5.448560 -1.20756453     -7.781090
## # ... with 12 more variables: lgaspcar_sd <dbl>, lincomep_sd <dbl>,
## #   lrpmg_sd <dbl>, lcarpcap_sd <dbl>, lgaspcar_max <dbl>,
## #   lincomep_max <dbl>, lrpmg_max <dbl>, lcarpcap_max <dbl>,
## #   lgaspcar_min <dbl>, lincomep_min <dbl>, lrpmg_min <dbl>,
## #   lcarpcap_min <dbl>
```

But maybe you're just interested in descriptive statistics for some variables, but not all those that start with "l"? What if you want to use another pattern? Easy to do with the `contains()` helper:

```r
gasoline %>%
  group_by(country) %>%
  summarise_at(vars(contains("car")), funs(mean, sd, max, min))
```

```
## # A tibble: 18 x 9
##     country lgaspcar_mean lcarpcap_mean lgaspcar_sd lcarpcap_sd
##      <fctr>         <dbl>         <dbl>       <dbl>       <dbl>
##  1  AUSTRIA      4.056487     -8.848114  0.06929942   0.4728231
##  2  BELGIUM      3.922286     -8.630392  0.10339189   0.4171514
##  3   CANADA      4.862402     -8.081975  0.02618377   0.1953069
##  4  DENMARK      4.189886     -8.583795  0.15819728   0.3486135
##  5   FRANCE      3.815198     -8.452957  0.04986425   0.3436969
##  6  GERMANY      3.893389     -8.506392  0.02389849   0.4060370
##  7   GREECE      4.878679    -10.782066  0.25467445   0.8388589
##  8  IRELAND      4.225560     -9.035927  0.04369894   0.3452272
##  9    ITALY      3.729646     -8.827179  0.22001527   0.6389769
## 10    JAPAN      4.699642     -9.945087  0.68411717   1.1969275
## 11 NETHERLA      4.080338     -8.817087  0.28642682   0.6173209
## 12   NORWAY      4.109773     -8.765066  0.12306866   0.4382484
## 13    SPAIN      4.055314     -9.896247  0.31696784   0.9596034
## 14   SWEDEN      4.006055     -8.250729  0.03639626   0.2422792
## 15 SWITZERL      4.237586     -8.541029  0.10178743   0.3775211
## 16   TURKEY      5.766355    -12.458858  0.32901391   0.7512506
## 17     U.K.      3.984685     -8.548493  0.04787887   0.2812851
## 18   U.S.A.      4.819075     -7.781090  0.02189802   0.1617998
## # ... with 4 more variables: lgaspcar_max <dbl>, lcarpcap_max <dbl>,
## #   lgaspcar_min <dbl>, lcarpcap_min <dbl>
```

There's also `summarise_if()`:

```
gasoline %>%
  group_by(country) %>%
  summarise_if(is.double, funs(mean, sd, min, max))
```

```
## # A tibble: 18 x 17
##     country lgaspcar_mean lincomep_mean  lrpmg_mean lcarpcap_mean
##      <fctr>         <dbl>         <dbl>       <dbl>         <dbl>
##  1  AUSTRIA      4.056487     -6.119613 -0.48578185     -8.848114
##  2  BELGIUM      3.922286     -5.852297 -0.32575856     -8.630392
##  3   CANADA      4.862402     -5.576948 -1.04918735     -8.081975
##  4  DENMARK      4.189886     -5.756725 -0.35761243     -8.583795
##  5   FRANCE      3.815198     -5.866167 -0.25277821     -8.452957
##  6  GERMANY      3.893389     -5.845314 -0.51718417     -8.506392
##  7   GREECE      4.878679     -6.606373 -0.03391043    -10.782066
##  8  IRELAND      4.225560     -6.441571 -0.34754288     -9.035927
##  9    ITALY      3.729646     -6.350354 -0.15219273     -8.827179
## 10    JAPAN      4.699642     -6.248629 -0.28662755     -9.945087
## 11 NETHERLA      4.080338     -5.920132 -0.36977928     -8.817087
## 12   NORWAY      4.109773     -5.753316 -0.27767861     -8.765066
## 13    SPAIN      4.055314     -5.627756  0.73937888     -9.896247
## 14   SWEDEN      4.006055     -7.816214 -2.70917074     -8.250729
## 15 SWITZERL      4.237586     -5.927320 -0.90165998     -8.541029
## 16   TURKEY      5.766355     -7.336992 -0.42151399    -12.458858
## 17     U.K.      3.984685     -6.015377 -0.45929339     -8.548493
## 18   U.S.A.      4.819075     -5.448560 -1.20756453     -7.781090
## # ... with 12 more variables: lgaspcar_sd <dbl>, lincomep_sd <dbl>,
## #   lrpmg_sd <dbl>, lcarpcap_sd <dbl>, lgaspcar_min <dbl>,
## #   lincomep_min <dbl>, lrpmg_min <dbl>, lcarpcap_min <dbl>,
## #   lgaspcar_max <dbl>, lincomep_max <dbl>, lrpmg_max <dbl>,
## #   lcarpcap_max <dbl>
```

This allows you to summarise every column that contain real numbers (if you use `is.numeric()` instead, `year` will also be summarised, which is not really interesting).

To go faster, you can also use `summarise_all()`:

```
gasoline %>%
  select(-year) %>%
  group_by(country) %>%
  summarise_all(funs(mean, sd, min, max))
```

```
## # A tibble: 18 x 17
##     country lgaspcar_mean lincomep_mean  lrpmg_mean lcarpcap_mean
##      <fctr>         <dbl>         <dbl>       <dbl>         <dbl>
##  1  AUSTRIA      4.056487     -6.119613 -0.48578185     -8.848114
##  2  BELGIUM      3.922286     -5.852297 -0.32575856     -8.630392
##  3   CANADA      4.862402     -5.576948 -1.04918735     -8.081975
##  4  DENMARK      4.189886     -5.756725 -0.35761243     -8.583795
##  5   FRANCE      3.815198     -5.866167 -0.25277821     -8.452957
##  6  GERMANY      3.893389     -5.845314 -0.51718417     -8.506392
##  7   GREECE      4.878679     -6.606373 -0.03391043    -10.782066
##  8  IRELAND      4.225560     -6.441571 -0.34754288     -9.035927
##  9    ITALY      3.729646     -6.350354 -0.15219273     -8.827179
## 10    JAPAN      4.699642     -6.248629 -0.28662755     -9.945087
## 11 NETHERLA      4.080338     -5.920132 -0.36977928     -8.817087
## 12   NORWAY      4.109773     -5.753316 -0.27767861     -8.765066
## 13    SPAIN      4.055314     -5.627756  0.73937888     -9.896247
## 14   SWEDEN      4.006055     -7.816214 -2.70917074     -8.250729
## 15 SWITZERL      4.237586     -5.927320 -0.90165998     -8.541029
## 16   TURKEY      5.766355     -7.336992 -0.42151399    -12.458858
## 17     U.K.      3.984685     -6.015377 -0.45929339     -8.548493
## 18   U.S.A.      4.819075     -5.448560 -1.20756453     -7.781090
## # ... with 12 more variables: lgaspcar_sd <dbl>, lincomep_sd <dbl>,
## #   lrpmg_sd <dbl>, lcarpcap_sd <dbl>, lgaspcar_min <dbl>,
## #   lincomep_min <dbl>, lrpmg_min <dbl>, lcarpcap_min <dbl>,
## #   lgaspcar_max <dbl>, lincomep_max <dbl>, lrpmg_max <dbl>,
## #   lcarpcap_max <dbl>
```

I removed the year variable because it's not a variable for which we want to have descriptive statistics.

4.1.3.5 mutate() and transmute()

4.1.3.6 case_when() and if_else()

4.1.3.7 coalesce()

4.1.3.8 arrange()

4.1.3.9 sample_n() and sample_frac()

4.1.3.10 Joining tibbles with full_join(), left_join(), right_join() and all the others

4.1.4 Functional programming with purrr and purrrlyr

4.1.5 Special packages for special kinds of data: forcats, lubridate, and stringr

4.2 Programming with the tidyverse

4.2.1 The naive approach

Functions are very powerful because by using them, we avoid repetition. This means that we must be able to write functions that allow the user to abstract over certain things, such as columns names of datasets. So for example, one would like to write a function that would look like that:

```
my_function(my_data, column)
```

and in this chapter we will learn together how to do that using dplyr (version 0.70 or above).

I advise you to also read the "Programming with dplyr" vignette here, which explains with great detail the concept I will only skim in this chapter!

Consider the following code:

```
data(mtcars)
simple_function <- function(dataset, col_name){
  dataset %>%
    group_by(col_name) %>%
    summarise(mean_mpg = mean(mpg)) -> dataset
  return(dataset)
}
```

When you try to run this:

```
simple_function(mtcars, "cyl")
```

This is the error you get:

```
 Error in grouped_df_impl(data, unname(vars), drop) :
  Column `col_name` is unknown
```

R is *literally* looking for a column called `col_name` in the `mtcars` dataset. How to solve this issue and make R understand to not take "cyl" literally as a string, but to interpret it?

4.2.2 Getting serious with rlang

One way is to use the `quo()` function, in conjunction with the `!!` operator introduced with `dplyr` 0.7 (but actually part of the `rlang` package, which gets used by `dplyr` seamlessly). First let's look at the solution and then I'll explain how it works:

```
library(dplyr)

simple_function <- function(dataset, col_name){
  col_name <- enquo(col_name)
  dataset %>%
    group_by(!!col_name) %>%
    summarise(mean_mpg = mean(mpg)) -> dataset
  return(dataset)
}


simple_function(mtcars, cyl)
```

```
## # A tibble: 3 x 2
##     cyl mean_mpg
##   <dbl>    <dbl>
## 1     4 26.66364
## 2     6 19.74286
## 3     8 15.10000
```

4.3 Exercises

1. Suppose you have an Excel workbook that contains data on three sheets. Create a function that reads entire workbooks, and that returns a list of tibbles, where each tibble is the data of one sheet (download the example Excel workbook, `example_workbook.xlsx`, from the `assets` folder on the books github).

Chapter 5

Unit testing

5.1 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.

5.2 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:dplyr':
##
##     matches

## The following objects are masked from 'package:magrittr':
##
##     equals, is_less_than, not

## 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))
})
```

5.3 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.

5.4 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.

5.5 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 6

Packages

6.1 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 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.

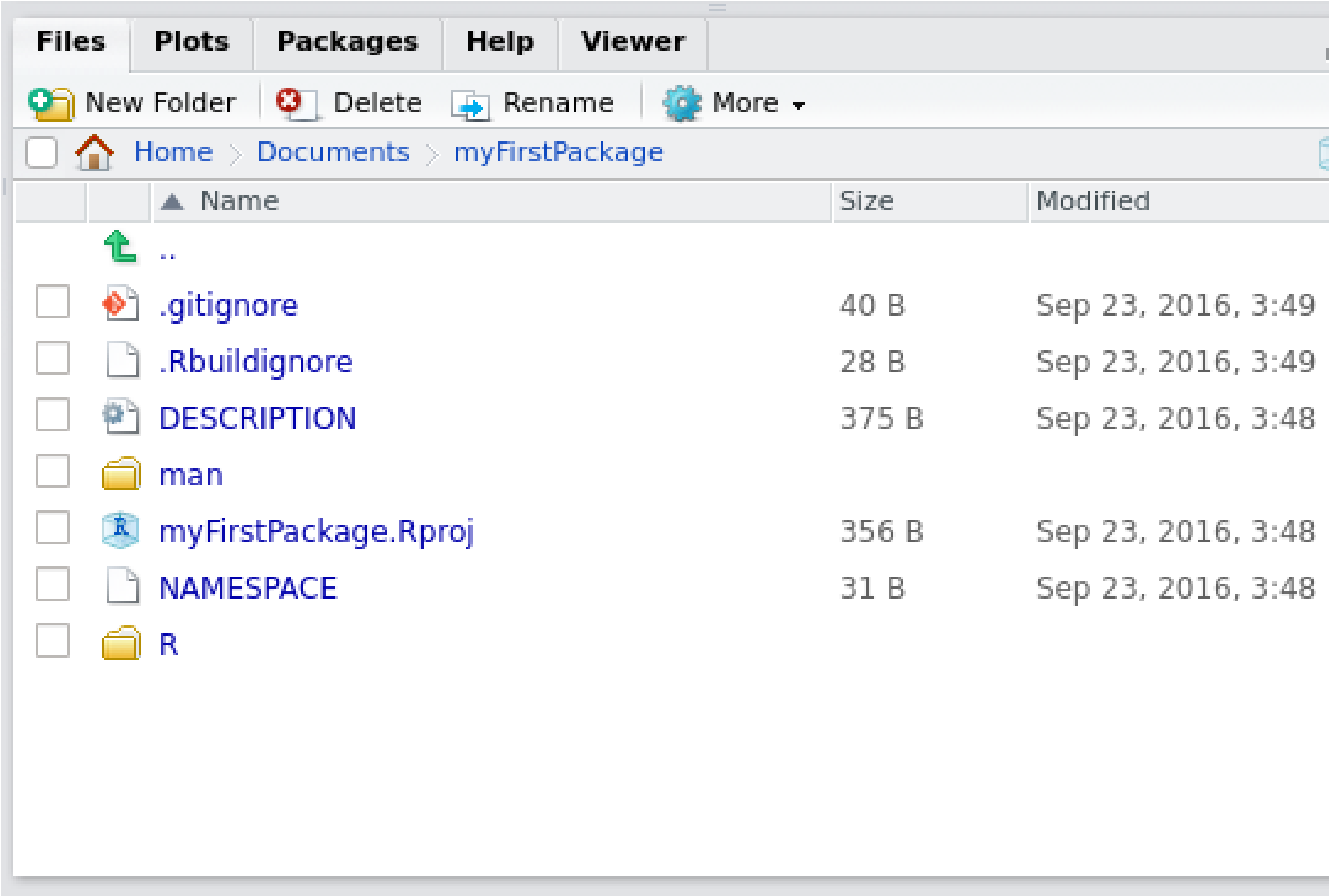
Having everything inside a package takes care of these headaches for you. And code that is inside packages is very easy to test, especially if you're using Rstudio. It also makes it possible to use the wonderful `covr` package, which tells you which lines in which functions are called by your tests. 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 (2014a). 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.

6.2 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
□	.Rbuildignore	28 B	Sep 23, 2016, 3:49
□	DESCRIPTION	375 B	Sep 23, 2016, 3:48
□	man		
□	myFirstPackage.Rproj	356 B	Sep 23, 2016, 3:48
□	NAMESPACE	31 B	Sep 23, 2016, 3:48
□	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.

6.3 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)
}
```

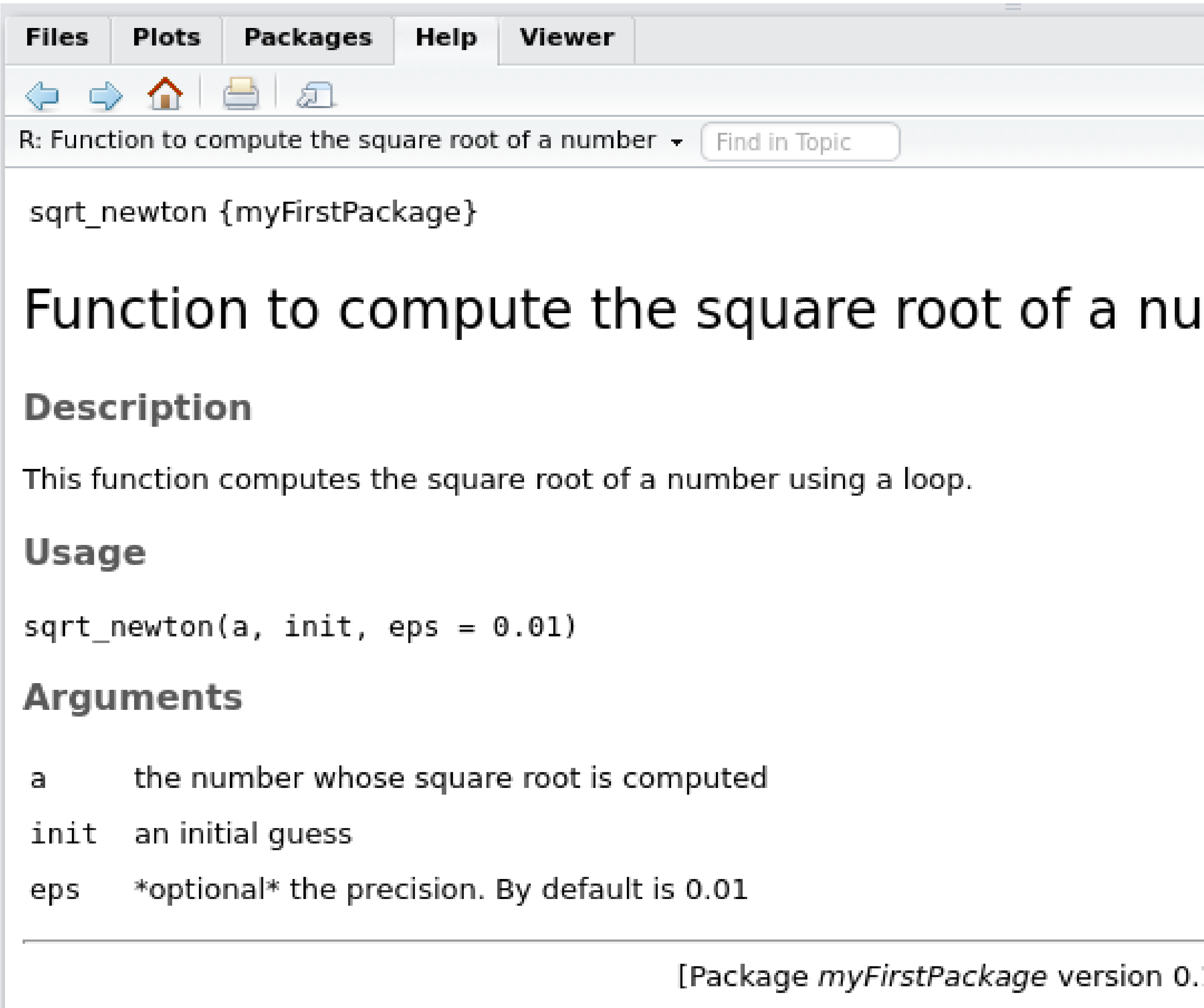
Usually, you would write comments to describe what your function does, what are its inputs and outputs. 'roxygen2' is a package that turns these comments into documentation. Here is what our function would look like with `roxygen2` type comments:

```
#' 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 two symbols, `#'` 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}

Function to compute the square root of a nu

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.

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. In the next subsection, I will mention two files that got created with your package, `NAMESPACE` and `DESCRIPTION`.

6.4 Extra files inside your package and dependencies

6.4.1 The NAMESPACE file

The NAMESPACE file gets generated automatically by roxygien2. You do not have to worry much about it; it lists the functions that are made available to the user when your package gets loaded. If you have helper functions that you do not want to make available to the user, remove the @export keyword from the function comments and the function will not get listed in the NAMESPACE file. There is still a way for the user to access these helper functions like so:

```
package::helper_function
```

This is somewhat similar to the concept of private/public methods in object oriented programming.

6.4.2 How can you use functions from other packages inside your package?

There are two solutions for this. Without going into much details, this is mostly handled by the DESCRIPTION file. This file is very important for two reasons: if you want to publish a package on CRAN, this is the file where you specify your name, contact information and the license of the package. But this does not mean that this file is not important if you want to write your own personal package to clean data: this is also the file where you specify the version of your package, but also where you specify the dependencies of your package. You have two fields for this: Imports and Depends. If you are lazy and want to use functions from other packages inside your package, you can list them in the Depends field. However, you should be careful with this approach; indeed, if you named some of your functions the same as functions from other packages, by loading your package after another, you will "overwrite" these functions. For example, imagine you have function f in package A but also in package B. If you first load A, and then B, f will refer to B::f. It's basically the same problem when attaching datasets, but with functions, so it's even worse! A cleaner way of solving the problem of using functions from other packages inside your package is to list them in the Imports field, and then use :: inside your functions. For example:

```
my_nice_function <- function(x){
    readr::read_csv( bla bla )
    bla bla
}
```

This is much cleaner and avoids the problem described above. If you have to use a lot of functions from the same package, having always to use :: can get annoying quite fast. In these cases, you can use the @import keyword when you document your function:

```
#' 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
#` @import dplyr
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)
}
```

Above, I've added the line:

```
#` @import dplyr
```

which allows me to call `dplyr` functions inside my functions. This is cleaner and does not pollute your session with attached packages. If you check the `NAMESPACE` you will see the following line:

```
import(dplyr)
```

6.5 Unit test your package

Now that we know the basics of package creation, we move on to unit testing your package. Unit testing is very useful, but requires some work, especially because you have to run your unit tests 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 `tests` in the root of your package and inside this `tests` folder create another folder, called `testthat`. The `testthat` folder will hold your unit tests. Inside the `tests` folder, create a script called `test_sqrt_newton.R` and put the following code in it:

```
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!

6.6 Checking the coverage of your unit tests with `covr`

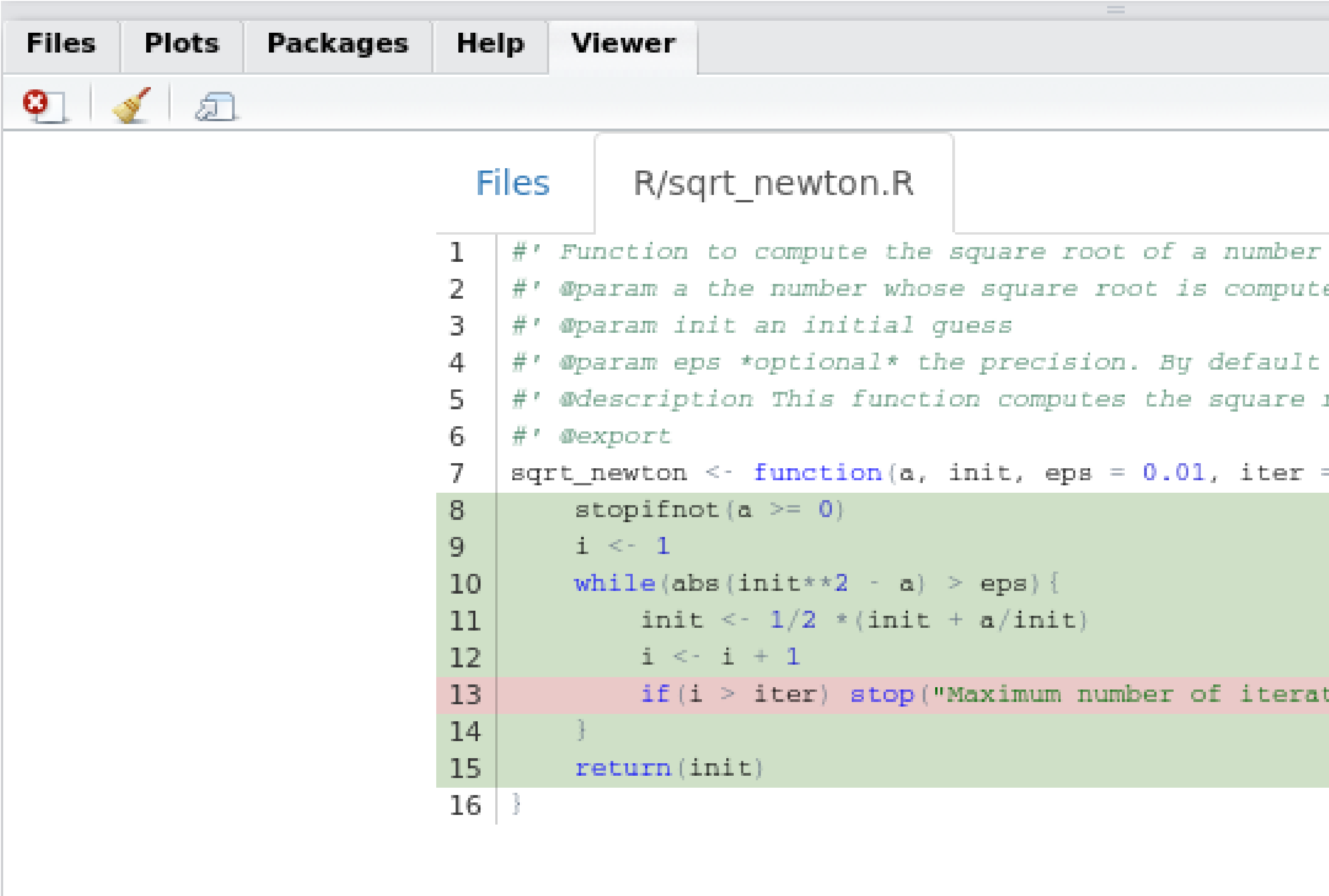
To check the coverage of your package run the following code:

```r
library("covr")

cov <- package_coverage()

shine(cov)
```

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

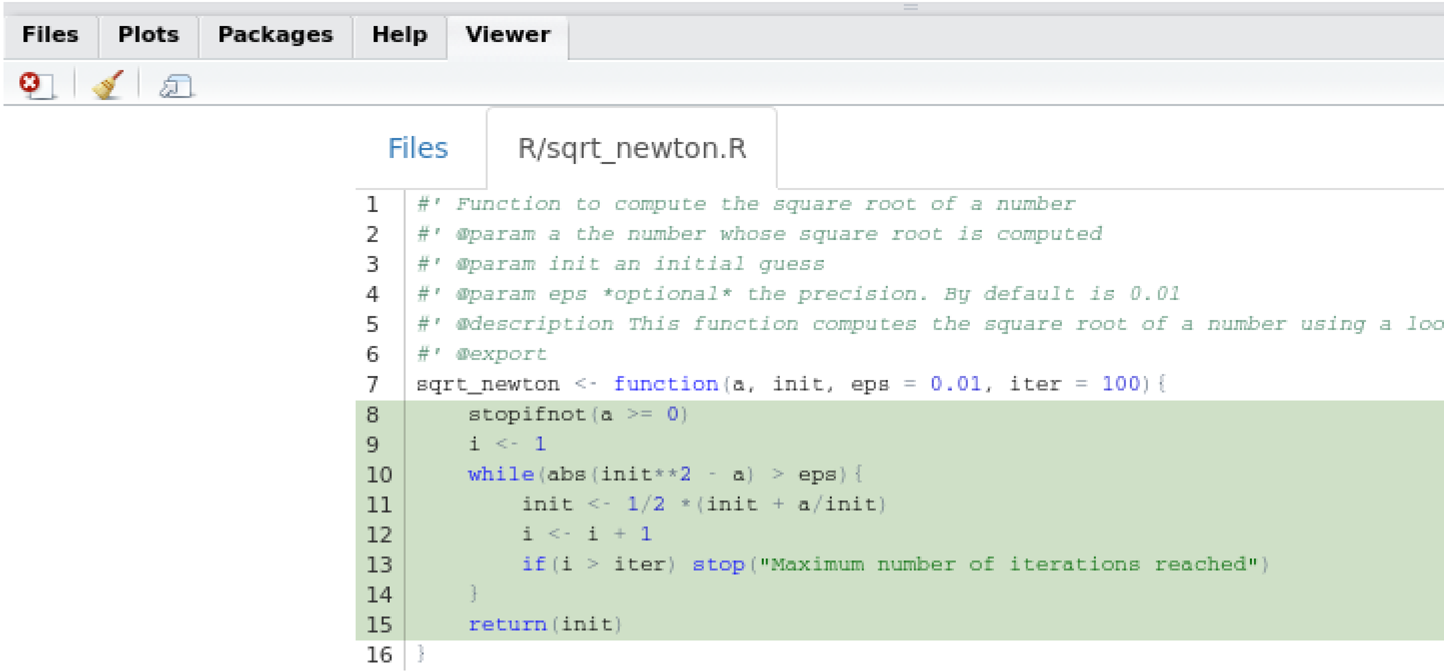


```
#' Function to compute the square root of a number
#' @param a the number whose square root is compute
#' @param init an initial guess
#' @param eps *optional* the precision. By default
#' @description This function computes the square r
#' @export
sqrt_newton <- function(a, init, eps = 0.01, iter =
    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 iterat
    }
    return(init)
}
```

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`:

```
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 loo
#' @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)
}
```

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!

6.7 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 7

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.

7.1 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:

- `data_2000.csv`
- `data_2001.csv`
- `data_2002.csv`
- `data_2003.csv`
- `data_2004.csv`

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

```r
data_2000 <- read.csv("/path/to/data/data_2000.csv", header = T)
data_2001 <- read.csv("/path/to/data/data_2001.csv", header = T)
data_2002 <- read.csv("/path/to/data/data_2002.csv", header = T)
data_2003 <- read.csv("/path/to/data/data_2003.csv", header = T)
data_2004 <- read.csv("/path/to/data/data_2004.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.

7.2 Your first data munging package: `prepareData`

7.2.1 Reading a lot of datasets at once

Using Rstudio, create a new project like shown in the previous chapter, and select *R package*. Give it a name, for example `prepareData`. 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. By the way, we are going to work on some test data sets that I created for illustration purposes. When you will develop your own package to work on your own data, you do not have to write unit tests that use you original data. A subset can be enough, or taking the time to create a small test dataset might be preferable. It depends on what features of your functions you want to test. 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 `broTools`[1] that contains all the little functions that I use daily. But for illustration purposes, we will put this function inside `prepareData`, 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(
        purrr::map(list_of_datasets,
                   read_and_assign,
                   read_func = read_func)
                   )

    # 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
```

[1] It stands for `Bruno Rodrigues' Tools`. I'm still working on releasing the package on Github, and maybe CRAN.

```
    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 main 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`. I use `purrr::map()` to apply `read_and_assign()`; a helper function whose role is to read in a dataset and save it with its name, to the whole list of datasets. This step is wrapped inside `invisible()` as to remove unecessary output. Finally I use `strsplit()` with a regular expression to remove the extension of the dataset from its name. The output is thus a list of datasets where each dataset is named as it is on your hard drive. Save this function in a script called `read_list.R` and save it in the `R` folder of your package. Now you need to invoke `roxygen2::roxygenise()` to create the documentation of your function. I suggest you also run `devtools::use_testtthat`. This creates the necessary folder to hold your tests as well as creating a small `testthat.R` file with the code that gets called to run your tests. Without this, you might encounter weird issues (for example, `covr` not finding your tests!).

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

```
First time using roxygen2. Upgrading automatically...
Updating roxygen version in  /home/bro/Dropbox/prepareData/DESCRIPTION
Writing NAMESPACE
Writing read_list.Rd
```

```
devtools::use_testthat()
```

```
* Adding testthat to Suggests
* Creating `tests/testthat`.
* Creating `tests/testthat.R` from template.
```

Now let us check the coverage of our package:

```
library("covr")

cov <- package_coverage()

shine(cov)
```

Unsurprisingly we get a coverage of 0% for our package. We will now write a unit test for this function. For example, let us see if the condition `stopifnot(length(list_of_datasets)>0)` works. Because you ran `detools::use_testthat()` you should have a folder called `tests` on the root of your project directory. In it, there is a folder called `testthat`. This is were you will save your unit tests, and any file needed for the tests to run (for example, mock datasets that are used by tests).

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

test_that("Try to import empty list of datasets: this may be caused because
          the path to the datasets is wrong for instance",{
```

```
    list_datasets <- NULL

    expect_error(read_list(list_datasets, read_csv, col_types = cols()))
})
```

Run the test using CTRL-SHIFT-T if you are on Rstudio.

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

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

This is the output you should see. If you check the coverage of your package, you should see that the line stopifnot(length(list_of_datasets)>0) is highlightened in green and you should have around 9% of coverage for your package. You can spend some to to get the coverage as high as possible, but you have to take into account the time it will take you to write tests vs the benefits you are going to get from them. In the case of this function, I do not really see what more you could test.

Let us use this function to read in the datasets:

```
library("readr")
library("purrr")
library("tibble")

list_of_data <- Sys.glob("assets/*.csv")

datasets <- read_list(list_of_data, read_csv, col_type = cols())
```

list_of_data is a variable that contains the path to the datasets. I used Sys.glob("assets/*.csv") to find the datasets. The datasets are saved in the assets folder of the book and end with the .csv extension. You could also use list.files("*.csv") to achieve the same. Let's take a look inside this list using head(). Since head() only works on single data frames or tibbles, we use map() to apply head() to each data frame on the list.

```
map(datasets, head)
```

```
## $`assets/data_2000`
## # A tibble: 6 x 6
##      id Variable1 other2000 gender2000 eggs2000      spam2000
##   <int>     <int>     <int>      <chr>    <int>         <chr>
## 1     1        32         3          F       80 -1.5035369157
## 2     2        28         2          F       20 -0.1836726393
## 3     3        36         4          M       58 -0.6851988608
## 4     4        28         1          F       30  1.9900760191
## 5     5        34         3          F       14  0.4324725273
## 6     6        30         3          F       40   -0.79001853
##
## $`assets/data_2001`
## # A tibble: 6 x 6
##      id VARIABLE1 other2001 Gender2001 eggs2001   spam2001
##   <int>     <int>     <int>      <chr>    <int>      <dbl>
## 1     1        32         3          F       80 -1.5035369
## 2     2        28         2          F       20 -0.1836726
## 3     3        36         4          M       58 -0.6851989
## 4     4        28         1          F       30  1.9900760
## 5     5        34         3          F       14  0.4324725
## 6     6        30         3          F       40 -0.7900185
##
## $`assets/data_2002`
## # A tibble: 6 x 6
##      ID variable1 Other2002 gender2002 eggs2002   Spam2002
##   <int>     <int>     <int>      <chr>    <int>      <dbl>
## 1     1        32         3          F       80 -1.5035369
## 2     2        28         2          F       20 -0.1836726
## 3     3        36         4          M       58 -0.6851989
## 4     4        28         1          F       30  1.9900760
## 5     5        34         3          F       14  0.4324725
## 6     6        30         3          F       40 -0.7900185
##
## $`assets/data_2003`
## # A tibble: 6 x 6
##      id variable1 other2003 gender2003 EGGS2003   spam2003
##   <int>     <int>     <int>      <chr>    <int>      <dbl>
## 1     1        32         3          F       80 -1.5035369
## 2     2        28         2          F       20 -0.1836726
## 3     3        36         4          M       58 -0.6851989
## 4     4        28         1          F       30  1.9900760
## 5     5        34         3          F       14  0.4324725
## 6     6        30         3          F       40 -0.7900185
##
## $`assets/data_2004`
## # A tibble: 6 x 6
##      Id Variable1 Other2004 Gender2004 Eggs2004   Spam2004
##   <int>     <int>     <int>      <chr>    <int>      <dbl>
## 1     1        32         3          F       80 -1.5035369
## 2     2        28         2          F       20 -0.1836726
## 3     3        36         4          M       58 -0.6851989
## 4     4        28         1          F       30  1.9900760
## 5     5        34         3          F       14  0.4324725
## 6     6        30         3          F       40 -0.7900185
```

The datasets we will work with all have the the same variables and the same inviduals. We have datasets for the years 2000 to 2004. It would be much better for analysis if we could have clean variable names and merge every datasets together in a single, longitudinal dataset. In short, what we need:

- Have nice names for the columns.
- Remove the year from the name of the columns and add a column containing the year.
- Merge every dataset together.

This is to make the dataset tidy, as explained Wickham (2014b). Of course, depending on your needs, you might need to add further operations, for example creating new variables etc. For now, we are going to focus on these three steps.

7.2.2 Treating the columns of your datasets

Let us take a look at the column names of the datasets:

```
map(datasets, colnames)
```

```
## $`assets/data_2000`
## [1] "id"         "Variable1"  "other2000"  "gender2000" "eggs2000"
## [6] "spam2000"
##
## $`assets/data_2001`
## [1] "id"         "VARIABLE1"  "other2001"  "Gender2001" "eggs2001"
## [6] "spam2001"
##
## $`assets/data_2002`
## [1] "ID"         "variable1"  "Other2002"  "gender2002" "eggs2002"
## [6] "Spam2002"
##
## $`assets/data_2003`
## [1] "id"         "variable1"  "other2003"  "gender2003" "EGGS2003"
## [6] "spam2003"
##
## $`assets/data_2004`
## [1] "Id"         "Variable1"  "Other2004"  "Gender2004" "Eggs2004"
## [6] "Spam2004"
```

This is very messy, we would need to have a function that would clean all this mess and "normalize" these column names. Turns out that we're lucky, and there is exactly what we are looking for in the `janitor` package. The function `janitor::clean_names()` does exactly this. Let's use it and see the output:

```
library("janitor")

datasets <- map(datasets, clean_names)

map(datasets, colnames)
```

```
## $`assets/data_2000`
## [1] "id"         "variable1"  "other2000"  "gender2000" "eggs2000"
## [6] "spam2000"
##
## $`assets/data_2001`
## [1] "id"         "variable1"  "other2001"  "gender2001" "eggs2001"
## [6] "spam2001"
##
## $`assets/data_2002`
## [1] "id"         "variable1"  "other2002"  "gender2002" "eggs2002"
## [6] "spam2002"
##
## $`assets/data_2003`
## [1] "id"         "variable1"  "other2003"  "gender2003" "eggs2003"
## [6] "spam2003"
##
## $`assets/data_2004`
## [1] "id"         "variable1"  "other2004"  "gender2004" "eggs2004"
## [6] "spam2004"
```

This is much better. If `clean_names()` didn't exist, you would have to have written your own function for this. This could have been a complicated exercise, depending on how messy and heterogenous the variable names would have been in your data. However `clean_names()` does a great job, so there's no need to reivent the wheel!

Now we would like to remove the years from the column names and add a column with the name of each dataset. Let us start by removing the years from the column names by writing a function. For this function, a little regular expression knowledge will not hurt. Here is what the function looks like:

```
#' Remove year strings from column names
#' @param list_of_datasets A list containing named datasets
#' @return A list of datasets with the supplied string prepended to the column names
#' @description This function removes year strings from column names, meaning that a column called
#' "eggs9000" gets renamed into "eggs"
#' @export
#' @examples
#' \dontrun{
#' #`list_of_data_sets` is a list containing named data sets
#' # For example, to access the first data set, called dataset_1 you would
#' # write
#' list_of_data_sets$dataset_1
#' remove_years_from_strings(list_of_data_sets)
#' }
remove_years_from_strings <- function(list_of_datasets){

  for_one_dataset <- function(dataset){
    # strsplit() accepts regular expressions, so it's easy to get rid of a number made up of
    # *exactly* 4 digits

    colnames(dataset) <- unlist(strsplit(colnames(dataset), "\\d{4}", perl = TRUE))
    return(dataset)
  }

  output <- purrr::map(list_of_datasets, for_one_dataset)

  return(output)
}
```

and here is the accompanying unit test:

```
library("testthat")
library("prepareData")
library("readr")

data_sets <- list.files(pattern = "2001")

data_list <- read_list(data_sets, read_csv, col_types = cols())

test_that("Test remove years from srings",{
    data_list_result <- purr::map(data_list, janitor::clean_names)
    data_list_result <- remove_years_from_strings(data_list_result)
    expect <- c("id", "year_", "variable1", "other", "gender", "eggs", "spam")
    actual <- colnames(data_list_result[[1]])
    expect_equal(expect, actual)
})
```

For the unit test to work, I had to add the dataset for the year 2001 in the `tests/testthat` directory. Again, this dataset does not have to be the real dataset you will ultimately be working on. A mock dataset with simulated data on 10 rows and with the same column names works exactly the same!

Let's take a look at the output:

```
datasets <- remove_years_from_strings(datasets)

map(datasets, colnames)
```

```
## $`assets/data_2000`
## [1] "id"        "variable1" "other"     "gender"    "eggs"      "spam"
##
## $`assets/data_2001`
## [1] "id"        "variable1" "other"     "gender"    "eggs"      "spam"
##
## $`assets/data_2002`
## [1] "id"        "variable1" "other"     "gender"    "eggs"      "spam"
##
## $`assets/data_2003`
## [1] "id"        "variable1" "other"     "gender"    "eggs"      "spam"
##
## $`assets/data_2004`
## [1] "id"        "variable1" "other"     "gender"    "eggs"      "spam"
```

This is starting to look like something!

Now, since we removed the years from the column names, we need to add a column containing the year to our datasets. And now to add the year column:

```
#' Adds the year column
#' @param list_of_datasets A list containing named datasets
#' @return A list of datasets with the year column
#' @description This function works by extracting the year string contained in
#' the data set name and appending a new column to the data set with the numeric
#' value of the year. This means that the data sets have to have a name of the
#' form data_set_2001 or data_2001_europe, etc
#' @export
#' @examples
#' \dontrun{
#' #`list_of_data_sets` is a list containing named data sets
#' # For example, to access the first data set, called dataset_1 you would
#' # write
#' list_of_data_sets$dataset_1
#' add_year_column(list_of_data_sets)
#' }
add_year_column <- function(list_of_datasets){
  for_one_dataset <- function(dataset, dataset_name){

    # Split the name of the dataset at the "_". The datasets must have a name of the
    # form "data_2000" (notice the underscore).
    name_year <- unlist(strsplit(dataset_name, "[_.]"))
    # Get the index of the string that contains digits
    index <- grep("\\d+", name_year)

    # Get the year
    year <- as.numeric(name_year[index])

    # Add it to the data set
    dataset$year <- year
    return(dataset)
  }

  output <- purrr::map2(list_of_datasets, names(list_of_datasets), for_one_dataset)
  return(output)
}
```

And its unit test:

```
library("testthat")
library("prepareData")
library("readr")


data_sets <- list.files(pattern = "data")

data_list <- read_list(data_sets, read_csv, col_types = cols())

test_that("Test add year column",{
    data_list_result <- purrr::map(data_list, janitor::clean_names)
    data_list_result <- add_year_column(data_list_result)
    expect <- list(rep(2001, 1000), rep(2002, 1000))
    actual <- list(data_list_result[[1]]$year, data_list_result[[2]]$year)
    expect_equal(expect, actual)
})
```

This function does not work if the names of the datasets are not of the form “data_2000”. This means that this function should have either an additional argument, where you specify the separator (for example “_” or “.” or even “-”) or fail if the name does not contain an “_“. I like the second solution better:

```
#' Adds the year column
#' @param list_of_datasets A list containing named datasets
#' @return A list of datasets with the year column
#' @description This function works by extracting the year string contained in
#' the data set name and appending a new column to the data set with the numeric
#' value of the year. This means that the data sets have to have a name of the
#' form data_set_2001 or data_2001_europe, etc
#' @export
#' @examples
#' \dontrun{
```

```
#' #`list_of_data_sets` is a list containing named data sets
#' # For example, to access the first data set, called dataset_1 you would
#' # write
#' list_of_data_sets$dataset_1
#' add_year_column(list_of_data_sets)
#' }
add_year_column <- function(list_of_datasets){

  for_one_dataset <- function(dataset, dataset_name){

    if(!("_" %in% unlist(strsplit(dataset_name, split = "")))){
    stop("Make sure that your datasets are named like
         `data_2000.csv` or similar. The `_` between `data`
         and `2000` is what matters")}

    # Split the name of the dataset at the "_". The datasets must have a name of the
    # form "data_2000" (notice the underscore).
    name_year <- unlist(strsplit(dataset_name, split = "[_.]"))
    # Get the index of the string that contains digits
    index <- grep("\\d+", name_year)

    # Get the year
    year <- as.numeric(name_year[index])

    # Add it to the data set
    dataset$year <- year
    return(dataset)
  }

  output <- purrr::map2(list_of_datasets, names(list_of_datasets), for_one_dataset)
  return(output)
}
```

If you check the coverage of this function, you will see that the lines that test if the datasets are correctly named do not get called. Let's add a unit test that does this, but first, we need to create *wrong* datasets. Just copy the datasets you have in your tests folder, and rename them to `wrongdata2001.csv` and `wrongdata2002.csv`. We expect our function to stop with an error message if it tries anything on these datasets:

```
data_sets <- list.files(pattern = "wrong")

data_list <- read_list(data_sets, read_csv, col_types = cols())

test_that("Test add year column: wrong name",{
    data_list_result <- purrr::map(data_list, janitor::clean_names)
    expect_error(add_year_column(data_list_result))
})
```

Now have fully covered your function, and you also know when the function breaks. With the informative error message, future you or your coworkers will know how to correctly name the datasets. Let's try `add_year_column()` to see how it behaves on our data:

```
datasets <- add_year_column(datasets)

map(datasets, head)
```

```
## $`assets/data_2000`
## # A tibble: 6 x 7
##      id variable1 other gender  eggs          spam  year
##   <int>     <int> <int>  <chr> <int>         <chr> <dbl>
## 1     1        32     3      F    80 -1.5035369157  2000
## 2     2        28     2      F    20 -0.1836726393  2000
## 3     3        36     4      M    58 -0.6851988608  2000
## 4     4        28     1      F    30  1.9900760191  2000
## 5     5        34     3      F    14  0.4324725273  2000
## 6     6        30     3      F    40   -0.79001853  2000
##
## $`assets/data_2001`
## # A tibble: 6 x 7
##      id variable1 other gender  eggs       spam  year
##   <int>     <int> <int>  <chr> <int>      <dbl> <dbl>
## 1     1        32     3      F    80 -1.5035369  2001
## 2     2        28     2      F    20 -0.1836726  2001
## 3     3        36     4      M    58 -0.6851989  2001
## 4     4        28     1      F    30  1.9900760  2001
## 5     5        34     3      F    14  0.4324725  2001
## 6     6        30     3      F    40 -0.7900185  2001
##
## $`assets/data_2002`
## # A tibble: 6 x 7
##      id variable1 other gender  eggs       spam  year
##   <int>     <int> <int>  <chr> <int>      <dbl> <dbl>
## 1     1        32     3      F    80 -1.5035369  2002
## 2     2        28     2      F    20 -0.1836726  2002
## 3     3        36     4      M    58 -0.6851989  2002
## 4     4        28     1      F    30  1.9900760  2002
## 5     5        34     3      F    14  0.4324725  2002
## 6     6        30     3      F    40 -0.7900185  2002
##
## $`assets/data_2003`
## # A tibble: 6 x 7
##      id variable1 other gender  eggs       spam  year
##   <int>     <int> <int>  <chr> <int>      <dbl> <dbl>
## 1     1        32     3      F    80 -1.5035369  2003
## 2     2        28     2      F    20 -0.1836726  2003
## 3     3        36     4      M    58 -0.6851989  2003
## 4     4        28     1      F    30  1.9900760  2003
## 5     5        34     3      F    14  0.4324725  2003
## 6     6        30     3      F    40 -0.7900185  2003
##
## $`assets/data_2004`
## # A tibble: 6 x 7
##      id variable1 other gender  eggs       spam  year
##   <int>     <int> <int>  <chr> <int>      <dbl> <dbl>
## 1     1        32     3      F    80 -1.5035369  2004
## 2     2        28     2      F    20 -0.1836726  2004
## 3     3        36     4      M    58 -0.6851989  2004
## 4     4        28     1      F    30  1.9900760  2004
## 5     5        34     3      F    14  0.4324725  2004
## 6     6        30     3      F    40 -0.7900185  2004
```

Just as expected!

TBC...

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

Wickham, H. (2014b). Tidy data. *Journal of Statistical Software*, 59(1):1–23.

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.