

# **Building reproducible analytical pipelines with the R programming language**

Bruno Rodrigues, ...

1/6/23

# Table of contents

<b>Preface</b>	<b>4</b>
<b>1 Introduction</b>	<b>5</b>
1.1 Prerequisites . . . . .	5
1.2 What is reproducibility? . . . . .	5
1.3 Are there different types of reproducibility? . . . . .	5
<b>2 Functional programming</b>	<b>6</b>
2.1 Introduction . . . . .	6
2.1.1 The state of your program . . . . .	6
2.1.2 Predictable functions . . . . .	7
2.1.3 Referentially transparent and pure functions . . . . .	11
2.2 Writing good functions . . . . .	12
2.2.1 Functions are first class objects . . . . .	12
2.2.2 Optional arguments . . . . .	16
2.2.3 Safe functions . . . . .	17
2.2.4 Recursive functions . . . . .	18
2.2.5 Anonymous functions . . . . .	20
2.2.6 The Unix philosophy applied to R . . . . .	20
2.3 Lists: a powerful data-structure . . . . .	21
2.3.1 Lists all the way down . . . . .	21
2.3.2 Lists can hold many things . . . . .	22
2.3.3 Lists as the cure to loops . . . . .	25
2.3.4 Data frames . . . . .	29
2.4 Functional programming in R . . . . .	41
2.4.1 Base capabilities . . . . .	41
2.4.2 purrr . . . . .	45
2.4.3 withr . . . . .	46
2.5 Conclusion . . . . .	48
<b>3 Version control</b>	<b>49</b>
<b>4 Keeping it DRY with literate programming</b>	<b>50</b>
4.1 Literate programming . . . . .	50
4.2 {knitr} basics . . . . .	58
4.2.1 Set up . . . . .	59

4.2.2	Markdown ultrabasics . . . . .	60
4.3	Keeping it DRY . . . . .	64
4.3.1	Generating R Markdown code from code . . . . .	64
4.3.2	Tables in R Markdown documents . . . . .	67
4.3.3	Parametrized reports . . . . .	68
<b>5</b>	<b>Project start</b>	<b>73</b>
5.1	Housing in Luxembourg . . . . .	73
5.2	Saving trapped data from Excel . . . . .	76
5.3	Your project is done (?) . . . . .	79
<b>6</b>	<b>Testing your code</b>	<b>80</b>
6.1	Assertive testing (and defenise programming?) . . . . .	80
<b>7</b>	<b>Packaging your code</b>	<b>81</b>
7.1	Benefits of packages . . . . .	81
7.2	Intro to packge dev . . . . .	81
7.3	Document your package (?) . . . . .	81
7.4	Managing package dependencies (?) . . . . .	81
7.5	Unit testing . . . . .	81
7.6	pkgdown . . . . .	81
<b>8</b>	<b>Build automation</b>	<b>82</b>
<b>9</b>	<b>Introduction to reproducibility</b>	<b>83</b>
<b>10</b>	<b>Advanced topics in reproducibility</b>	<b>84</b>
10.1	First steps with Docker . . . . .	84
10.2	A primer on the Linux command line . . . . .	84
10.3	Dockrizing your project . . . . .	84
<b>11</b>	<b>Continuous integration and continuous deployment/delivery</b>	<b>85</b>
	<b>References</b>	<b>86</b>

# Preface

This is a Quarto book.

To learn more about Quarto books visit <https://quarto.org/docs/books>.

# 1 Introduction

This book will teach how to structure your projects in such a way as to make them reproducible.

*I think we should also explain that this book presents one way of doing things, and that it is by no means the only, canonic way, of solving the problems we will be talking about.*

## 1.1 Prerequisites

You should be comfortable with the R programming language. This book will assume that you have been using R for some projects already, and want to improve not only your knowledge of the language itself, but also how to successfully manage complex projects.

## 1.2 What is reproducibility?

## 1.3 Are there different types of reproducibility?

Reproducibility is on a continuum.

## 2 Functional programming

This chapter will teach you the fundamentals of functional programming. *Functional programming* might sound scary, but we will focus on only a handful of concepts that are quite accessible while providing many benefits. Using these functional programming concepts will make your code more reliable, easier to test, document, share, and ultimately rerun.

### 2.1 Introduction

You are very likely already familiar with some aspects of functional program. Let's start by discussing the two central elements of functional programming: functions and lists.

There are several ways that you can structure a program, called programming paradigms. Functional programming is a paradigm that relies exclusively on the evaluation of functions to achieve the desired end result. If you have already written your own functions in the past, what follows will not be very new. But in order to write a good functional program, the functions that you write and evaluate have to have certain properties. Before discussing these properties, let's start by with *state*.

#### 2.1.1 The state of your program

Let's suppose that you start a fresh R session, and immediately run this next line:

```
ls()
```

If you did not modify any of R's configuration files that get automatically loaded on startup, you should see the following:

```
character(0)
```

Let's suppose that now you load some data:

```
data(mtcars)
```

and define a variable `a`:

```
a <- 1
```

Running `ls()` now shows the following:

```
[1] "a"      "mtcars"
```

You have just altered the state of your program. You can think of the *state* as a box that holds everything that gets defined by the user and is accessible at any time. Let's now define a simple function that prints a sentence:

```
f <- function(name){  
  print(paste0(name, " likes lasagna"))  
}  
  
f("Bruno")
```

and here's the output:

```
[1] "Bruno likes lasagna"
```

Let's run `ls()` again:

```
[1] "a"      "f"      "mtcars"
```

Function `f()` is now listed there as well. This function has two nice properties:

- For a given input, it always returns exactly the same output. So `f("Bruno")` will always return "Bruno likes lasagna".
- This function does not change the state of your program, by adding new objects every time it's run.

### 2.1.2 Predictable functions

Let's now define another function called `g()`, that does not have the same properties as `f()`. First, let's define a function that does not always return the same output given a particular input:

```
g <- function(name){  
  food <- sample(c("lasagna", "cassoulet", "feijoadada"), 1)  
  print(paste0(name, " likes ", food))  
}
```

For the same input, “Bruno”, this function now produces (potentially) a different output:

```
g("Bruno")  
[1] "Bruno likes lasagna"
```

```
g("Bruno")  
[1] "Bruno likes feijoada"
```

And now let’s consider function `h()` that modifies the state of the program:

```
h <- function(name){  
  food <- sample(c("lasagna", "cassoulet", "feijoada"), 1)  
  
  if(exists("food_list")){  
    food_list <- append(food_list, food)  
  } else {  
    food_list <- append(list(), food)  
  }  
  
  print(paste0(name, " likes ", food))  
}
```

This function uses the `<-` operator. This operator saves definitions that are made inside the body of functions in the global environment. Before calling this function, run `ls()` again. You should see the same objects as before, plus the new functions we’ve defined:

```
[1] "a"          "f"          "g"          "h"          "mtcars"
```

Let’s now run `h()` once:

```
h("Bruno")  
[1] "Bruno likes feijoada"
```

And now `ls()` again:

```
[1] "a"          "f"          "food_list" "g"          "h"          "mtcars"
```

Running `h()` did two things: it printed the message, but also created a variable called “`food_list`” in the global environment with the following contents:

```
food_list
```



```
[[1]]  
[1] "feijoada"
```

Let's run `h()` again:

```
h("Bruno")  
[1] "Bruno likes cassoulet"
```

and let's check the contents of "food\_list":

```
food_list  
  
[[1]]  
[1] "feijoada"  
  
[[2]]  
[1] "cassoulet"
```

If you keep running `h()`, this list will continue growing. Let me just say that I hesitated showing you this; this is because if you didn't know `<<-`, you might find the example above useful. But while useful, it is quite dangerous as well. Generally, we want to avoid using functions that change the state as much as possible because these function are unpredictable, especially if randomness is involved. It is much safer to define `h()` like this instead:

```
h <- function(name, food_list = list()){  
  
  food <- sample(c("lasagna", "cassoulet", "feijoada"), 1)  
  
  food_list <- append(food_list, food)  
  
  print(paste0(name, " likes ", food))  
  
  food_list  
}
```

The difference now is that we made `food_list` the second argument of the function. Also, we defined it as being optional by writing:

```
food_list = list()
```

This means that if we omit this argument, the empty list will get used by default. This avoids the users having to manually specify it.

We can call it like this:

```
food_list <- h("Bruno", food_list) # since food_list is already defined, we don't need to  
  
[1] "Bruno likes feijoada"
```

We save the output back to `food_list`. Let's now check its contents:

```
food_list  
  
[[1]]  
[1] "feijoada"  
  
[[2]]  
[1] "cassoulet"  
  
[[3]]  
[1] "feijoada"
```

The only thing that we need now to deal with is the fact that the food gets chosen randomly. I'm going to show you the simple way of dealing with this, but later in this chapter we are going to use the `{withr}` package for situations like this. Let's redefine `h()` one last time:

```
h <- function(name, food_list = list(), seed = 123){  
  
  # We set the seed, making sure that we get the same selection of food for a given seed  
  set.seed(seed)  
  food <- sample(c("lasagna", "cassoulet", "feijoada"), 1)  
  
  # We now need to unset the seed, because if we don't, guess what, the seed will stay set  
  set.seed(NULL)  
  
  food_list <- append(food_list, food)  
  
  print(paste0(name, " likes ", food))  
  
  food_list  
}
```

Let's now call `h()` several times with its default arguments:

```
h("Bruno")
```

```
[1] "Bruno likes feijoadá"  
[[1]]  
[1] "feijoadá"
```

```
h("Bruno")
```

```
[1] "Bruno likes feijoadá"  
[[1]]  
[1] "feijoadá"
```

```
h("Bruno")
```

```
[1] "Bruno likes feijoadá"  
[[1]]  
[1] "feijoadá"
```

As you can see, every time this function runs, it now produces the same result. Users can change the seed to have this function produce, consistently, another result.

### 2.1.3 Referentially transparent and pure functions

A referentially transparent function is a function that does not use any variable that is not also one of its inputs. For example, the following function:

```
bad <- function(x){  
  x + y  
}
```

is not referentially transparent, because `y` is not one of the functions inputs. What happens if you run `bad()` is that `bad()` needs to look for `y`. Because `y` is not one of its inputs, `bad()` then looks for it in the global environment. If `y` is defined there, it then gets used. Defining and using such functions must be avoided at all costs, because these functions are unpredictable. For example:

```
y <- 10  
bad <- function(x){  
  x + y
```

```
}  
  
bad(5)
```

This will return 15. But if `y <- 45` then `bad(5)` would this time around return 50. It is much safer, and easier to make `y` an explicit input of the function instead of having to keep track of `y`'s value:

```
good <- function(x, y){  
  x + y  
}
```

`good()` is a referentially transparent function; it is much safer than `bad()`. `good()` is also a pure function, because it's a function that does not interact in any way with the global environment. It does not write anything to the global environment, nor requires anything from the global environment. Function `h()` from the previous section was not pure, because it created an object and wrote it to the global environment (the `food_list` object). Turns out that pure functions are thus necessarily referentially transparent.

So the first lesson in your functional programming journey that you have to remember is to only use pure functions.

## 2.2 Writing good functions

### 2.2.1 Functions are first class objects

In a functional programming language, functions are first class objects. Contrary to what the name implies, this means that functions, especially the ones you define yourself, are nothing special. A function is an object like any other, and can thus be manipulated as such. Think of anything that you can do with any object in R, and you can do the same thing with a function. For example, let's consider the `+`() function. It takes two numeric objects and returns their sum:

```
1 + 5.3
```

```
[1] 6.3
```

```
# or alternatively: `+`(1, 5.3)
```

You can replace the numbers by functions that return numbers:

```
sqrt(1) + log(5.3)
```

```
[1] 2.667707
```

It's also possible to define a function that explicitly takes another function as an input:

```
h <- function(number, f){  
  f(number)  
}
```

You can call then use `h()` as a wrapper for `f()`:

```
h(4, sqrt)
```

```
[1] 2
```

```
h(10, log10)
```

```
[1] 1
```

Because `h()` takes another function as an argument, `h()` is called a higher-order function.

If you don't know how many arguments `f()`, the function you're wrapping, has, you can use the ...:

```
h <- function(number, f, ...){  
  f(number, ...)  
}
```

... are simply a placeholder for any potential additional argument that `f()` might have:

```
h(c(1, 2, NA, 3), mean, na.rm = TRUE)
```

```
[1] 2
```

```
h(c(1, 2, NA, 3), mean, na.rm = FALSE)
```

```
[1] NA
```

`na.rm` is an argument of `mean()`. As the developer of `h()`, I don't necessarily know what `f()` might be, or maybe I know `f()` and know all its arguments, but don't want to have to rewrite them all to make them arguments of `h()`, so I can use `...` instead. The following is also possible:

```
w <- function(...){
  paste0("First argument: ", ..1, ", second argument: ", ..2, ", last argument: ", ..3)
}

w(1, 2, 3)
```

```
[1] "First argument: 1, second argument: 2, last argument: 3"
```

If you want to learn more about `...`, type `?dots` in an R console.

Because functions are nothing special, you can also write functions that return functions. As an illustration, we'll be writing a function that converts warnings to errors. This can be quite useful if you want your functions to fail early, which often makes debugging easier. For example, try running this:

```
sqrt(-5)
```

```
Warning in sqrt(-5): NaNs produced
```

```
[1] NaN
```

This only raises a warning and returns `NaN` (Not a Number). This can be quite dangerous, especially when working non-interactively, which is what we will be doing a lot later on. It is much better if a pipeline fails early due to an error, than dragging an `NaN` value. This also happens with `log()`:

```
sqrt(-10)
```

```
Warning in sqrt(-10): NaNs produced
```

```
[1] NaN
```

So it could be useful to redefine this functions to raise an error instead, for example like this:

```
strict_sqrt <- function(x){
  if(x <= 0) stop("x is negative")
  sqrt(x)
}
```

This function now throws an error for negative x:

```
strict_sqrt(-10)
```

Error in strict\_sqrt(-10) : x is negative

However, it can be quite tedious to redefine every function that we need in our pipeline. This is where a function factory is useful. We can define a function that takes a function as an argument, converts any warning thrown by that function into an error, and returns the new function. For example it could look like this:

```
strictly <- function(f){
  function(...){
    tryCatch({
      f(...)
    },
    warning = function(warning)stop("Can't do that chief"))
  }
}
```

This function makes use of `tryCatch()` which catches warnings raised by an expression (in this example the expression is `f(...)`) and then raises an error instead with the `stop()` function. It is now possible to define new functions like this:

```
s_sqrt <- strictly(sqrt)
```

```
s_sqrt(-4)
```

Error in value[[3L]](cond) : Can't do that chief

```
s_log <- strictly(log)
```

```
s_log(-4)
```

```
Error in value[[3L]](cond) : Can't do that chief
```

Functions that return functions are called *functions factories* and they're incredibly useful. I use this so much that I've written a package, available on CRAN, called `{chronicler}`, that does this:

```
s_sqrt <- chronicler::record(sqrt)
```

```
result <- s_sqrt(-4)
```

```
result
```

```
NOK! Value computed unsuccessfully:
```

```
-----
```

```
Nothing
```

```
-----
```

```
This is an object of type `chronicle`.
```

```
Retrieve the value of this object with pick(.c, "value").
```

```
To read the log of this object, call read_log(.c).
```

Because the expression above resulted in an error, `Nothing` is returned. `Nothing` is a special value defined in the `{maybe}` package (check it out, very interesting package!). We can then even read the log to see what went wrong:

```
chronicler::read_log(result)
```

```
[1] "Complete log:"
```

```
[2] "NOK! sqrt() ran unsuccessfully with following exception: NaNs produced at 2023-01-30 19
```

```
[3] "Total running time: 0.0010223388671875 secs"
```

The `{purrr}` package also comes with function factories that you might find useful (`{possibly}`, `{safely}` and `{quietly}`).

## 2.2.2 Optional arguments

It is possible to make function arguments optional, by using `NULL`. For example:



```
g <- function(x, y = NULL){
  if(is.null(y)){
    print("optional argument y is NULL")
    x
  } else {
    if(y == 5) print("y is present"); x+y
  }
}
```

Calling `g(10)` prints the message “Optional argument y is NULL”, and returns 10. Calling `g(10, 5)` however, prints “y is present” and returns 15. It is also possible to use `missing()`:

```
g <- function(x, y){
  if(missing(y)){
    print("optional argument y is missing")
    x
  } else {
    if(y == 5) print("y is present"); x+y
  }
}
```

I however prefer the first approach, because it is clearer which arguments are optional, which is not the case with the second approach, where you need to read the body of the function.

### 2.2.3 Safe functions

It is important that your functions are safe and predictable. You should avoid writing functions that behave like `nchar()`, a base R function. Let’s see why this function is not safe:

```
nchar("10000000")
```

```
[1] 8
```

It returns the expected result of 8. But what if I remove the quotes?

```
nchar(10000000)
```

```
[1] 5
```

What is going on here? I’ll give you a hint: simply type 10000000 in the console:

```
10000000
```

```
[1] 1e+07
```

10000000 gets represented as `1e+07` by R. This number in scientific notation gets then converted into the character `"1e+07"` by `nchar()`, and this conversion happens silently. `nchar()` then counts the number of characters, and *correctly* returns 5. The problem is that it doesn't make sense to provide a number to a function that expects a character. This function should have returned an error message, or at the very least raised a warning that the number got converted into a character. Here is how you could rewrite `nchar()` to make it safer:

```
nchar2 <- function(x, result = 0){  
  if(!isTRUE(is.character(x))){  
    stop(paste0("x should be of type 'character', but is of type '",  
               typeof(x), "' instead."))  
  } else if(x == ""){  
    result  
  } else {  
    result <- result + 1  
    split_x <- strsplit(x, split = "")[[1]]  
    nchar2(paste0(split_x[-1],  
                  collapse = ""), result)  
  }  
}
```

This function now returns an error message if the input is not a character:

```
nchar2(10000000)
```

```
Error in nchar2(10000000) : x should be of type 'character', but is of type 'integer' instead
```

## 2.2.4 Recursive functions

You may have noticed that in the last lines of `nchar2()`, that `nchar2()` calls itself. A function that calls itself in its own body is called a recursive function. It is sometimes easier to write down a function in its recursive form than in an iterative form. The most common example is the factorial function. However, there is an issue with recursive functions (in the R programming language, other programming languages may not have the same problem, like Haskell):

while it is sometimes easier to write down a function using a recursive algorithm than an iterative algorithm, like for the factorial function, recursive functions in R are quite slow. Let's take a look at two definitions of the factorial function, one recursive, the other iterative:

```
fact_iter <- function(n){
  result = 1
  for(i in 1:n){
    result = result * i
    i = i + 1
  }
  result
}

fact_recur <- function(n){
  if(n == 0 || n == 1){
    result = 1
  } else {
    n * fact_recur(n-1)
  }
}
```

Using the `{microbenchmark}` package we can benchmark the code:

```
microbenchmark::microbenchmark(
  fact_recur(50),
  fact_iter(50)
)
```

Unit: microseconds

	expr	min	lq	mean	median	uq	max	neval
fact_recur(50)		21.501	21.701	23.82701	21.901	22.0515	68.902	100
fact_iter(50)		2.000	2.101	2.74599	2.201	2.3510	21.000	100

We see that the recursive factorial function is 10 times slower than the iterative version. In this particular example it doesn't make much of a difference, because the functions only take microseconds to run. But if you're working with more complex functions, this is a problem. If you want to keep using the recursive function and not switch to an iterative algorithm, there are workarounds. The first is called *trampolining*. I won't go into details, but if you're interested, there is an R package that allows you to use trampolining with R, aptly called `{trampoline}`. Another solution is using the `{memoise}` package.

## 2.2.5 Anonymous functions

It is possible to define a function and not give it a name. For example:

```
function(x)(x+1)(10)
```

Since R version 4.1, there is even a shorthand notation for anonymous functions:

```
(\ (x)(x+1))(10)
```

Because we don't name them, we cannot reuse them. So why is this useful? Anonymous functions are useful when you need to apply a function somewhere inside a pipe once, and don't want to define a function just for this. This will become clearer once we learn about lists, but before that, let's philosophize a bit.

## 2.2.6 The Unix philosophy applied to R

This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

*Doug McIlroy, in A Quarter Century of Unix<sup>1</sup>*

We can take inspiration from the Unix philosophy and rewrite it like this for our purposes:

*Write functions that do one thing and do it well. Write functions that work together. Write functions that handle lists, because that is a universal interface.*

Strive for writing simple functions that only perform one task. Don't hesitate to split a big function into smaller ones. Small functions that only perform one task are easier to maintain, test, document and debug. These smaller functions can then be chained using the `|>` operator. In other words, it is preferable to have something like:

```
a |> f() |> g() |> h()
```

where `a` is for example a path to a data set, and where `f()`, `g()` and `h()` successively read, clean, and plot the data, than having something like:

```
big_function(a)
```

---

<sup>1</sup><https://stackoverflow.com/a/68690065/1298051>

that does all the steps above in one go.

This idea of splitting the problem into smaller chunks, each chunk in turn split into even smaller units that can be handled by functions and then the results of these function combined into a final output is called composability.

The advantage of splitting `big_function()` into `f()`, `g()` and `h()` is that you can tackle big problems *one bite at a time*, and also reusing these smaller functions in other projects is much easier. So what's important is that you can make small functions work together by sharing a common interface. The list is usually a good candidate for this.

## 2.3 Lists: a powerful data-structure

Lists are the second important ingredient of functional programming. In the R philosophy inspired from UNIX, I stated that *lists are an universal interface* in R, so our functions should handle lists. This of course depends on what it is your doing. If you need functions to handle numbers, then there's little value in placing these numbers inside lists. But in practice, you will very likely manipulate objects that are more complex than numbers, and this is where lists come into play.

### 2.3.1 Lists all the way down

Lists are extremely flexible, and most very complex objects classes that you manipulate are actually lists, but just fancier. For example, a data frame is a list:

```
data(mtcars)

typeof(mtcars)
```

```
[1] "list"
```

A fitted model is a list:

```
my_model <- lm(hp ~ mpg, data = mtcars)

typeof(my_model)
```

```
[1] "list"
```

A `ggplot` is a list:

```
library(ggplot2)

my_plot <- ggplot(data = mtcars) +
  geom_line(aes(y = hp, x = mpg))

typeof(my_plot)
```

```
[1] "list"
```

It's lists all the way down, and it's not a coincidence. It's because, as stated, lists are very powerful. So it's important to know what you can do with lists.

### 2.3.2 Lists can hold many things

If you write a function that needs to return many objects, the only solution is to place them inside a list. For example, consider this function:

```
sqrtnewton <- function(a, init = 1, eps = 0.01, steps = 1){
  stopifnot(a >= 0)
  while(abs(init**2 - a) > eps){
    init <- 1/2 *(init + a/init)
    steps <- steps + 1
  }
  list(
    "result" = init,
    "steps" = steps
  )
}
```

This function returns the square root of a number using Newton's algorithm, as well as the number of steps, or iterations, it took to reach the solution:

```
result_list <- sqrtnewton(1600)

result_list
```

```
$result
[1] 40
```

```
$steps
[1] 10
```

It is quite common to instead print the number of steps to the console instead of returning them. But the issue with a function that prints something to the console instead of returning it, is that such a function is not pure, as it changes something outside of its scope. It is preferable to instead make the function pure by returning everything inside a neat list. It is then possible to separately save these objects if needed:

```
result <- result_list$result

result_steps <- result_list$steps
```

Or you could define functions that know how to deal with the list:

```
f <- function(result_list){
  list(
    "result" = result_list$result * 10,
    "steps" = result_list$steps + 1
  )
}

f(result_list)
```

```
$result
[1] 400
```

```
$steps
[1] 11
```

It all depends on what you want to do. But it is usually better to keep everything neatly inside a list.

Lists can also hold objects of different types:

```
list(
  "a" = head(mtcars),
  "b" = ~lm(y ~ x)
)
```

```
$a
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4

Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

\$b

~lm(y ~ x)

The list above has two elements, the first is the head of the `mtcars` data frame, the second is a formula object. Lists can even hold other lists:

```
list(
  "a" = head(mtcars),
  "b" = list(
    "c" = sqrt,
    "d" = my_plot # Remember this ggplot object from before?
  )
)
```

\$a

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

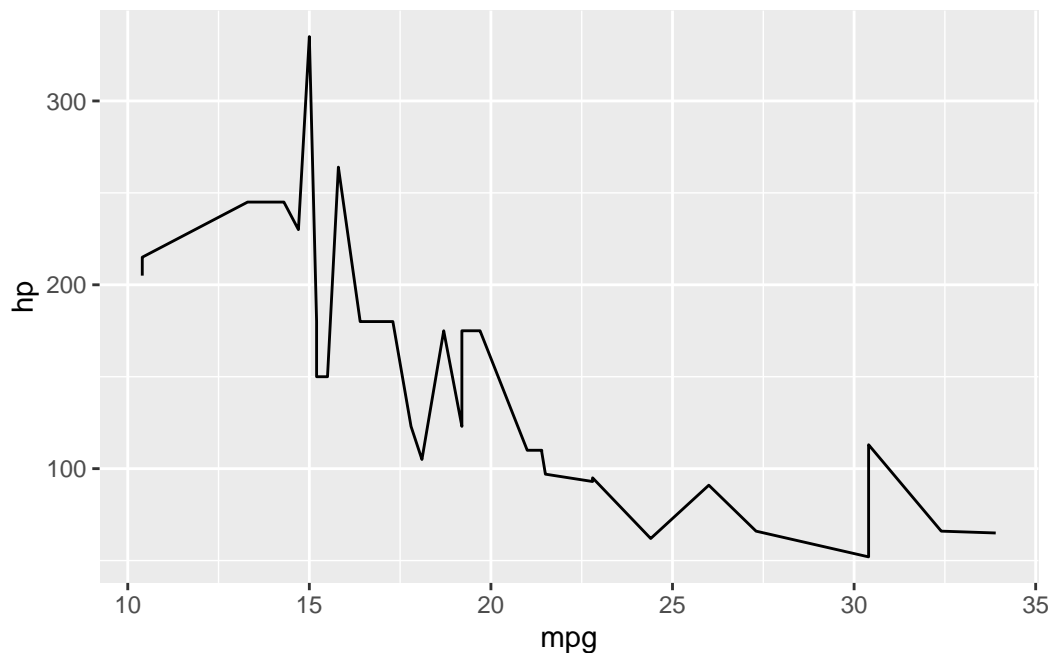
\$b

\$b\$c

function (x) .Primitive("sqrt")

\$b\$d





Use this to your advantage.

### 2.3.3 Lists as the cure to loops

Loops are incredibly useful, and you are likely familiar with them. The problem with loops is that they are a concept from iterative programming, not functional programming, and this is a problem because loops rely on changing the state of your program to function. For example, let's suppose that you wish to use a for-loop to compute the sum of the first 100 integers:

```
result <- 0
for (i in 1:100){
  result <- result + i
}

print(result)
```

```
[1] 5050
```

If you run `ls()` now, you should see that there's a variable `i` in your global environment. This could cause issues further down in your pipeline if you need to re-use `i`. Also, writing loops is, in my opinion, quite error prone. But how can we avoid using loops? For looping in a

functional programming language, we need to use higher-order functions and lists. A reminder: a higher-order function is a function that takes another function as an argument. Looping is a task like any other, so we can write a function that does the looping for us. We will call it `looping()`, which will take a function as an argument, as well as list. The list will serve as the container to hold our numbers:

```
looping <- function(a_list, a_func, init = NULL, ...){

  # If the user does not provide an `init` value, set the head of the list as the initial
  if(is.null(init)){
    init <- a_list[[1]]
    a_list <- tail(a_list, -1)
  }

  # Separate the head from the tail of the list and apply the function to the initial value
  head_list = a_list[[1]]
  tail_list = tail(a_list, -1)
  init = a_func(init, head_list, ...)

  # Check if we're done: if there is still some tail, rerun the whole thing until there's
  if(length(tail_list) != 0){
    looping(tail_list, a_func, init, ...)
  }
  else {
    init
  }
}
```

Now, this might seem much more complicated than a for loop. However, now that we have abstracted the loop away inside a function, we can keep reusing this function:

```
looping(as.list(seq(1:100)), `+`)
```

```
[1] 5050
```

Of course, because this is so useful, `looping()` actually ships with R, and is called `Reduce()`:

```
Reduce(`+`, seq(1:100)) # the order of the arguments is `function` then `list` for `Reduce`
```

```
[1] 5050
```

But this is not the only way that we can loop. We can also write a loop that applies a function to each element of a list, instead of operating on the whole list:

```
result <- as.list(seq(1:5))
for (i in seq_along(result)){
  result[[i]] <- sqrt(result[[i]])
}

print(result)
```

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

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051

[[4]]
[1] 2

[[5]]
[1] 2.236068
```

Here again, we have to pollute the global environment by first creating a vessel for our results, and then apply the function at each index. We can abstract this process away in a function:

```
applying <- function(a_list, a_func, ...){

  head_list = a_list[[1]]
  tail_list = tail(a_list, -1)
  result = a_func(head_list, ...)

  # Check if we're done: if there is still some tail, rerun the whole thing until there's
  if(length(tail_list) != 0){
    append(result, applying(tail_list, a_func, ...))
  }
  else {
    result
  }
}
```

```
}
```

Once again this might seem complicated, and I would agree. Abstraction is complex. But once we have it, we can focus on the task at hand, instead of having to always tell the computer what we want:

```
applying(as.list(seq(1:5)), sqrt)
```

```
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

Of course, R ships with its own, much more efficient, implementation of this function:

```
lapply(list(seq(1:5)), sqrt)
```

```
[[1]]  
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

In other programming languages, `lapply()` is often called `map()`. The `{purrr}` packages ships with other such useful higher-order functions that abstract loops away. For example, there's the function called `map2()`, that maps a function of two arguments to each element of two atomic vectors or lists, two at a time:

```
library(purrr)  
  
map2(  
  .x = seq(1:5),  
  .y = seq(1:5),  
  .f = `+`  
)
```

```
[[1]]  
[1] 2
```

```
[[2]]  
[1] 4
```

```
[[3]]  
[1] 6
```

```
[[4]]  
[1] 8
```

```
[[5]]  
[1] 10
```

If you have more than two lists, you can use `pmap()` instead.

### 2.3.4 Data frames

As mentioned in the introduction of this section, data frames are a special type of list of atomic vectors. This means that just as I can use `lapply()` to compute the square root of the elements of an atomic vector, as in the previous example, I can also operate on all the columns of a data frame. For example, it is possible to determine the class of every variable like this:

```
lapply(iris, class)
```

```
$Sepal.Length  
[1] "numeric"
```

```
$Sepal.Width  
[1] "numeric"
```

```
$Petal.Length  
[1] "numeric"
```

```
$Petal.Width  
[1] "numeric"
```

```
$Species  
[1] "factor"
```

Unlike a list however, the elements of a data frame must be of the same length. Data frames remain very flexible however, and using what we have learned until now, it is possible to use the data frame as a structure for all our computations. For example, suppose that we have a data frame that contains data on unemployment for the different subnational divisions of the Grand-Duchy of Luxembourg, the country the author of this book hails from. Let's suppose that I want to generate several plots, per subnational division and per year. Typically, we would use a loop for this, but we can use what we've learned here, as well as some functions from the `{dplyr}`, `{purrr}`, `{ggplot2}` and `{tidyr}` packages. I will be downloading data

that I made available inside a package, but instead of installing the package, we will download the `.rda` file (which is the file format of packaged data) and then load that data into our R session:

```
# Create a temporary file
unemp_path <- tempfile(fileext = ".rda")

# Download the data and save it to the path of the temporary file
download.file("https://github.com/b-rodrigues/myPackage/raw/main/data/unemp.rda", destfile = unemp_path)

# Load the data. The data is now available as 'unemp'
load(unemp_path)
```

Let's load the required packages and take a look at the data:

```
library(dplyr)
```

Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

`filter`, `lag`

The following objects are masked from 'package:base':

`intersect`, `setdiff`, `setequal`, `union`

```
library(purrr)
library(ggplot2)
library(tidyr)

glimpse(unemp)
```

Rows: 472

Columns: 9

\$ year	<dbl> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, ~
\$ place_name	<chr> "Luxembourg", "Capellen", "Dippach", "Gar~
\$ level	<chr> "Country", "Canton", "Commune", "Commune"~
\$ total_employed_population	<dbl> 223407, 17802, 1703, 844, 1431, 4094, 214~

```

$ of_which_wage_earners      <dbl> 203535, 15993, 1535, 750, 1315, 3800, 187~
$ of_which_non_wage_earners  <dbl> 19872, 1809, 168, 94, 116, 294, 272, 113,~
$ unemployed                 <dbl> 19287, 1071, 114, 25, 74, 261, 98, 45, 66~
$ active_population          <dbl> 242694, 18873, 1817, 869, 1505, 4355, 224~
$ unemployment_rate_in_percent <dbl> 7.947044, 5.674773, 6.274078, 2.876870, 4~

```

Column names are self-descriptive, but the `level` column needs some explanations. The country of Luxembourg is divided into *Cantons*, and these *Cantons* themselves into *Communes*.

You should know that the word *Luxembourg* can refer to the country, the canton or the commune of Luxembourg. Now let's suppose that I want a separate plot for the three communes of Luxembourg, Esch-sur-Alzette and Wiltz. Instead of creating three separate data frames and feeding them to the same ggplot code, I can instead take advantage of the fact that data frames are lists, and are thus quite flexible. Let's start with filtering:

```

filtered_unemp <- unemp %>%
  filter(
    level == "Commune",
    place_name %in% c("Luxembourg", "Esch-sur-Alzette", "Wiltz")
  )

glimpse(filtered_unemp)

```

```

Rows: 12
Columns: 9
$ year      <dbl> 2013, 2013, 2013, 2014, 2014, 2014, 2015,~
$ place_name <chr> "Esch-sur-Alzette", "Luxembourg", "Wiltz"~
$ level      <chr> "Commune", "Commune", "Commune", "Commune~
$ total_employed_population <dbl> 12725, 39513, 2344, 13155, 40768, 2377, 1~
$ of_which_wage_earners      <dbl> 12031, 35531, 2149, 12452, 36661, 2192, 1~
$ of_which_non_wage_earners  <dbl> 694, 3982, 195, 703, 4107, 185, 710, 4140~
$ unemployed                 <dbl> 2054, 3855, 318, 1997, 3836, 315, 2031, 3~
$ active_population          <dbl> 14779, 43368, 2662, 15152, 44604, 2692, 1~
$ unemployment_rate_in_percent <dbl> 13.898099, 8.889043, 11.945905, 13.179778~

```

We are now going to use the fact that data frames are lists, and that lists can hold any type of object. For example, remember this list from before where one of the elements is a data frame, and the second one a formula:

```

list(
  "a" = head(mtcars),
  "b" = ~lm(y ~ x)
)

```

```
)
```

```
$a
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

```
$b
```

```
~lm(y ~ x)
```

`{dplyr}` comes with a function called `group_nest()` which groups the data frame by a variable (such that the next computations will be performed group-wise) and then nests the other columns into a smaller data frame. Let's try it and see what happens:

```
nested_unemp <- filtered_unemp %>%  
  group_nest(place_name)
```

Let's see how this looks like:

```
nested_unemp
```

```
# A tibble: 3 x 2  
  place_name      data  
  <chr>         <list<tibble[,8]>>  
1 Esch-sur-Alzette [4 x 8]  
2 Luxembourg       [4 x 8]  
3 Wiltz            [4 x 8]
```

`nested_unemp` is a new data frame of 3 rows, one per commune (“Esch-sur-Alzette”, “Luxembourg”, “Wiltz”), and of two columns, one for the names of the communes, and the other contains every other variable inside a smaller data frame. So this is a data frame that has one column where each element of that column is itself a data frame. Such a column is called a list-column. This is essentially a list of lists.

Let's now think about this for a moment. If the column titled `data` is a list of data frames, it should be possible to use a function like `map()` or `lapply()` to apply a function on each of these data frames. Remember that `map()` or `lapply()` require a list of elements of whatever



type and a function that accepts objects of this type as input. So this means that we could apply a function that plots the data to each element of the column titled `data`. Since each element of this column is a data frame, this functions needs a data frame as an input. As a first, simple, example to illustrate this, let's suppose that we want to determine the number of rows of each data frame. This is how we would do it:

```
nested_unemp %>%
  mutate(nrows = map(data, nrow))
```

```
# A tibble: 3 x 3
  place_name      data nrows
  <chr>          <list<tibble[,8]>> <list>
1 Esch-sur-Alzette [4 x 8] <int [1]>
2 Luxembourg       [4 x 8] <int [1]>
3 Wiltz            [4 x 8] <int [1]>
```

The new column, titled `nrows` is a list of integers. We can simplify it by converting it directly to an atomic vector of integers by using `map_int()` instead of `map()`:

```
nested_unemp %>%
  mutate(nrows = map_int(data, nrow))
```

```
# A tibble: 3 x 3
  place_name      data nrows
  <chr>          <list<tibble[,8]>> <int>
1 Esch-sur-Alzette [4 x 8]      4
2 Luxembourg       [4 x 8]      4
3 Wiltz            [4 x 8]      4
```

Let's try for a more complex example now. What if we want to filter rows? (The simplest way would of course to filter the rows we need before nesting the data frame). We need to apply the function `filter()` where its first argument is a data frame and the second argument is a predicate:

```
nested_unemp %>%
  mutate(nrows = map(data, \(x)filter(x, year == 2015)))
```

```
# A tibble: 3 x 3
  place_name      data nrows
  <chr>          <list<tibble[,8]>> <list>
```

```
1 Esch-sur-Alzette      [4 x 8] <tibble [1 x 8]>
2 Luxembourg            [4 x 8] <tibble [1 x 8]>
3 Wiltz                  [4 x 8] <tibble [1 x 8]>
```

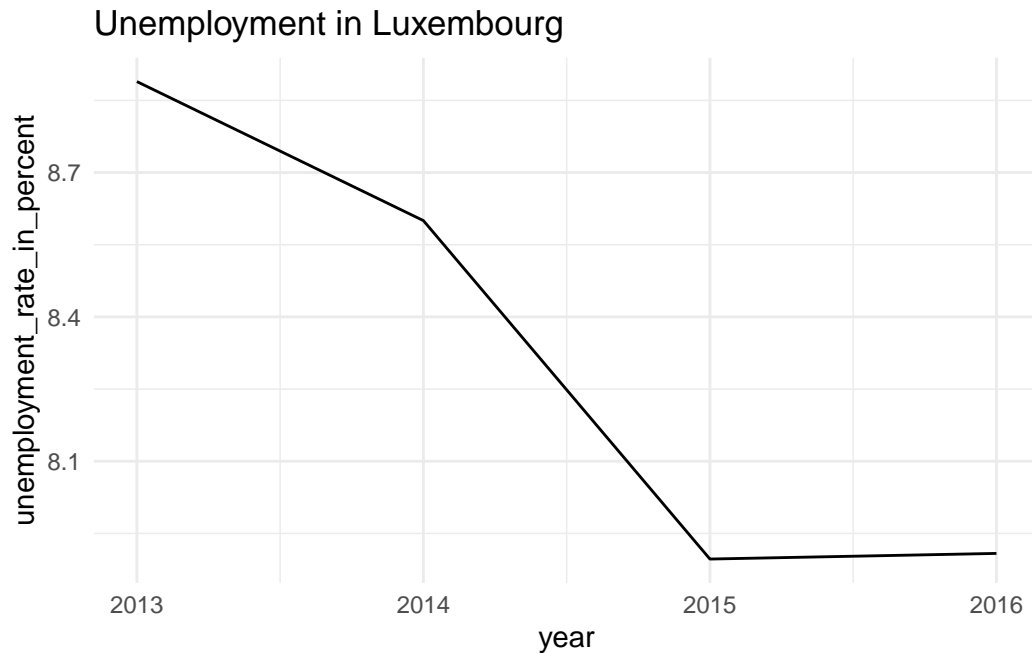
In this case here, we need to use an anonymous function. This is because `filter()` has two arguments, and we need to make clear what it is we are mapping over and what argument stays fixed; we are mapping over, or iterating if you will, data frames, but the predicate stays fixed.

We are now ready to plot our data. The best way to continue is to first get the function right by creating one plot for one single commune. Let's select the dataset for the commune of Luxembourg:

```
lux_data <- nested_unemp %>%
  filter(place_name == "Luxembourg") %>%
  unnest(data)
```

To plot this data, we can now write the required `ggplot2()` code:

```
ggplot(data = lux_data) +
  theme_minimal() +
  geom_line(
    aes(year, unemployment_rate_in_percent, group = 1)
  ) +
  labs(title = "Unemployment in Luxembourg")
```



To turn the lines of code above into a function, you need to think about how many arguments that function would have. There is an obvious one, the data itself (in the snippet above, the data is the `lux_data` object). Another one that is less obvious is in the title:

```
labs(title = "Unemployment in Luxembourg")
```

```
$title  
[1] "Unemployment in Luxembourg"  
  
attr("class")  
[1] "labels"
```

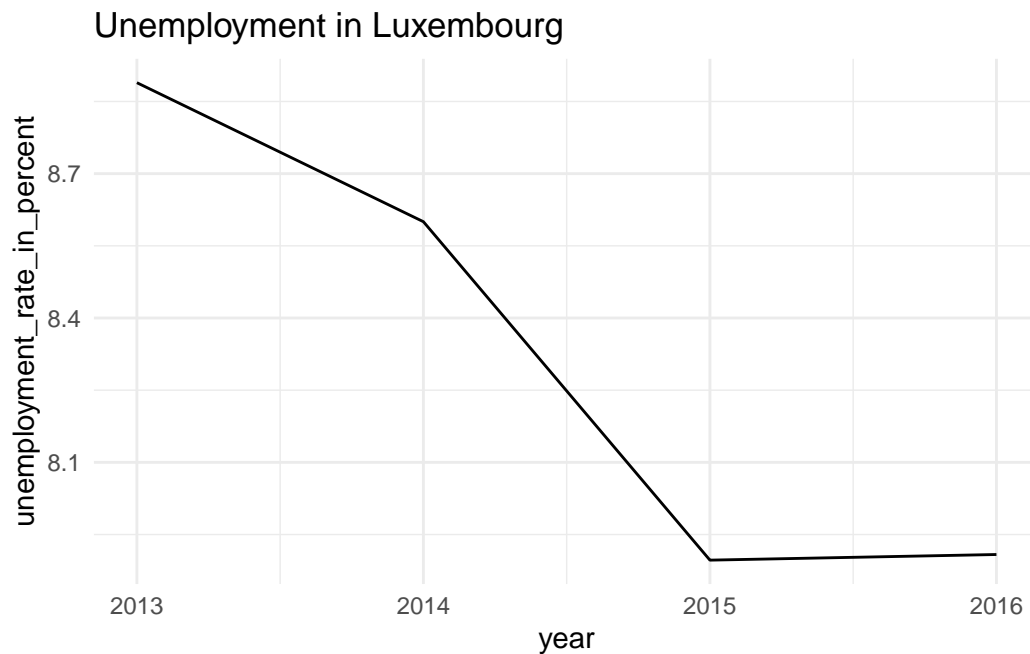
Ideally, we would want that title to change, depending on the data set. So we could write the function like so:

```
make_plot <- function(x, y){  
  ggplot(data = x) +  
    theme_minimal() +  
    geom_line(  
      aes(year, unemployment_rate_in_percent, group = 1)  
    ) +
```

```
  labs(title = paste("Unemployment in", y))
}
```

Let's try it on our data:

```
make_plot(lux_data, "Luxembourg")
```



Ok, so now, we simply need to apply this function to our nested data frame:

```
nested_unemp <- nested_unemp %>%
  mutate(plots = map2(
    .x = data,
    .y = place_name,
    .f = make_plot
  ))
```

```
nested_unemp
```

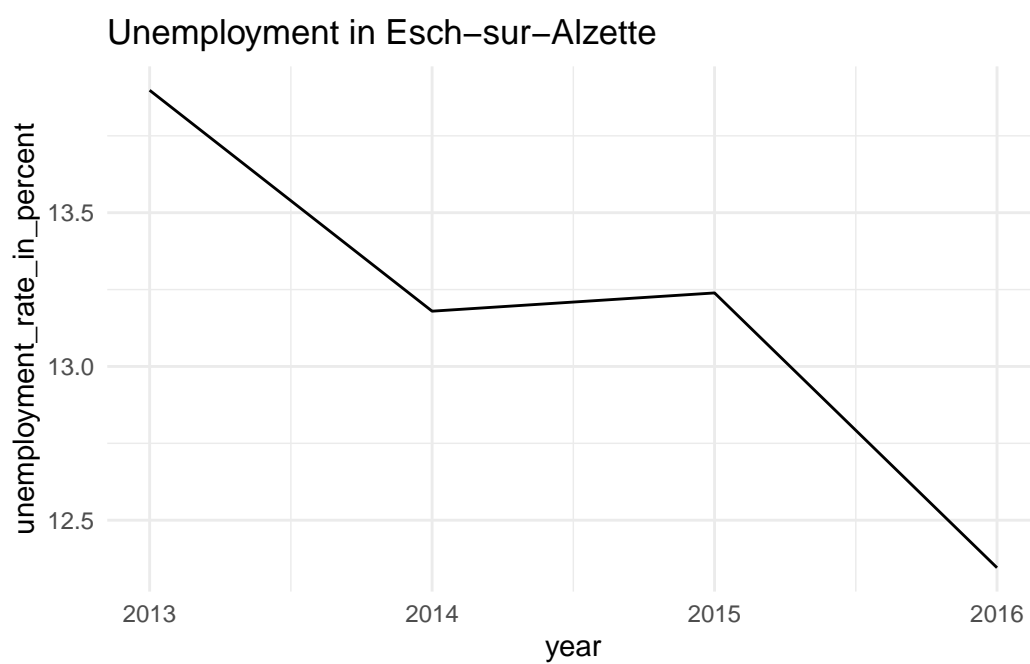
```
# A tibble: 3 x 3
  place_name      data plots
  <chr>          <list<tibble[,8]>> <list>
```

```
1 Esch-sur-Alzette      [4 x 8] <gg>
2 Luxembourg            [4 x 8] <gg>
3 Wiltz                 [4 x 8] <gg>
```

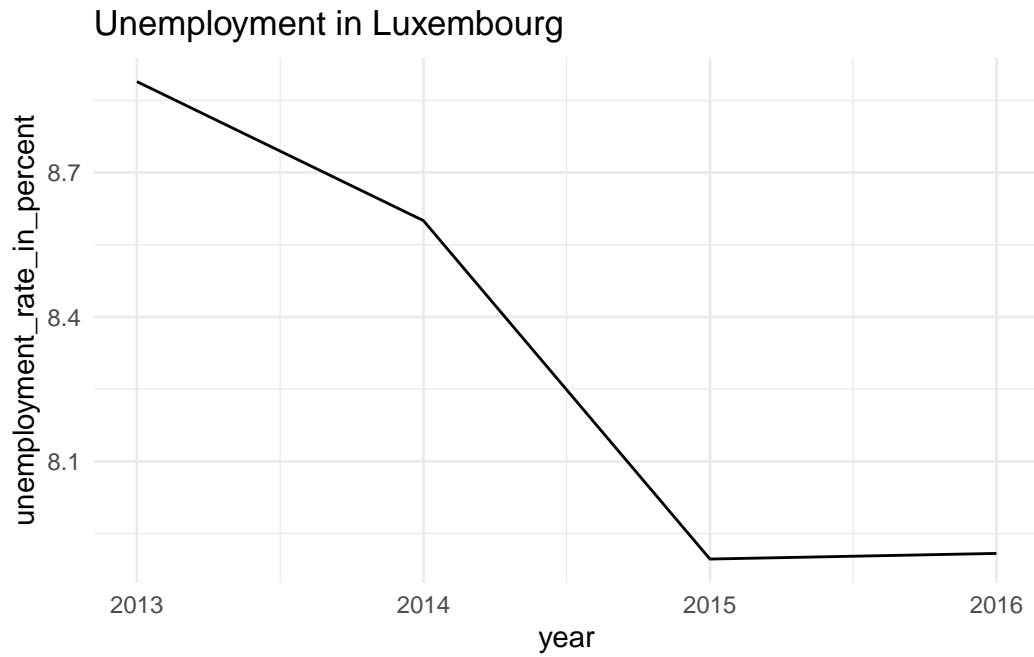
If you look at the `plots` column, you see that it is a list of `gg` objects: these are our plots. Let's take a look at them:

```
nested_unemp$plots
```

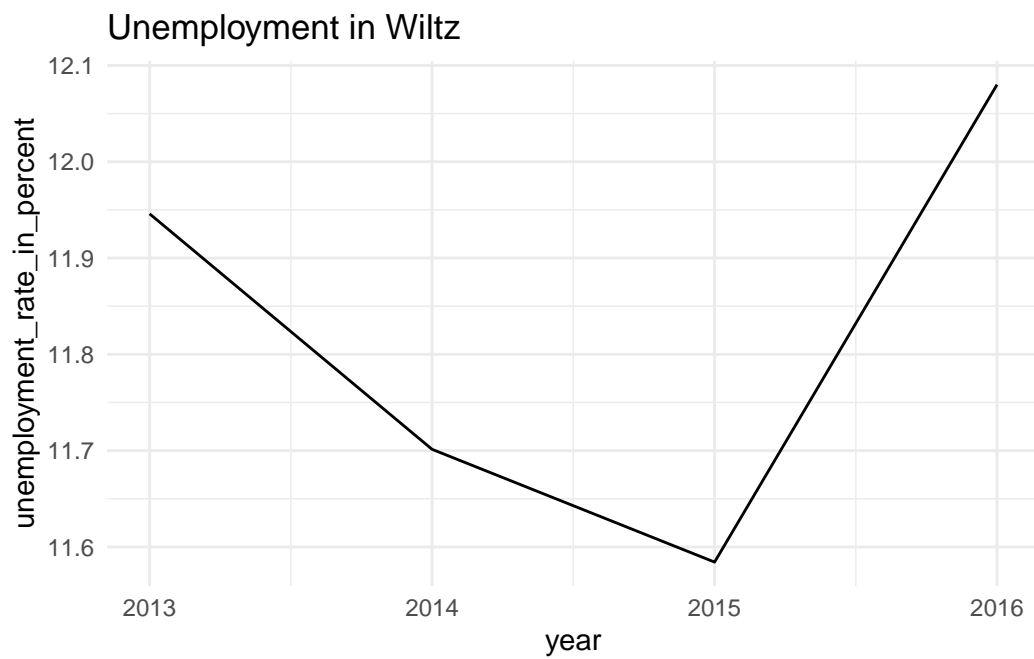
```
[[1]]
```



```
[[2]]
```



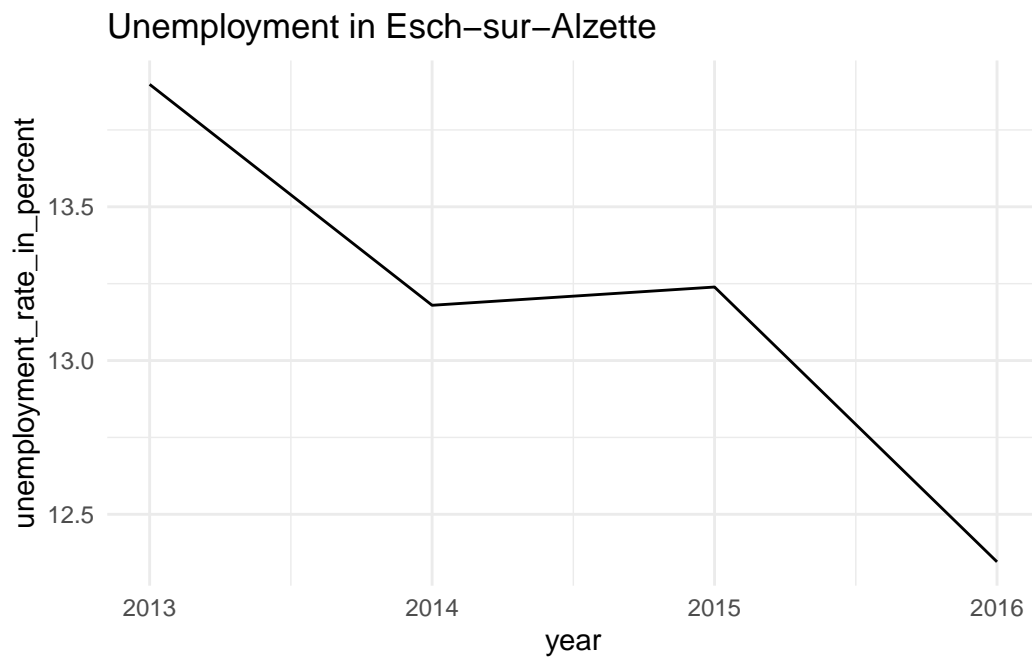
[[3]]



We could also have used an anonymous function:

```
nested_unemp %>%  
  mutate(plots2 = map2(  
    .x = data,  
    .y = place_name,  
    .f = \(.x,.y)(  
      ggplot(data = .x) +  
        theme_minimal() +  
        geom_line(  
          aes(year, unemployment_rate_in_percent, group = 1)  
        ) +  
        labs(title = paste("Unemployment in", .y))  
      )  
    ) %>%  
  pull(plots2)
```

[[1]]



[[2]]



[[3]]





This column-list based workflow is extremely powerful and I highly advise you to take the required time to master it.

## 2.4 Functional programming in R

Up until now we focused more on concepts than on specificities of the R programming language when it comes to functional programming. In the section, we will be focusing entirely on R-specific capabilities and packages for functional programming.

### 2.4.1 Base capabilities

R is a functional programming language, as already stated, and as such it comes with many functions out of the box to write functional code. We have already discussed `lapply()` and `Reduce()`. You should know that depending on what you want to achieve, there are other functions that are similar to `lapply()`: `apply()`, `sapply()`, `vapply()`, `mapply()` and `tapply()`. There's also `Map()` which is a wrapper around `mapply()`. Each function performs the same basic task of applying a function over all the elements of a list or list-like structure, but it can be hard to keep them apart. This is why `{purrr}`, which we will discuss in the next section, is quite an interesting alternative to base R's offering.

Another one of the quintessential functional programming functions (alongside `Reduce()` and `Map()`) that ships with R is `Filter()`. If you know `dplyr::filter()` you should be familiar with the concept of filtering rows of a data frame where the elements of one particular column satisfy a predicate. `Filter()` works the same way, but focusing on lists instead of data frame:

```
Filter(is.character,  
      list(  
        seq(1:5),  
        "Hey")  
      )
```

```
[[1]]  
[1] "Hey"
```

The call above only returns the elements where `is.character()` evaluates to `TRUE`.

Another useful function is `Negate()` which is a function factory that takes a boolean function as an input and returns the opposite boolean function. As an illustration, suppose that in the example above we wanted to get everything *but* the character:

```
Filter(Negate(is.character),
      list(
        seq(1:5),
        "Hey")
      )
```

```
[[1]]
[1] 1 2 3 4 5
```

There are some other functions like this that you might want to check out: type `?Negate` in console to read more about them.

Before continuing with R packages that extend R's functional programming capabilities it's also important to stress that just as R is a functional programming language, it is also an object oriented language. In fact, R is what John Chambers called a *functional OOP* language (Chambers (2014)). We won't delve too much into what this means (read Wickham (2019) for this), but as a short discussion, consider the `print()` function. Depending on what type of object the user gives it, it seems as if somehow `print()` knows what to do with it:

```
print(5)
```

```
[1] 5
```

```
print(head(mtcars))
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

```
print(str(mtcars))
```

```
'data.frame': 32 obs. of 11 variables:
 $ mpg : num 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num 6 6 4 6 8 6 8 4 4 6 ...
```

```

$ disp: num 160 160 108 258 360 ...
$ hp : num 110 110 93 110 175 105 245 62 95 123 ...
$ drat: num 3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
$ wt : num 2.62 2.88 2.32 3.21 3.44 ...
$ qsec: num 16.5 17 18.6 19.4 17 ...
$ vs : num 0 0 1 1 0 1 0 1 1 1 ...
$ am : num 1 1 1 0 0 0 0 0 0 0 ...
$ gear: num 4 4 4 3 3 3 3 4 4 4 ...
$ carb: num 4 4 1 1 2 1 4 2 2 4 ...
NULL

```

The way this works, is essentially a mixture of functional and object oriented programming, so functional OOP. Let's take a closer look at the source code of `print()` by simply typing `print` without brackets, into a console:

```
print
```

```

function (x, ...)
UseMethod("print")
<bytecode: 0x5638394c43c8>
<environment: namespace:base>

```

Quite unexpectedly, the source code of `print()` is one line long and is just `UseMethod("print")`. So all `print()` does is use a generic method called “print”. If your text editor has autocompletion enabled, you might see that there are actually quite a lot of `print()` functions. For example, type `print.data.frame` into a console:

```
print.data.frame
```

```

function (x, ..., digits = NULL, quote = FALSE, right = TRUE,
  row.names = TRUE, max = NULL)
{
  n <- length(row.names(x))
  if (length(x) == 0L) {
    cat(sprintf(ngettext(n, "data frame with 0 columns and %d row",
      "data frame with 0 columns and %d rows"), n), "\n",
      sep = "")
  }
  else if (n == 0L) {
    print.default(names(x), quote = FALSE)
    cat(gettext("<0 rows> (or 0-length row.names)\n"))
  }
}

```

```

}
else {
  if (is.null(max))
    max <- getOption("max.print", 99999L)
  if (!is.finite(max))
    stop("invalid 'max' / getOption(\"max.print\"): ",
         max)
  omit <- (n0 <- max%%length(x)) < n
  m <- as.matrix(format.data.frame(if (omit)
    x[seq_len(n0), , drop = FALSE]
  else x, digits = digits, na.encode = FALSE))
  if (!isTRUE(row.names))
    dimnames(m)[[1L]] <- if (isFALSE(row.names))
      rep.int("", if (omit)
        n0
      else n)
    else row.names
  print(m, ..., quote = quote, right = right, max = max)
  if (omit)
    cat(" [ reached 'max' / getOption(\"max.print\") -- omitted",
        n - n0, "rows ]\n")
}
invisible(x)
}
<bytecode: 0x5638399c60d0>
<environment: namespace:base>

```

This is the `print` function for `data.frame` objects. So what `print()` does is look at the class of its argument `x`, and then look for the right `print` function. In more traditional OOP languages, users would type something like:

```
mtcars.print()
```

In these languages, objects encapsulate method (the equivalent of our functions), so if `mtcars` is a data frame, it encapsulates a `print()` method that then does the printing. R is different, because classes and methods are kept separate. If a package developer creates a new class of object, then the developer also must implement the required methods. For example in the `{chronicler}` package, the `chronicler` class is defined and a `print.chronicler()` function is defined to print these objects.

All of this to say that if you want to extend R by writing packages, learning some OOP essentials is also important. But for data analysis, functional programming does the job perfectly. To learn more about R's different OOP systems (yes, R can do OOP in different

ways and the one I sketched here is the simplest, but probably the most used as well, one), take a look at Wickham (2019).

## 2.4.2 purrr

The `{purrr}` package, developed by Posit, contains many functions to make functional programming with R more smooth. In the previous section, we discussed the `apply()` family of function; they all do a very similar thing, which is looping over a list and applying a function to the elements of the list, but it is not quite easy to remember which one does what. Also, for some of these functions like `apply()`, the list comes first, and then the function, but in the case of `mapply()`, the function comes first. Another issue with these functions is that it is not always easy to know what type the output is going to be. List? Atomic vector? Something else?

`{purrr}` solves this issue by offering the `map()` family of functions, which behave in a very consistent way. The basic function is called `map()` and we've already used it:

```
map(seq(1:5), sqrt)
```

```
[[1]]  
[1] 1  
  
[[2]]  
[1] 1.414214  
  
[[3]]  
[1] 1.732051  
  
[[4]]  
[1] 2  
  
[[5]]  
[1] 2.236068
```

But there are many interesting variants:

```
map_dbl(seq(1:5), sqrt)
```

```
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

`map_dbl()` coerces the output to an atomic vector of doubles instead of a list of doubles. Then there's:

```
map_chr(letters, toupper)
```

```
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"
```

for when the output needs to be an atomic vector of characters.

There are many others, so take a look at the document with `?map`. There's also `walk()` which is used if you're only interested in the side-effect of the function (for example if the function takes paths as input and saves something to disk).

`{purrr}` also has functions to replace `Reduce()`, simply called `reduce()` and `accumulate()`, and there are many, many other useful functions. Read through the [documentation of the package](#) and take the time to learn about all it has to offer.

### 2.4.3 withr

`{withr}` is a powerful package that makes it easy to “purify” functions that behave in a way that can cause problems. Remember the function from the introduction that randomly gave out a recipe Bruno liked? Here it is again:

```
h <- function(name, food_list = list()){
  food <- sample(c("lasagna", "cassoulet", "feijoada"), 1)
  food_list <- append(food_list, food)
  print(paste0(name, " likes ", food))
  food_list
}
```

Because this function returns results that are not consistent for a fixed input, this function is not referentially transparent. So we improved the function like this:

```
h2 <- function(name, food_list = list(), seed = 123){
  # We set the seed, making sure that we get the same selection of food for a given seed
  set.seed(seed)
```

```

food <- sample(c("lasagna", "cassoulet", "feijoada"), 1)

# We now need to unset the seed, because if we don't, guess what, the seed will stay set
set.seed(NULL)

food_list <- append(food_list, food)

print(paste0(name, " likes ", food))

food_list
}

```

The problem with this approach is that we need to modify our function. We can instead use `withr::with_seed()` to achieve the same effect:

```

withr::with_seed(seed = 123,
                 h("Bruno"))

```

```
[1] "Bruno likes feijoada"
```

```
[[1]]
[1] "feijoada"
```

It is also easier to create a wrapper if needed:

```

h3 <- function(..., seed){
  withr::with_seed(seed = seed,
                  h(...))
}

```

```
h3("Bruno", seed = 123)
```

```
[1] "Bruno likes feijoada"
```

```
[[1]]
[1] "feijoada"
```

Before we downloaded a dataset and loaded it into memory; we did so by first created a temporary file, then downloading it and then loading it. Suppose that instead of loading this

data into our session, we simply wanted to test whether the link was still working. We wouldn't want to keep the loaded data in our session, so to avoid having to delete it again manually, we could use `with_tempfile()`:

```
withr::with_tempfile("unemp", {
  download.file("https://github.com/b-rodrigues/myPackage/raw/main/data/unemp.rda", destfile = "unemp.rda")
  load(unemp)
  nrow(unemp)
})
```

[illegible]

[1] 472

The data got downloaded, and then loaded, and then we computed the number of rows of the data, without touching the global environment, or state, of our current session.

Just like for `{purrr}`, `{withr}` has many useful functions which I encourage you to [familiarize yourself with](#).

## 2.5 Conclusion

If there is only one thing that you should remember from this chapter, it would be pure functions. This is in my opinion not very difficult to do and comes with many benefits. But, avoiding loops and replacing them with higher-order functions (`lapply()`, `Reduce()`, `purrr::map()` – and its variants –) also pays off. While this chapter stresses the advantages of functional programming, you should not forget that R is not a pure, and solely, functional programming language and that other paradigms, like object oriented programming, are also available to you, and if your goal is to master the language (instead of “just” using it to solve data analysis problems), then you also need to know about R’s OOP capabilities.



## 3 Version control

What Miles said on the matter:

*There are still a lot of people that find git intimidating and still potential for some things to go badly for a project if git is used in the wrong way. I once had a colleague who assured me they knew how to use git proceed to use a repo like their personal dropbox folder. Perhaps the details of git usage can be basically waved away, but some detail about good git workflow could be incorporated. For example: The branching model to use. IMHO trunk-based development works much better than gitflow for analysis teams. Version number discipline. Why you always bump the version number when making changes to your packages. Why keeping commits small and confined to just one target at a time if possible is useful when tracing problems with a pipeline.*

## 4 Keeping it DRY with literate programming

In order to build a reproducible pipeline, we will be working together on a project. The project itself will be quite simple and will simply consist in getting some data “trapped” inside a Microsoft Excel Workbook into a data frame, and then creating a plot. The point of this book is not to teach you data analysis no data visualization, but instead, engineering the project in such a way that it is:

- Documented;
- Tested;
- Reproducible.

We will start our project like someone with time constraints does, and see how we can reach our goal of having a nice reproducible project as painlessly as possible. There is only one catch; to make the whole process as easy as possible, we will be using literate programming, and not simply programming.

### 4.1 Literate programming

In literate programming, we mix code and prose together, in a way that makes it easy for both non-technical users and programmers to understand what is going on. Scripts written this way are also very easy to compile, or render, into a variety of document formats like `html` and `docx`. R supports several ways of doing literate programming: Sweave, knitr and Quarto.

Sweave was the first tool available to R (and S) users, and allowed the mixing of R and LaTeX code to create a document. Friedrich Leisch developed Sweave in 2002 and described it in his 2002 paper (Leisch (2002)). As Leisch argues, *the traditional way of writing a report as part of a statistical data analysis project uses two separate steps*: running the analysis using some software, and then copy and pasting the results into a word processing tool. The problem with this approach is that much time is wasted copy and pasting things, so experimenting with different layouts or data analysis techniques is very time consuming. Copy and paste mistakes will also happen (it’s not a question of if, but when) and updating reports (for example, when new data comes in) means that someone will have, again, to copy and paste the updated results into a new document.

Sweave provided (and still provides, as it is still well functioning!) a way to embed the analysis in the document itself, in this case a LaTeX source file, and R code was executed whenever

the document was compiled. This gave researchers considerable time savings when it was time to update a report or drafting a research paper.

The snippet below shows the example from Leisch's paper:

```
\documentclass[a4paper]{article}
```

```
\begin{document}
```

In this example we embed parts of the examples from the `\texttt{kruskal.test}` help page into a LaTeX document:

```
<<>>=
data (airquality)
kruskal.test(Ozone ~ Month, data = airquality)
@
```

which shows that the location parameter of the Ozone distribution varies significantly from month to month. Finally we include a boxplot of the data:

```
\begin{center}
<<fig=TRUE,echo=FALSE>>=
boxplot(Ozone ~ Month, data = airquality)
@
\end{center}

\end{document}
```

Even if you've never seen a LaTeX source file, you should be able to figure out what's going on. The first line states what type of document we're writing. Then comes `\begin{document}` which tells the compiler where the document starts. Then comes the content. You can see that it's a mixture of plain English with R code defined inside chunks starting with `<<>>=` and ending with `@`. Finally, the document ends with `\end{document}`. Getting a human readable PDF from this source is a two-step process: first this source gets converted into a `.tex` file and then this `.tex` file into a PDF. Sweave is included with every R installation since version 1.5.0, and still works to this day. For example, we can test that our Sweave installation works just fine by compiling the example above. This is what the final output looks like:

Let us just state that the fact that it is still possible to compile this example more than 20 years later is an incredible testament to how mature and stable this software is (both R, Sweave, and LaTeX). But as impressive as this is, LaTeX has a steep learning curve, and Leisch even advocated the use of the Emacs text editor to edit Sweave files, which also has a very steep

In this example we embed parts of the examples from the `kruskal.test` help page into a  $\text{\LaTeX}$  document:

```
> data (airquality)
> kruskal.test(Ozone ~ Month, data = airquality)
```

Kruskal-Wallis rank sum test

data: Ozone by Month

Kruskal-Wallis chi-squared = 29.267, df = 4, p-value = 6.901e-06

which shows that the location parameter of the Ozone distribution varies significantly from month to month. Finally we include a boxplot of the data:

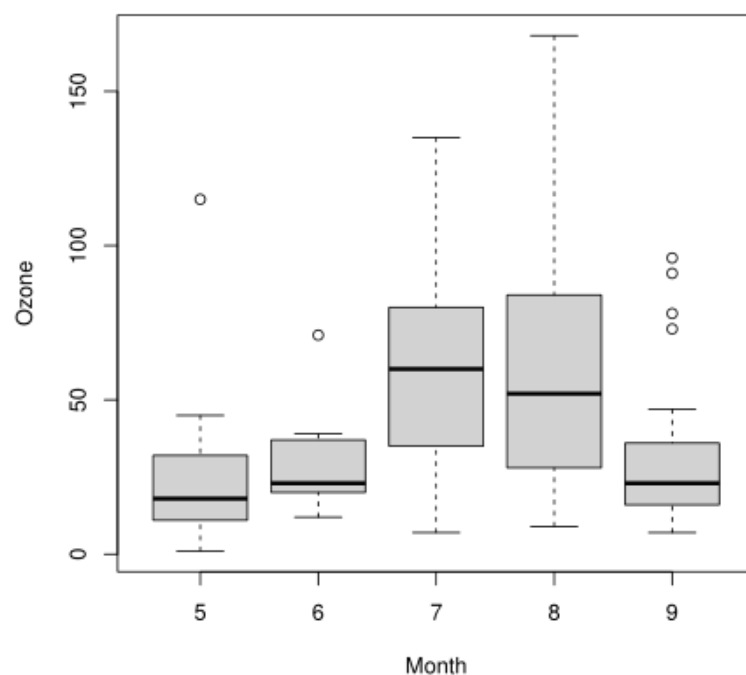


Figure 4.1: More than 20 years later, the output is still the same

learning curve (but this is entirely optional; for example we edited and compiled the example on the Rstudio IDE).

The next generation of literate programming tools was provided by a package called `{knitr}` in 2012. From the perspective of the user, the biggest change from Sweave is that `{knitr}` is able to use many different formats as source files. The one that became very likely the most widely used format is a flavour of the Markdown markup language, R Markdown (Rmd). But `{knitr}` can also run code chunks for other languages, such as Python, Perl, Awk, Haskell, bash and more (Xie (2014)). Since version 1.18, `{knitr}` uses the `{reticulate}` package to provide a Python engine for the Rmd format. To illustrate the Rmd format, let's rewrite the example from Leisch's Sweave paper into it:

```
---
output: pdf_document
---

In this example we embed parts of the examples from the
\texttt{kruskal.test} help page into a LaTeX document:

```{r}
data (airquality)
kruskal.test(Ozone ~ Month, data = airquality)
```

which shows that the location parameter of the Ozone
distribution varies significantly from month to month.
Finally we include a boxplot of the data:

```{r, echo = FALSE}
boxplot(Ozone ~ Month, data = airquality)
```
```

This is what the output looks like:

Just like in a Sweave document, an Rmd source file also has a header in which authors can define a number of options. Here we only specified that we wanted a pdf document as an output file. We then copy and pasted the contents from the Sweave source, but changed the chunk delimiters from `<<>=` and `@` to ````{r}` to start an R chunk and ````` to end it. Remember; we need to specify the engine in the chunk because `{knitr}` supports many engines. For example, it is possible to run a bash command by adding this chunk to the source:

In this example we embed parts of the examples from the `kruskal.test` help page into a  $\text{\LaTeX}$  document:

```
data (airquality)
kruskal.test(Ozone ~ Month, data = airquality)
```

```
##
## Kruskal-Wallis rank sum test
##
## data:  Ozone by Month
## Kruskal-Wallis chi-squared = 29.267, df = 4, p-value = 6.901e-06
```

which shows that the location parameter of the Ozone distribution varies significantly from month to month. Finally we include a boxplot of the data:

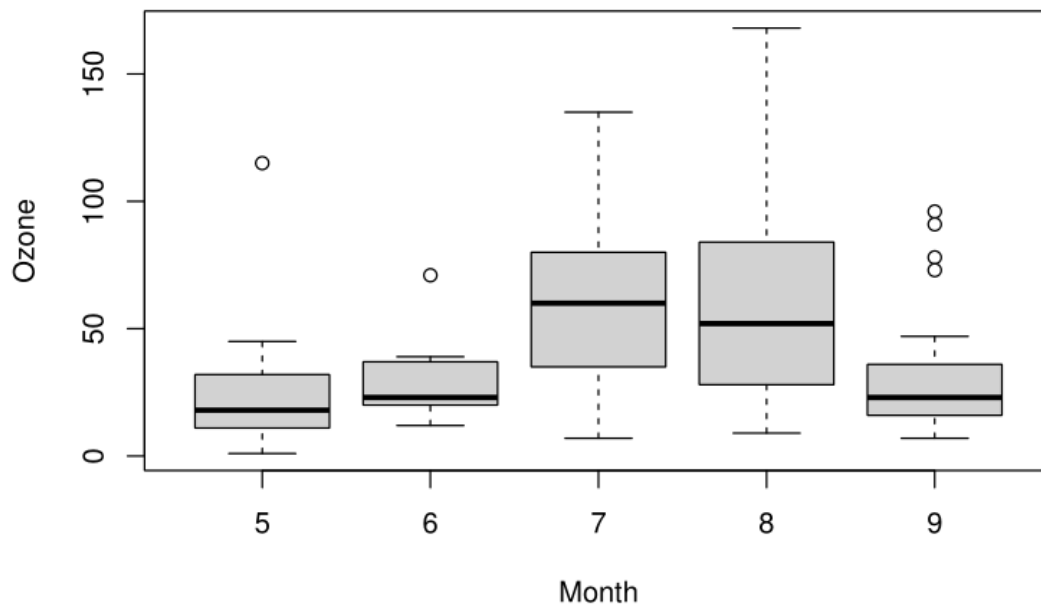


Figure 4.2: It's very close to the Sweave output

```
---
output: pdf_document
---
```

In this example we embed parts of the examples from the `\texttt{kruskal.test}` help page into a LaTeX document:

```
```{r}
data (airquality)
kruskal.test(Ozone ~ Month, data = airquality)
```
```

which shows that the location parameter of the Ozone distribution varies significantly from month to month. Finally we include a boxplot of the data:

```
```{r, echo = FALSE}
boxplot(Ozone ~ Month, data = airquality)
```
```

```
```{bash}
pwd
```
```

(bash's `pwd` command shows the current working directory). You may have noticed that we also keep two LaTeX commands in the source Rmd, `\texttt{}` and `LaTeX`. This is because Rmd files get first converted into LaTeX files and then into a PDF. If you're using RStudio, this document can be compiled by clicking a button or using a keyboard shortcut, but you can also use the `rmarkdown::render()` function. This function does two things transparently: it first converts the Rmd file into a source LaTeX file, and then converts it into a PDF. It is of course possible to convert the document to a Word document as well however, LaTeX commands will be ignored. Html is another widely used output format.

If you're a researcher and prefer working with LaTeX directly instead of having to switch to Markdown, you can either use Sweave, or use `{knitr}` but instead of writing your documents using the R Markdown format, you can use the `Rnw` format which is basically the same as Sweave, but uses `{knitr}` for compilation. Take a look at [this example](#) from the `{knitr}` github repository for example.

You should know that `{knitr}` makes it possible to author many, many different types of documents. It is possible to write books, blogs, package documentation (and even entire packages, as we shall see later in this book), Powerpoint slides... It is extremely powerful because we can use the same general R Markdown knowledge to build many different outputs:

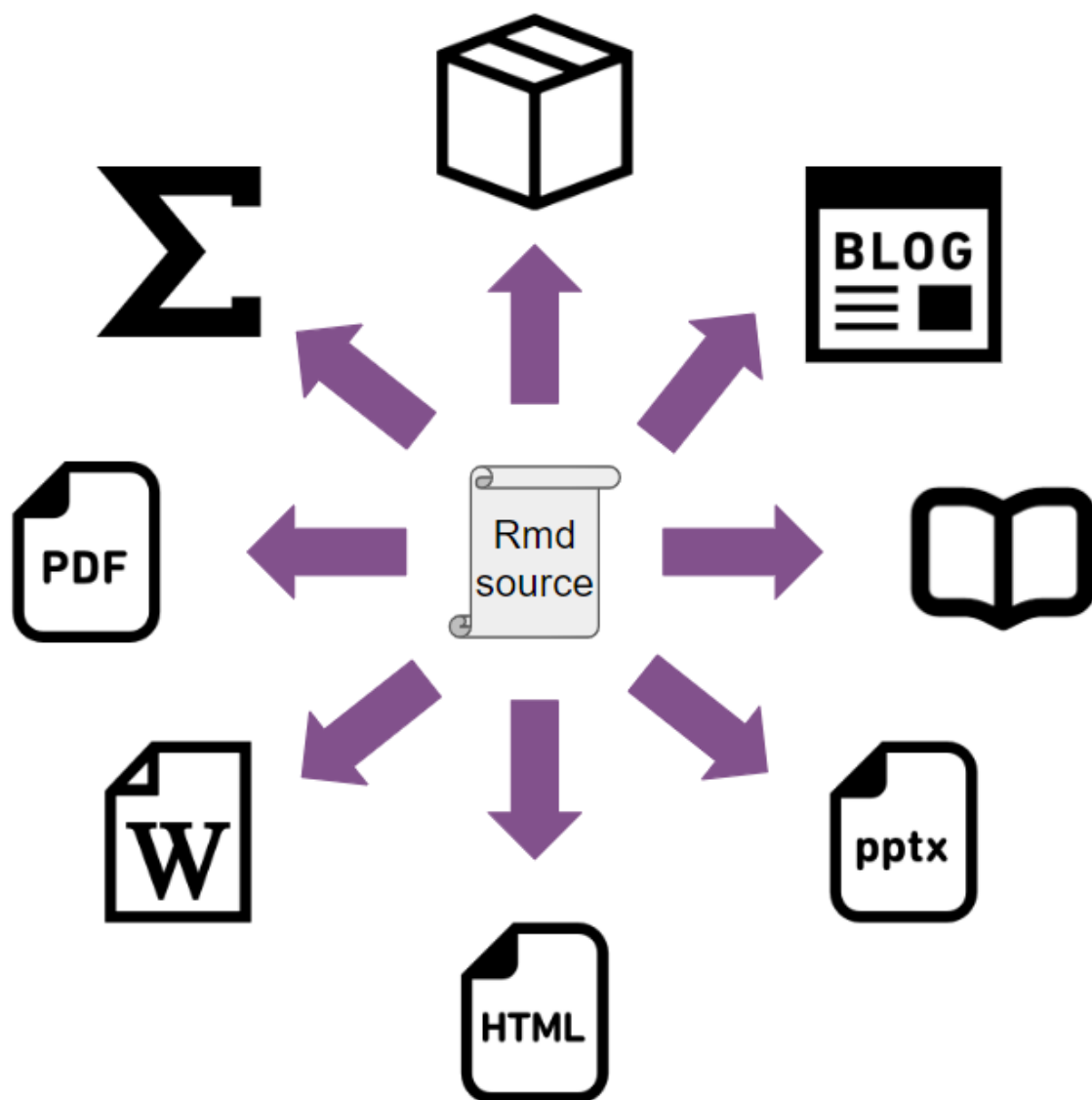


Figure 4.3: One format to rule them all



Finally, the latest in literate programming for R is a new tool developed by Posit, called Quarto. If you're an R user and already know `{knitr}` and the Rmd format, you should be able to immediately use Quarto. So what's the difference? In practice and for R users not much. There are many things that Quarto is able to do out of the box that you can't, without extensions, do with `{knitr}`. Quarto has some nice defaults; in fact this book is written in Quarto instead of `{knitr}` because the default Quarto output looks nicer than the default `{knitr}` output. However, there may even be things that Quarto can't do at all (at least for now) when compared to `{knitr}`. So why bother switching? Well, Quarto provides sane defaults and some nice features out of the box, and the cost of switching from the Rmd format to Quarto's Qmd format is basically 0. Also, and this is probably the biggest reason to use Quarto, is that Quarto is not tied to R. Quarto is actually a standalone tool that needs to be installed alongside your R installation, and works completely independently. In fact, you can use Quarto without having R installed at all, as Quarto, just like `{knitr}` supports many engines. This means that if you're primarily using Python, you can author documents with Quarto. Quarto also supports the Julia programming language and Observable JS, making it possible to include interactive visualisations into an Html document. Let's take a look at how the example from Leisch's paper looks as a Qmd file:

```
---
output: pdf
---

In this example we embed parts of the examples from the
\texttt{kruskal.test} help page into a LaTeX document:

```{r}
data (airquality)
kruskal.test(Ozone ~ Month, data = airquality)
```

which shows that the location parameter of the Ozone
distribution varies significantly from month to month.
Finally we include a boxplot of the data:

```{r, echo = FALSE}
boxplot(Ozone ~ Month, data = airquality)
```
```

(I've omitted the bash chunk from before, not because Quarto does not support it, but to keep close to the original example from the paper.)

As you can see, it's exactly the same as the Rmd file from before. The only difference is in the header. In the Rmd file we specified the output format as:

```
---  
output: pdf_document  
---
```

whereas in the Qmd file we changed it to:

```
---  
output: pdf  
---
```

While Quarto is the latest option in literate programming, it is quite recent, and as such, we feel it might be better to stick with `{knitr}` and the Rmd format for now, so that's what we're going to use going forward. Also, the `{knitr}` and the Rmd format are here to stay, so there's little risk in keeping using it, and anyways, as already stated, if switching to Quarto becomes a necessity, the cost of switch would be very, very low. In what follows, we won't be focused on anything really `{knitr}` or Rmd specific, so should you want to use Quarto instead, you should be able to follow along without any problems at all, since the Rmd and Qmd formats have so much overlap (but whenever there would be major differences between the two formats, we will highlight them).

## 4.2 `{knitr}` basics

This section will be a very small intro to `{knitr}`. We are going to teach you just enough to get started, and we are going to focus on the Rmd format. There are many resources out there that you can use if you want to dig deeper, for instance the [R Markdown website](#) from Posit, or the [R Markdown: The Definitive Guide](#) and [R Markdown Cookbook](#) eBooks. We will also not assume that you are using the RStudio IDE and give you instead the lower level commands to render documents. If you use RStudio and want to know how to use it effectively to author Rmd documents, you should take a look at [Quick Tour](#) page. In fact, this section will basically focus on the same topics, but without focusing on how to use RStudio.

### 4.2.1 Set up

The first step is to install the `{knitr}` and the `{rmarkdown}` packages. Simple type:

```
install.packages("rmarkdown")
```

in an R console. Since `{knitr}` is required to install `{rmarkdown}`, it gets installed automatically. If you want to compile PDF documents, you should also have a working LaTeX distribution. You can skip this next part if you're only interested in generating PDF and Word files. For what follows, we will only be rendering Html documents, so no need to install LaTeX (by the way, you do not need a working Word installation to compile documents to the docx format). However, if you already have a working LaTeX installation, you shouldn't have to do anything else to generate PDF documents. If you don't have a working LaTeX distribution, then Yihui Xie, the creator of `{knitr}` created an R package called `{tinytex}` that makes it extremely easy to install a LaTeX distribution. In fact, this is the way we recommend installing LaTeX even if you're not an R user (it is possible to use the tinytex distribution without R; it's just that the `{tinytex}` R package provides many functions that makes installing and maintaining it very easy). Simply run these commands in an R console to get started:

```
install.packages("tinytex")
tinytex::install_tinytex()
```

and that's it! If you need to install specific LaTeX packages, then refer to the **Maintenance** section on [tinytex's](#) website. For example, to compile the example from Leisch's article on Sweave discussed previously, we had to install the `grfext` LaTeX package (as explained by the error output in the console when we tried compiling). So, we simply needed to run the following command to get it:

```
tlmgr_install("grfext")
```

After you've installed `{knitr}`, `{rmarkdown}` and, optionally, `{tinytex}`, simply try to compile the following document:

```
---
output: html_document
---

# Document title

## Section title
```

```
### Subsection title
```

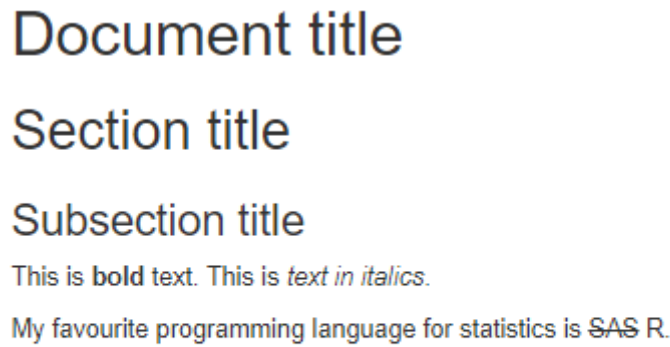
```
This is bold text. This is text in italics.
```

```
My favourite programming language for statistics is SAS R.
```

save this document into a file called `rmd_intro.rmd` using you're favourite text editor. Then render it into an Html file by running the following command in the R console:

```
rmarkdown::render("path/to/rmd_test.rmd")
```

This should create a file called `rmd_test.html`; open it with your web browser and you should see the following:



Document title

Section title

Subsection title

This is **bold** text. This is *text in italics*.

My favourite programming language for statistics is ~~SAS~~ R.

Figure 4.4: It's very close to the Sweave output

Congratulations, you just *knitted* your first Rmd document!

#### 4.2.2 Markdown ultrabasics

R Markdown is a flavour of Markdown, which means that you should know some Markdown to really take full advantage of R Markdown. The document from before should have already shown you some basics: titles, sections and subsections all start with a `#` and the depth level is determined by the number of `#`s. For bold text, simply put the words in between `**` and for italics use only one `*`. If you want ***bold and italics***, use `***`. The original designer of Markdown did not think that underlining text was important, so there is no *easy* way of doing it unfortunately. For this, you need to use a somewhat hidden feature; without going into

too much technical details, the program that converts Rmd files to the final output format is called Pandoc, and it's possible to use some of Pandoc's features to format text. For example, for underlining:

```
[This is some underlined text in a R Markdown document]{.underline}
```

This will underline the text between square brackets.<sup>1</sup>

The next step is actually to mix code and prose. As you've seen from Leisch's canonical example, this is quite easily achieved by using R code chunks. The R Markdown example below shows various code chunks alongside some options. For example, a code chunk that uses the `echo = FALSE` option will not appear (but the output of the computation will):

```
---
title: "Document title"
output: html_document
date: "2023-01-28"
---
```

```
# R code chunks
```

```
This below is an R code chunk:
```

```
```${r}
data(mtcars)

plot(mtcars)
```
```

```
The code chunk above will appear in the final output. The code chunk below will be hidden:
```

```
```${r, echo = FALSE}
data(iris)

plot(iris)
```
```

---

<sup>1</sup><https://stackoverflow.com/a/68690065/1298051>

This next code chunk will not be evaluated:

```
```{r, eval = FALSE}
data(titanic)

str(titanic)
```
```

The last one runs, but code and output from the code is not shown in the final document. This is useful for loading libraries and hiding startup messages:

```
```{r, include = FALSE}
library(dplyr)
```
```

If you use RStudio and create a new R Markdown file from the menu, a new R Markdown file is generated for you to fill out. The first R chunk is this one:

```
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```
```

This is an R chunk named `setup` and with the option `include = FALSE`. Naming chunks is optional, but we are going to make use of this later on. What this chunk runs is one line of code that defines a global option to show all chunks by default (which is the default behaviour). You can change `TRUE` to `FALSE` if you want to hide every code chunk instead (if you're using Quarto, global options are set [differently](#)).

Something else you might have noticed in the previous example, is that we've added some more content in the header:

```
---
title: "Document title"
output: html_document
date: "2023-01-28"
---
```

There are several other options available that you can define in the header. Later on, when we'll be building our project together, we will provide some more options (like having a table of contents).

The finish this part on code chunks, you should know about inline code chunks. Take a look

at the following example:

```
---
title: "Document title"
output: html_document
date: "2023-01-28"
---

# R code chunks

```${r, echo = FALSE}
data(iris)
```

The iris dataset has `${r nrow(iris)}` rows.
```

The last sentence from this example has an inline code chunk. This is quite useful, as it allows to parametrize sentences and paragraphs, and thus avoids needing to copy and paste (and we will go quite far into how to avoid copy and pasting, thanks to templating and parametrized reports).

To finish this crash course, you should know that to use footnotes you need to write the following:

```
This sentence has a footnote.[^1]

[^1]: This is the footnote.
```

and that you can write LaTeX formulas as well. For example, add the following into the example from before and render either a PDF or a html document (don't put the LaTeX formula below inside a chunk, simply paste it as if it were normal text. This doesn't work for Word output because Word does not support LaTeX equations):

```
\begin{align*}
S(\omega)
&= \frac{\alpha g^2}{\omega^5} e^{\left[-0.74\mathrm{bigl}\left\{\frac{\omega}{U}\right\}g\right]\mathrm{bigr}}\{-\frac{\alpha g^2}{\omega^5} \exp\mathrm{Bigl}\left[-0.74\mathrm{Bigl}\left\{\frac{\omega}{U}\right\}g\right]\mathrm{Bigr}\}
\end{align*}
```

## 4.3 Keeping it DRY

We're now coming to why this chapter is titled as it is.

Whatever you're doing, you should keep it DRY - DRY stands for *don't repeat yourself*. Repeating yourself by, say, copy and pasting, leads to errors and makes reading and understanding your code more difficult. This can be avoided by using functions, as we discussed in the previous chapter, but what if you need to write a document that has the following structure:

- A title
- A section
- A table inside this section
- Another section
- Another table inside this section
- Yet another section
- Yet another table inside this section

Is there a way to automate the creation of such a document by taking advantage of the repeating structure? Of course there is. The question is not, *is it possible to do X?*, but *how to do X?*.

### 4.3.1 Generating R Markdown code from code

The example below is a fully working minimal example of this. Copy it inside a document titled something like `rmd_templating.Rmd` and render it. You will see that the output contains more sections than defined in the source. This is because we use templating at the end. Take some time to read the document, as the text inside explains what is going on:

```
---
title: "Templating"
output: html_document
date: "2023-01-27"
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

## A function that creates tables
```



```
```{r}
create_table <- function(dataset, var){
  table(dataset[var]) |>
    knitr::kable()
}
```
```

The function above uses the `table()` function to create frequency tables, and then this gets passed to the `knitr::kable()` function that produces a good looking table for our rendered document:

```
```{r}
create_table(mtcars, "am")
```
```

Let' suppose that we want to generate a document that would look like this:

- first a section title, with the name of the variable of interest
- then the table

So it would look like this:

```
## Frequency table for variable: "am"
```

```
```{r}
create_table(mtcars, "am")
```
```

We don't want to create these sections for every variable by hand.

Instead, we can define a function that returns the R markdown code required to create this. This is this function:

```
```{r}
return_section <- function(dataset, var){
```

```

a <- knitr::knit_expand(text = c("## Frequency table for variable: {{variable}}",
                                create_table(dataset, var)),
                        variable = var)
cat(a, sep = "\n")
}
...

```

This new function, `return_section()` uses `knitr::knit_expand()` to generate R Markdown code. Words between `{{}}` get replaced by the provided `var` argument to the function. So when we call `return_section("am")`, `{{variable}}` is replaced by `"am"`. `"am"` then gets passed down to `create_table()` and the frequency table gets generated. We can now generate all the section by simply applying our function to a list of column names:

```

```{r, results = "asis"}
invisible(lapply(colnames(mtcars), return_section, dataset = mtcars))
```

```

The last function, named `return_section()` uses `knitr::knit_expand()`, which is the one that does the heavy lifting. This function returns literal R Markdown code. It returns `## Frequency table for variable: {{variable}}` which creates a level 2 section title with the text *Frequency table for variable: xxx* where the `xxx` will get replaced by the variable passed to `return_section()`. So calling `return_section(mtcars, "am")` will print the following in your console:

```
## Frequency table for variable: am
am	Freq
0	19
1	13

```

We now simply need to find a clever way to apply this function to each variable in the `mtcars` dataset. For this, we are going to use `lapply()` which implements a for loop (you could use `purrr::map()` just as well for this):

```

invisible(lapply(colnames(mtcars),
                 return_section,
                 dataset = mtcars))

```

This will create, for each variable in `mtcars`, the same R Markdown code as above. Notice that the R Markdown chunk where the call to `lapply()` is has the option `results = "asis"`. This is because the function returns literal Markdown code, and we don't want the parser to have to parse it again. We tell the parser "don't worry about this bit of code, it's already good". As you see, the call to `lapply()` is wrapped inside `invisible()`. This is because `return_section()` does not return anything, it just prints something to the console. No object is returned. So if you don't wrap the call to `lapply()` inside `invisible()`, then a bunch of NULLs will also get printed. To avoid this, use `invisible()` (and use `purrr::walk()` rather than `purrr::map()`).

Click [here](#) to see the output.

This is not an easy topic, so take the time to play around with the example above. Try to print a plot instead of a table, try to generate more complex Markdown code, remove the call to `invisible()` and knit the document and see what happens with the output, replace the call to `lapply()` with `purrr::walk()` or `purrr::map()`. Really take the time to understand what is going on.

### 4.3.2 Tables in R Markdown documents

Getting tables right in Rmd documents is not always an easy task. There are several packages specifically made just for this task.

In this short section, we want to point you towards two packages that check the following boxes:

- Work the same way regardless of output format we want to knit our document into:
- Work for any type of table: summary tables, regression tables, two-way tables, etc.

Let's start with the simplest type of table, which would be showing the head of a dataset for example. `{knitr}` comes with the `kable()` function, but this function generates a very plain looking output. For something publication-worthy, we recommend the `{flextable}` package, developed by Gohel and Skintzos (2023):

```
library(flextable)

my_table <- head(mtcars)

flextable(my_table) |>
  set_caption(caption = "Head of the mtcars dataset") |>
  theme_booktabs()
```

| mpg  | cyl | disp | hp  | drat | wt    | qsec  | vs | am | gear |
|------|-----|------|-----|------|-------|-------|----|----|------|
| 21.0 | 6   | 160  | 110 | 3.90 | 2.620 | 16.46 | 0  | 1  | 4    |
| 21.0 | 6   | 160  | 110 | 3.90 | 2.875 | 17.02 | 0  | 1  | 4    |
| 22.8 | 4   | 108  | 93  | 3.85 | 2.320 | 18.61 | 1  | 1  | 4    |
| 21.4 | 6   | 258  | 110 | 3.08 | 3.215 | 19.44 | 1  | 0  | 4    |
| 18.7 | 8   | 360  | 175 | 3.15 | 3.440 | 17.02 | 0  | 0  | 3    |
| 18.1 | 6   | 225  | 105 | 2.76 | 3.460 | 20.22 | 1  | 0  | 4    |

We won't go into much detail on how `{flextable}` works, but it is very powerful, and the fact that it works for PDF, Html, Word and Powerpoint outputs is really a massive plus. If you want to learn more about `{flextable}`, there's a [whole, free, ebook on it](#). `{flextable}` can create very complicated tables, so really take the time to dig in!

The next package is `{modelsummary}`, by Arel-Bundock (2022), and this one focuses on regression and summary tables. It is extremely powerful as well, and just like `{flextable}`, works for any type of output. It is very simple to get started:

```
library(modelsummary)

model_1 <- lm(mpg ~ hp + am, data = mtcars)
model_2 <- lm(mpg ~ hp, data = mtcars)

models <- list("Model 1" = model_1,
               "Model 2" = model_2)

modelsummary(models)
```

Here again, we won't go into much detail, but recommend instead that you read the package's [website](#) which has very detailed documentation.

These packages can help you keeping it DRY, so take some time to learn them.

### 4.3.3 Parametrized reports

Templating is very powerful, but sometimes you don't want to have one section dedicated to each unit of analysis within the same report, but rather, you want a complete separate report by unit of analysis. This is also possible thanks to parametrized reports.

|             | Model 1           | Model 2           |
|-------------|-------------------|-------------------|
| (Intercept) | 26.585<br>(1.425) | 30.099<br>(1.634) |
| hp          | −0.059<br>(0.008) | −0.068<br>(0.010) |
| am          | 5.277<br>(1.080)  |                   |
| Num.Obs.    | 32                | 32                |
| R2          | 0.782             | 0.602             |
| R2 Adj.     | 0.767             | 0.589             |
| AIC         | 164.0             | 181.2             |
| BIC         | 169.9             | 185.6             |
| Log.Lik.    | −78.003           | −87.619           |
| RMSE        | 2.77              | 3.74              |

Let's modify the example from before, which consisted in creating one section per column of the `mtcars` dataset and a frequency table, and make it now one separate report for each column. The R Markdown file will look like this:

```
---
title: "Report for column `r params$var` of dataset `r params$dataset`"
output: html_document
date: "2023-01-27"
params:
  dataset: mtcars
  var: "am"
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

## Frequency table for `r params$var`

```{r, echo = F}
create_table <- function(dataset, var){

  dataset <- get(dataset)

  table(dataset[var]) |>
```

```

      knitr::kable()
    }
    ...

The table below is for variable `r params$var` of dataset `r params$dataset`.

```{r}
create_table(params$dataset, params$var)
...

```{r, eval = FALSE, echo = FALSE}
# Run these lines to compile the document
# Set eval and echo to FALSE, so that this does not appear
# in the output, and does not get evaluated when knitting
rmarkdown::render(
  input = "param_report_example.Rmd",
  params = list(
    dataset = "mtcars",
    var = "cyl"
  )
)
...

```

Save the code above into an Rmd file title something like “param\_report\_example.Rmd” (preferably inside its own folder). At the end of the document, we wrote the lines to render this document inside a chunk that does not get shown to the reader, and does not get evaluated:

```

rmarkdown::render(
  input = "param_report_example.Rmd",
  params = list(
    dataset = "mtcars",
    var = "cyl"
  )
)

```

You need to run these lines yourself to knit the document.

This will pass the list `params` with elements “mtcars” and “cyl” down to the report. Every `params$dataset` and `params$var` in the report gets replaced by “mtcars” and “cyl” respectively. Also, notice that in the header of the document, we defined default values for the params. Something else you need to be aware of, is that the function `create_table()` inside the report is slightly different than before. It now starts with the following line:

```
dataset <- get(dataset)
```

Let's break this down. `params$dataset` contains the string "mtcars". I made the decision to pass the dataset as a string, so that I could use it in the title of the document. But then, inside the `create_table()` function, I have the following code:

```
dataset[var]
```

`dataset` can't be a string here, but needs to be a variable name, so `mtcars` and not "mtcars". This means that I need to *convert* that string into a name. `get()` searches an object by name, and then makes it possible to save it to a new variable called `dataset`. The rest of the function is then the same as before. This little difficulty can be avoided by hardcoding the dataset inside the R Markdown file, or by passing the dataset as the `params$dataset` and not the string, in the render function. However, if you pass down the name of the dataset as a variable instead of the dataset name as a string, then you need to convert it to a string if you want to use it in the text (so `mtcars` to "mtcars").

If you now want to create one report per variable, you could compile all the documents at once with:

```
columns <- colnames(mtcars)

lapply(columns,
  \(x)rmarkdown::render(
    input = "param_report_example.Rmd",
    output_file = paste0("param_report_example_", x, ".html"),
    params = list(
      dataset = "mtcars",
      var = x
    )
  )
)
```

By now, this should not intimidate you anymore; we use `lapply()` to loop over a list of column names (that we get using `colnames()`). Because we don't want to overwrite the report we need to change the name of the output file. We do so by using `paste0()` which creates a new string that contains the variable name, so each report gets its own name. `x` inside the `paste0()` function is each element, one after the other, of the `columns` variable we defined first. Think of it as the `i` in a for loop. We then must also pass this to the `params` list, hence the `var = x`. The complete call to `rmarkdown::render()` is wrapped inside an anonymous function, because we need to use the argument `x` (which is each column defined in the `columns` list) in different places.

Before continuing, we highly recommend that you try running this yourself, and also that you try to build your own little parametrized reports. Maybe start by replacing “mtcars” by “iris” in the code to compile the reports and see what happens, and then when you’re comfortable with parametrized reports, try templating inside a parametrized report!

It is important to not fall to the temptation of copy and pasting sections of your report, or parts of your script, instead of using these more advanced features provided by the language. It is tempting, especially under time pressure, to just copy and paste bits of code and get things done instead of writing what seems to be unnecessary code to finally achieve the same thing. The problem however, is that in practice copy and pasting code to simply get things done will come bite you sooner rather than later. Especially when you’re still in the exploration/drafting phase of the project. It make take more time to set up, but once you’re done, it is much easier to experiment with different parameters, test the code or even re-use the code for other projects. Not only that, but forcing you to actually think about how to set up your code in a way that avoids repeating yourself also helps with truly understanding the problem at hand. What part of the problem is constant and does not change? What does change? How often, and why? Can you also fix these parts or not? What if instead of five sections that I need to copy and paste, I had 50 sections? How could I scale that up?

Asking yourself these questions, and solving them, will ultimately make you better programmer.

Remember: keep it DRY.



## 5 Project start

In this chapter, we are going to work together on a very simple project. This project will stay with us until the end of the book. For now, we are going to keep it simple; our goal here is to get an analysis done. We are going to download some data, analyze it, and write a little report. Then, we are going to think about how to go from this analysis to something more robust: what's important to understand, is that everything we will be doing to get our little report can be seen as a *proto-pipeline*. We are going to download data, write functions, write comments, write some tests along the way to make sure that everything is working well, and put everything into an R Markdown source file that outputs the finalized document in one command line.

The idea is to take this, and make it more a truly reproducible pipeline. So one of the first steps once we're done with the analysis is to understand why we are actually not done yet.

But for now, our main concern is to get our work done.

### 5.1 Housing in Luxembourg

We are going to download data about house prices in Luxembourg. Luxembourg is a little Western European country that looks like a shoe and is about the size of .98 Rhode Islands from the author hails from. Did you know that Luxembourg was a constitutional monarchy, and not a kingdom like Belgium, but a Grand-Duchy, and actually the last Grand-Duchy in the Word in the World? Also, what you should know to understand what we will be doing is that the country of Luxembourg is divided into Cantons, and each Cantons into Communes. Basically, if Luxembourg was the USA, Cantons would be States and Communes would be Counties (or Parishes or Boroughs). What's confusing is that "Luxembourg" is also the name of a Canton, and of a Commune, which also has the status of a city and is the capital of the country. So Luxembourg the country, is divided into Cantons, one of which is called Luxembourg as well, cantons are divided into communes, and inside the canton of Luxembourg there's the commune of Luxembourg which is also the city of Luxembourg, sometimes called Luxembourg-City, which is the capital of the country.

What you should also know is that the population is about 645.000 as of writing (January 2023), half of which are foreigners. Around 400.000 persons work in Luxembourg, of which half do not live in Luxembourg; so every morning from Monday to Friday, 200.000 people enter the country to work, and leave on the evening to go back to either Belgium, France

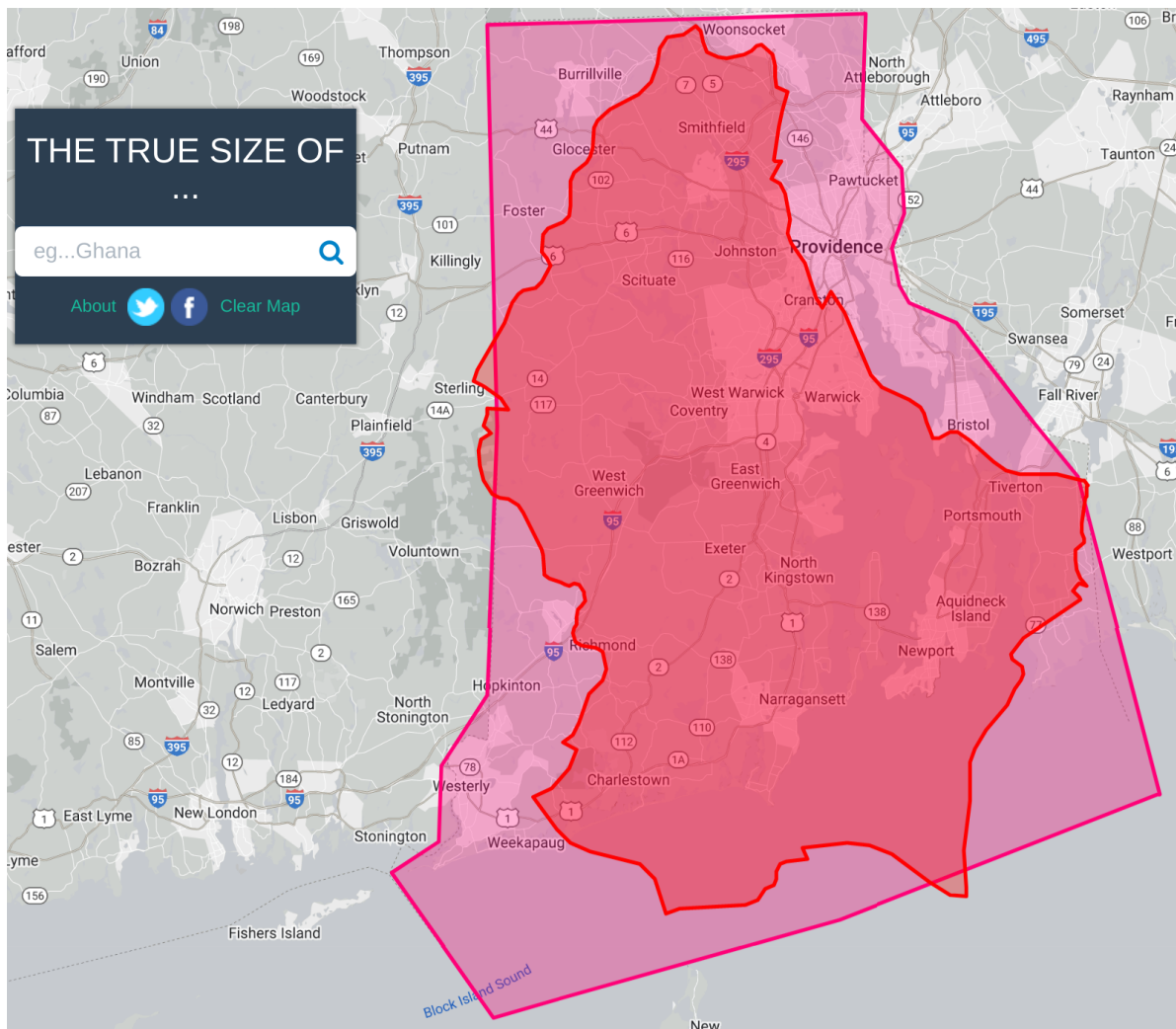
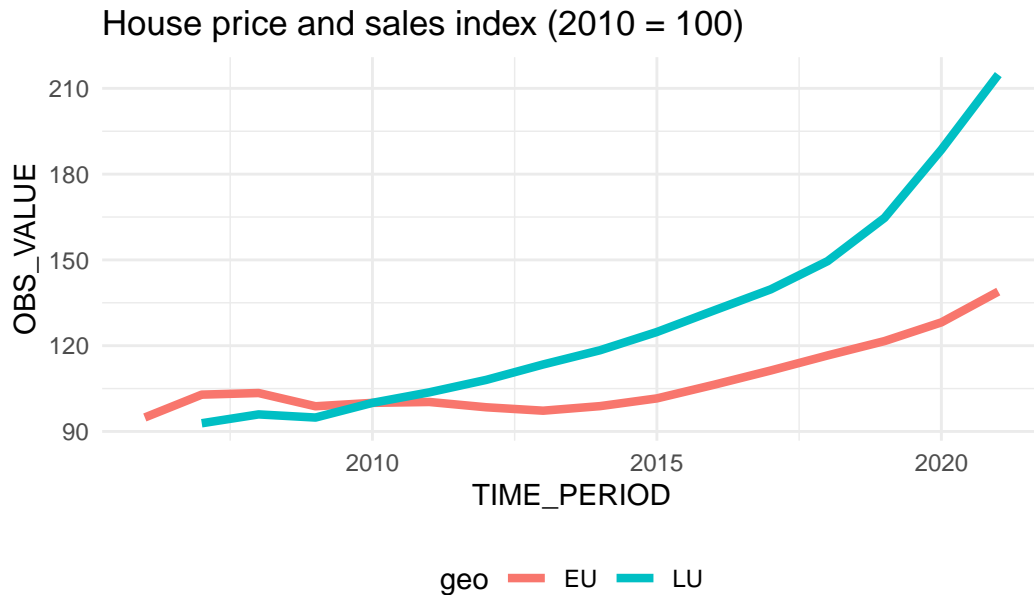


Figure 5.1: Luxembourg is about as big as the US State of Rhode Island

or Germany, the neighboring countries. As you can imagine, this puts enormous pressure on the transportation system and on the roads, but also on the housing market; everyone wants to live in Luxembourg to avoid the horrible daily commute, and everyone wants to live either in the capital city, or in the second largest urban area in the south, in a city called Esch-sur-Alzette.

The plot below shows the value of the House Price Index through time for Luxembourg and the European Union:



Source: Eurostat

If you want to download the data, click [here](#).

Let us paste the definition of the HPI in here (taken from the HPI's [metadata](#) page):

*The House Price Index (HPI) measures inflation in the residential property market. The HPI captures price changes of all types of dwellings purchased by households (flats, detached houses, terraced houses, etc.). Only transacted dwellings are considered, self-build dwellings are excluded. The land component of the dwelling is included.*

So from the plot, we can see that the price of dwellings more than doubled between 2010 and 2021; the value of the index is 214.81 in 2021 for Luxembourg, and 138.92 for the European Union as a whole.

There is a lot of heterogeneity though; the capital and the communes immediately next to the capital are much more expensive than communes from the less urbanized north, for example. The south of the country is also more expensive than the north, but not as much as the capital and surrounding communes. Not only is price driven by hand demand, but also by

scarcity; in 2021, .5% of residents owned 50% of the buildable land for housing purposes (Source: *Observatoire de l'Habitat*, Note 29, [archived download link](#)).

Our project will be quite simple; we are going to download some data, supplied as an Excel file, compiled by the Housing Observatory (*Observatoire de l'Habitat*), a service from the Ministry of Housing, which monitors the evolution of prices in the housing market, among other useful services like the identification of vacant lots for example. The advantage of their data when compared to Eurostat's data is that the data is disaggregated by commune. The disadvantage is that they only supply nominal prices, and no index. Nominal prices are the prices that you read on price tags in shops. The problem with nominal prices is that it is difficult to compare them through time. Ask yourself the following question: would you prefer to have had 500€ (or USDs) in 2003 or in 2023? You probably would have preferred them in 2003, as you could purchase a lot more with 500€ then than now. In fact, according to a random inflation calculator I googled, to match the purchasing power of \$500 in 2003, you'd need to have \$793 in 2023 (and I'd say that we find very similar values for €). But it doesn't really matter if that calculation is 100% correct: what matters is that the value of money changes, and comparisons through time are difficult, hence why an index is quite useful. So we are going to convert these nominal prices to real prices. Real prices take inflation into account and so allow us to compare prices through time. So we will need to also get some data to achieve this.

So to summarise; our goal is to:

- Get data trapped inside an Excel file into a neat data frame;
- Convert nominal to real prices using a simple method;
- Make some tables and plots and call it a day (or will we?).

And we are going to make all of this inside a R Markdown file.

## 5.2 Saving trapped data from Excel

Getting data from Excel into a tidy data frame can be very tricky. This is because very often, Excel is used as some kind of dashboarding, or presentation tool. So data is made human-readable, in contrast to machine readable. Let us quickly discuss this topic as it is essential to grasp the difference between the two (and in our experience, a lot of collective pain inflicted to statisticians and researchers could have been avoided if this concept was more well-known). The picture below shows an Excel made for human consumption:

So why is this file not machine-readable? Here are some issues:

- The table does not start in the top-left corner of the spreadsheet, which is where most importing tools expect it to be;
- The spreadsheet start with a head that contains an image and some text;
- Numbers are text and use “,” as the thousands separator;

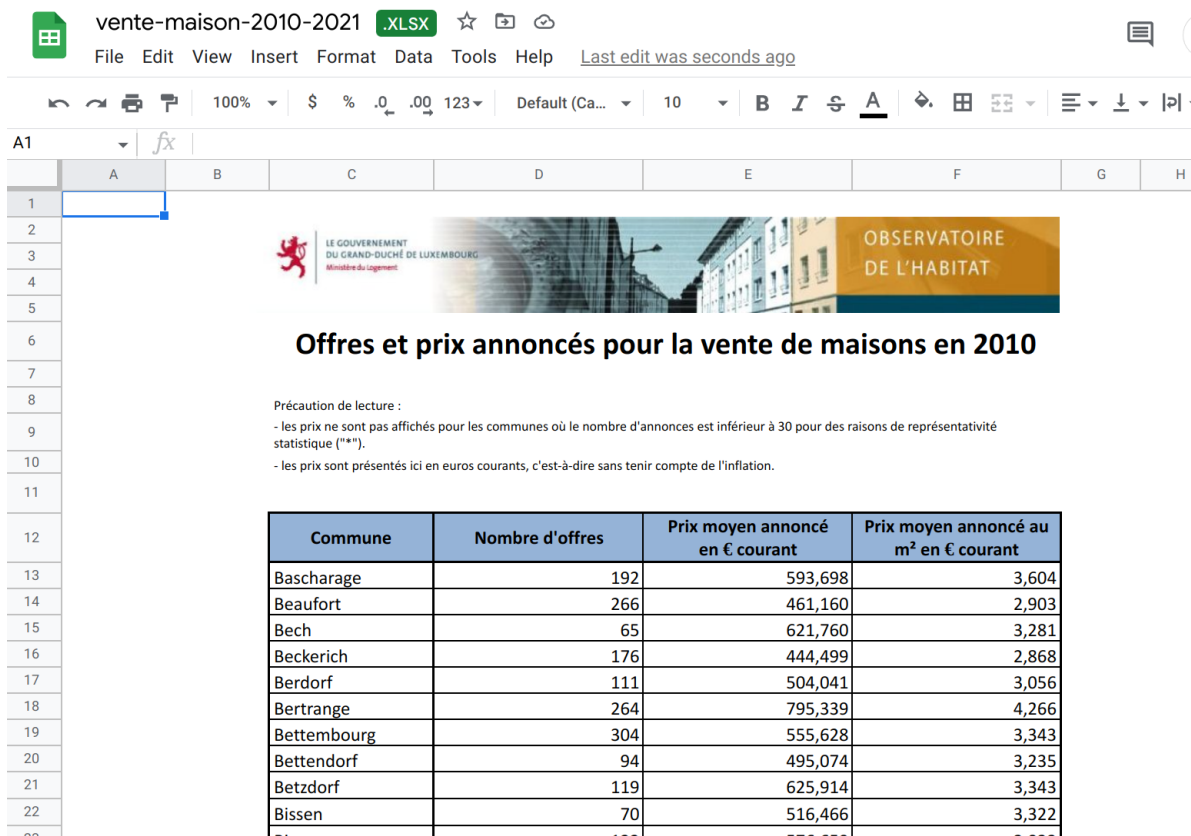


Figure 5.2: An Excel file meant for human eyes

- You don't see it in the screenshot, but each year is in a separate sheet.

That being said, this one is still very nice, and going from this Excel to a tidy dataframe will not be too difficult. In fact, we suspect that whoever made this Excel file is well aware of the contradicting requirements of human and machine readable formatting of data, and strove to find a compromise. Because more often than not, getting human readable data into a machine readable formatting is a nightmare.

This is actually the file that we are going to use for our project, so if you want to follow along, you can download it [here](#) (downloaded on january 2023 from the [luxembourgish open data portal](#)).

For ease of presentation, we are going to show you each function here separately, but we'll be putting everything together in a single Rmd file at the end. So first, let's read in the data. For this, we will be using this function here:

```
get_raw_data <- function(url = "https://github.com/b-rodrigues/rap4all/raw/master/datasets/
raw_data <- tempfile(fileext = ".xlsx")

download.file(url, raw_data)

sheets <- excel_sheets(raw_data)

read_clean <- function(..., sheet){
  read_excel(..., sheet = sheet) %>%
    mutate(year = sheet)
}

raw_data <- map_dfr(sheets,
                    ~read_clean(raw_data,
                                skip = 10,
                                sheet = .)) %>%

  clean_names()

raw_data %>%
  rename(locality = commune,
         n_offers = nombre_doffres,
         average_price_nominal_euros = prix_moyen_annonce_en_courant,
         average_price_m2_nominal_euros = prix_moyen_annonce_au_m2_en_courant,
         average_price_m2_nominal_euros = prix_moyen_annonce_au_m2_en_courant
  ) %>%
  mutate(locality = str_trim(locality)) %>%
  select(year, locality, n_offers, starts_with("average"))
```

```
}
```

Let's discuss each part of the function. This function has a single argument, the url pointing towards the data. The data is hosted on this book's Github repository, but you could just as well replace this url by the one from the Open Data Portal for example. The file gets downloaded to a temporary file (created using `tempfile(fileext = ".xlsx")`), and the name of the sheets from the file get extracted using `readxl::excel_sheets()`. This is our first dependency: for now, let's simply call `library(readxl)` at the start of our Rmd file, as anyone would do.

Then we define a new function, called `read_clean()`. This function is a wrapper around the `readxl::read_excel()` function. Remember, our source Excel Workbook contains one sheet per year, starting with 2010. Also, remember this huge header on each sheet? We can skip all these useless lines using `skip = 10`, which skips the first 10 lines of each sheet. Thankfully, this is consistent across the sheets. The sheets are named "2010", "2011", etc. So we take advantage of this to read in the required sheet, and then also add a new column with the year which is the name of the sheet. So if we would use our function on one sheet, say, "2011", we would have the data from the that sheet (because we skip the first 10 lines with the header) and a new column called `year` with "2011" everywhere.

hpi: <https://ec.europa.eu/eurostat/databrowser/bookmark/e2758aae-6a88-4684-9f6d-d0946cae3f6b?lang=fr>

deflated hpi: <https://ec.europa.eu/eurostat/databrowser/view/tipsho10/default/table?lang=en>

prix des logements en euros courants: <https://data.public.lu/fr/datasets/r/1d20f982-57e1-4ae2-a278-dc78c88c21dc>

## 5.3 Your project is done (?)

*So here the project is done, but actually it's just an Qmd file that gets compiled, so we would need to explain why this is not enough, and motivate the readers to go the full way: developing packages, using targets, and so on*

## 6 Testing your code

### 6.1 Assertive testing (and defensive programming?)

The analysis is still in Quarto, so how could the readers of this book test their code? Copying here what Miles wrote on the subject:

*‘Assertive programming’ is a topic that might be missing from the book. I think of it as a kind of dual of unit testing. Unit testing is for more generally applicable packaged code. But when you have functions in your analysis pipeline that operate on a very specific kind of input data, unit testing becomes kind of nonsensical because you’re left to dream up endless variations of your input dataset that may never occur. It’s a bit easier to flip the effort to validating the assumptions you have about your input and output data, which you can do in the pipeline functions themselves rather than separate unit testing ones. This is nice because it ensures the validation is performed in the pipeline run, and so is backed by the same reproducibility guarantees.*

I think at the end of the chapter we should hint at unit testing, but leave it as a subsection of the next chapter that deals with packaging code.



## 7 Packaging your code

In this chapter you're going to learn how to create your own package.

*We should make clear that this does not mean publishing the package on CRAN.*

### 7.1 Benefits of packages

### 7.2 Intro to package dev

*This is where fusen comes into play I guess; so we start from the Qmd file that was written before, containing the functions and the analysis, and see how we can now create a package from it, and use that file as a vignette? Copying here what Sébastien said on the matter*

### 7.3 Document your package (?)

*I guess fusen makes this process easy and leverages roxygen?*

### 7.4 Managing package dependencies (?)

*Discuss NAMESPACE and DESCRIPTION and all that. I think it's important to also discuss here how to define dependencies from remotes, not just CRAN.*

### 7.5 Unit testing

*This is where I think we should discuss unit testing*

### 7.6 pkgdown

## 8 Build automation

*Why build automation: removes cognitive load, is a form of documentation in and of itself, as Miles said*

*It is possible to communicate a great deal of domain knowledge in code, such that it is illuminating beyond the mere mechanical number crunching. To do this well the author needs to make use of certain styles and structures that produce code that has layers of domain specific abstraction a reader can traverse up and down as they build their understanding of the project. Functional programming style, coupled with a dependency graph as per {targets} are useful tools in this regard.*

## 9 Introduction to reproducibility

*Since we said in the intro to the book that reproducibility is on a continuum, I think that this chapter should focus on the bare minimum, which would culminate with renv*

*Then at the end, explain why renv is not enough (does nothing for R itself, nor the environment the code is running on)*

## 10 Advanced topics in reproducibility

*Now that the readers are familiar with renv, but also its shortcomings, we can go a step further and introduce Docker. I think some primer on the Linux command line could be included here*

### 10.1 First steps with Docker

*To write your own Dockerfile, you need some familiarity with the Linux cli, so here's...*

### 10.2 A primer on the Linux command line

### 10.3 Dockrizing your project

## **11 Continuous integration and continuous deployment/delivery**

## References

- Arel-Bundock, Vincent. 2022. “modelssummary: Data and Model Summaries in R.” *Journal of Statistical Software* 103 (1): 1–23. <https://doi.org/10.18637/jss.v103.i01>.
- Chambers, John M. 2014. “Object-Oriented Programming, Functional Programming and R.” *Statistical Science* 29 (2): 167–80. <https://doi.org/10.1214/13-STS452>.
- Gohel, David, and Panagiotis Skintzos. 2023. *Flextable: Functions for Tabular Reporting*.
- Leisch, Friedrich. 2002. “Sweave: Dynamic Generation of Statistical Reports Using Literate Data Analysis.” In *Compstat*, edited by Wolfgang Härdle and Bernd Rönz, 575–80. Physica-Verlag HD.
- Wickham, Hadley. 2019. *Advanced r*. CRC press.
- Xie, Yihui. 2014. “Knitr: A Comprehensive Tool for Reproducible Research in R.” In *Implementing Reproducible Computational Research*, edited by Victoria Stodden, Friedrich Leisch, and Roger D. Peng. Chapman; Hall/CRC. <http://www.crcpress.com/product/isbn/9781466561595>.