

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

Jane Doe

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
<b>3 Version control</b>	<b>29</b>
<b>4 Getting started with your project</b>	<b>30</b>
4.1 Literate programming . . . . .	30
4.1.1 Why bother? . . . . .	30
4.2 Quarto basics . . . . .	30
4.3 Parametrized reports . . . . .	30
4.4 Your project is done (?) . . . . .	30
<b>5 Testing your code</b>	<b>31</b>
5.1 Assertive testing (and defensive programming?) . . . . .	31

<b>6</b>	<b>Packaging your code</b>	<b>32</b>
6.1	Benefits of packages . . . . .	32
6.2	Intro to packge dev . . . . .	32
6.3	Document your package (?) . . . . .	32
6.4	Managing package dependencies (?) . . . . .	32
6.5	Unit testing . . . . .	32
6.6	pkgdown . . . . .	32
<b>7</b>	<b>Build automation</b>	<b>33</b>
<b>8</b>	<b>Introduction to reproducibility</b>	<b>34</b>
<b>9</b>	<b>Advanced topics in reproducibility</b>	<b>35</b>
9.1	First steps with Docker . . . . .	35
9.2	A primer on the Linux command line . . . . .	35
9.3	Dockrizing your project . . . . .	35
<b>10</b>	<b>Continuous integration and continuous deployment/delivery</b>	<b>36</b>
	<b>References</b>	<b>37</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 2023-01-15 10
```

```
[3] "Total running time: 0.000967979431152344 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)
```

that does all the steps above in one go. The advantage of splitting `big_function()` into `f()`, `g()` and `h()` is that you can reuse these smaller functions in other projects much more easily. Make them work together by sharing a common interface. The list is usually a good candidate for this.

---

<sup>1</sup><http://www.catb.org/~esr/writings/taoup/html/ch01s06.html>

## 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 you're 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:

```
sqrt_newton <- 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 <- sqrt_newton(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

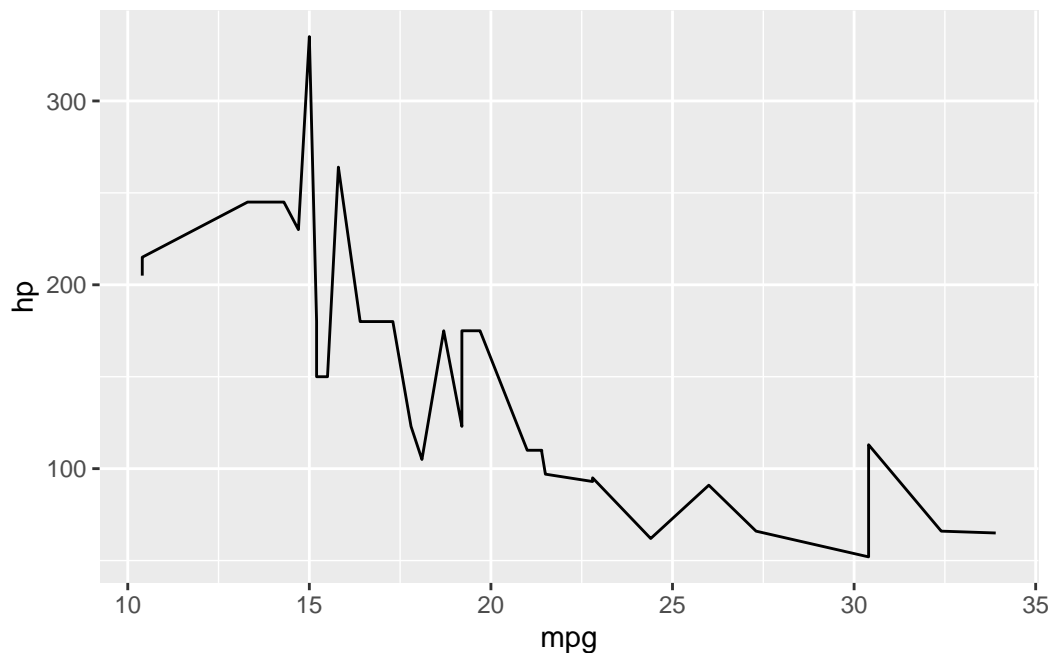
`$b`

`$b$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.

## 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 Getting started with your project

We suggest to not start with an R script, but start from a Qmd file.

*What Sébastien said on the matter:*

\*There seems to be two way of presenting such a workflow:

What does the complete project looks like at the end and what tools do I need? How do I start working on it?\*

*This questions should get answered in the intro?*

Refer to <https://github.com/b-rodrigues/rap4all/issues/1#issuecomment-1327627267>

### 4.1 Literate programming

#### 4.1.1 Why bother?

*Allows you to explain what you're doing as you're coding. This file can later be inflated, if necessary, to make a package using {fusen}.*

### 4.2 Quarto basics

*Teach some Quarto basics*

### 4.3 Parametrized reports

### 4.4 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*

## 5 Testing your code

### 5.1 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.

## 6 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.*

### 6.1 Benefits of packages

### 6.2 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*

### 6.3 Document your package (?)

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

### 6.4 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.*

### 6.5 Unit testing

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

### 6.6 pkgdown



## 7 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.*

## 8 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)*

## 9 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*

### 9.1 First steps with Docker

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

### 9.2 A primer on the Linux command line

### 9.3 Dockrizing your project

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

## References