6 R programming

The tools in Chapters 2-5 will allow you to manipulate, summarise, and visualise your data in all sorts of ways. But what if you need to compute some statistic that there isn't a function for? What if you need automatic checks of your data and results? What if you need to repeat the same analysis for a large number of files? This is where the programming tools you'll learn about in this chapter, like loops and conditional statements, come in handy. And this is where you take the step from being able to use R for routine analyses to being able to use R for *any* analysis.

After working with the material in this chapter, you will be able to use R to:

- Write your own R functions,
- Use several new pipe operators,
- Use conditional statements to perform different operations depending on whether or not a condition is satisfied,
- Iterate code operations multiple times using loops,
- Iterate code operations multiple times using functionals, and
- Measure the performance of your R code.

6.1 Functions

Suppose that we wish to compute the mean of a vector `x`. One way to do this would be to use `sum` and `length`:

```
x <- 1:100
# Compute mean:
sum(x)/length(x)
```

Now suppose that we wish to compute the mean of several vectors. We could do this by repeated use of `sum` and `length`:

```
x <- 1:100
y <- 1:200
z <- 1:300

# Compute means:
sum(x)/length(x)
sum(y)/length(y)
sum(z)/length(x)
```

But wait! I made a mistake when I copied the code to compute the mean of `z` – I forgot to change `length(x)` to `length(z)`! This is an easy mistake to make when you repeatedly copy and paste code. In addition, repeating the same code multiple times just doesn't look good. It would be much more convenient to have a single function for computing the means. Fortunately, such a function exists – `mean`:

```
# Compute means
mean(x)
mean(y)
mean(z)
```

As you can see, using `mean` makes the code shorter and easier to read and reduces the risk of errors induced by copying and pasting code (we only have to change the argument of one function instead of two).

You've already used a ton of different functions in R: functions for computing means, manipulating data, plotting graphics, and more. All these functions have been written by somebody who thought that they needed to repeat a task (e.g., computing a mean or plotting a bar chart) over and over again. And in such cases, it is much more convenient to have a function that does that task than to have to write or copy code every time you want to do it. This is true also for your own work – whenever you need to repeat the same task several times, it is probably a good idea to write a function for it. It will reduce the amount of code you have to write and lessen the risk of errors caused by copying and pasting old code. In this section, you will learn how to write your own functions.

6.1.1 Creating functions

For the sake of the example, let's say that we wish to compute the mean of several vectors but that the function `mean` doesn't exist. We would therefore like to write our own function for computing the mean of a vector. An R function takes some variables as input (arguments or parameters) and returns an object. Functions are defined using `function`. The definition follows a particular format:

```
function_name <- function(argument1, argument2, ...)
{
      # ...
      # Some rows with code that creates some_object
      # ...
      return(some_object)
}
```

In the case of our function for computing a mean, this could look like:

```
average <- function(x)
{
      avg <- sum(x)/length(x)
      return(avg)
}
```

This defines a function called `average` that takes an object called `x` as input. It computes the sum of the elements of `x`, divides that by the number of elements in `x`, and returns the resulting mean.

If we now make a call to `average(x)`, our function will compute the mean value of the vector `x`. Let's try it out, to see that it works:

```
x <- 1:100
y <- 1:200
average(x)
average(y)
```

6.1.2 Local and global variables

Note that despite the fact that the vector was called `x` in the code we used to define the function, `average` works regardless of whether the input is called `x` or `y`. This is because R distinguishes between *global variables* and *local variables*. A global variable is created in the *global environment* outside a function. It is available to all functions (these are the variables that you can see in the Environment panel in RStudio). A local variable is created in the *local environment* inside a function. It is only available to that particular function. For instance, our `average` function creates a variable called `avg`, yet when we attempt to access `avg` after running `average` this variable doesn't seem to exist:

```
average(x)
avg
```

Because `avg` is a local variable, it is only available inside of the `average` function. Local variables take precedence over global variables inside the functions to which they belong. Because we named the argument used in the function `x`, `x` becomes the name of a local

variable in `average`. As far as `average` is concerned, there is only one variable named `x`, and that is whatever object that was given as input to the function, regardless of what its original name was. Any operations performed on the local variable `x` won't affect the global variable `x` at all.

Functions can access global variables:

```
y_squared <- function()
{
      return(y^2)
}

y <- 2
y_squared()
```

But operations performed on global variables inside functions won't affect the global variable:

```
add_to_y <- function(n)
{
      y <- y + n
}

y <- 1
add_to_y(1)
y
```

Suppose you really need to change a global variable inside a function[41]. In that case, you can use an alternative assignment operator, `<<-`, which assigns a value to the variable in the *parent environment* to the current environment. If you use `<<-` for assignment inside a function that is called from the global environment, this means that the assignment takes place in the global environment. But if you use `<<-` in a function (function 1) that is called by another function (function 2), the assignment will take place in the environment for function 2, thus affecting a local variable in function 2. Here is an example of a global assignment using `<<-`:

```
add_to_y_global <- function(n)
{
      y <<- y + n
}

y <- 1
add_to_y_global(1)
y
```

6.1.3 Will your function work?

It is always a good idea to test if your function works as intended, and to try to figure out what can cause it to break. Let's return to our `average` function:

```
average <- function(x)
{
      avg <- sum(x)/length(x)
      return(avg)
}
```

We've already seen that it seems to work when the input `x` is a numeric vector. But what happens if we input something else instead?

```
average(c(1, 5, 8)) # Numeric input
average(c(TRUE, TRUE, FALSE)) # Logical input
average(c("Lady Gaga", "Tool", "Dry the River")) # Character input
average(data.frame(x = c(1, 1, 1), y = c(2, 2, 1))) # Numeric df
average(data.frame(x = c(1, 5, 8), y = c("A", "B", "C"))) # Mixed type
```

The first two of these render the desired output (the `logical` values being represented by 0s and 1s), but the rest don't. Many R functions include checks that the input is of the correct type, or checks to see which method should be applied depending on what data type the input is. We'll learn how to perform such checks in Section 6.3.

As a side note, it is possible to write functions that don't end with `return`. In that case, the output (i.e., what would be written in the Console if you'd run the code there) from the last line of the function will automatically be returned. I prefer to (almost) always use `return` though,

as it is easy to accidentally make the function return nothing by finishing it with a line that yields no output. Below are two examples of how we could have written `average` without a call to `return`. The first doesn't work as intended, because the function's final (and only) line doesn't give any output.

```
average_bad <- function(x)
{
      avg <- sum(x)/length(x)
}

average_ok <- function(x)
{
      sum(x)/length(x)
}

average_bad(c(1, 5, 8))
average_ok(c(1, 5, 8))
```

6.1.4 More on arguments

It is possible to create functions with as many arguments as you like, but it will become quite unwieldy if the user has to supply too many arguments to your function. It is therefore common to provide default values to arguments, which is done by setting a value in the `function` call. Here is an example of a function that computes x^n, using $n = 2$ as the default:

```
power_n <- function(x, n = 2)
{
      return(x^n)
}
```

If we don't supply `n`, `power_n` uses the default `n = 2`:

```
power_n(3)
```

But if we supply an `n`, `power_n` will use that instead:

```
power_n(3, 1)
power_n(3, 3)
```

For clarity, you can specify which value corresponds to which argument:

```
power_n(x = 2, n = 5)
```

…and can then even put the arguments in the wrong order:

```
power_n(n = 5, x = 2)
```

However, if we only supply `n` we get an error, because there is no default value for `x`:

```
power_n(n = 5)
```

It is possible to pass a function as an argument. Here is a function that takes a vector and a function as input and applies the function to the first two elements of the vector:

```
apply_to_first2 <- function(x, func)
{
      result <- func(x[1:2])
      return(result)
}
```

By supplying different functions to `apply_to_first2`, we can make it perform different tasks:

```
x <- c(4, 5, 6)
apply_to_first2(x, sqrt)
apply_to_first2(x, is.character)
apply_to_first2(x, power_n)
```

But what if the function that we supply requires additional arguments? Using `apply_to_first2` with `sum` and the vector `c(4, 5, 6)` works fine:

```
apply_to_first2(x, sum)
```

But if we instead use the vector `c(4, NA, 6)`, the function returns `NA`:

```
x <- c(4, NA, 6)
apply_to_first2(x, sum)
```

Perhaps we'd like to pass `na.rm = TRUE` to `sum` to ensure that we get a `numeric` result, if at all possible. This can be done by adding `...` to the list of arguments for both functions, which indicates additional parameters (to be supplied by the user) that will be passed to `func`:

```
apply_to_first2 <- function(x, func, ...)
{
      result <- func(x[1:2], ...)
      return(result)
}

x <- c(4, NA, 6)
apply_to_first2(x, sum)
apply_to_first2(x, sum, na.rm = TRUE)
```

~

Exercise 6.1 Write a function that converts temperature measurements in degrees Fahrenheit to degrees Celsius, and apply it to the `Temp` column of the `airquality` data.

(Click here to go to the solution.)

Exercise 6.2 Practice writing functions by doing the following:

1. Write a function that takes a vector as input and returns a vector containing its minimum and maximum, without using `min` and `max`.
2. Write a function that computes the mean of the squared values of a vector using `mean`, and that takes additional arguments that it passes on to `mean` (e.g., `na.rm`).

(Click here to go to the solution.)

6.1.5 Namespaces

It is possible, and even likely, that you will encounter functions in packages with the same name as functions in other packages. Or, similarly, that there are functions in packages with the same names as those you have written yourself. This is of course a bit of a headache, but it's actually something that can be overcome without changing the names of the functions. Just like variables can live in different environments, R functions live in *namespaces*, usually corresponding to either the global environment or the package they belong to. By specifying which namespace to look for the function in, you can use multiple functions that all have the same name.

For example, let's create a function called `sqrt`. There is already such a function in the `base` package[42] (see `?sqrt`), but let's do it anyway:

```
sqrt <- function(x)
{
      return(x^10)
}
```

If we now apply `sqrt` to an object, the function that we just defined will be used:

```
sqrt(4)
```

But if we want to use the `sqrt` from `base`, we can specify that by writing the namespace (which almost always is the package name) followed by `::` and the function name:

```
base::sqrt(4)
```

The `::` notation can also be used to call a function or object from a package without loading the package's namespace:

```
msleep # Doesn't work if ggplot2 isn't loaded
ggplot2::msleep # Works, without loading the ggplot2 namespace!
```

When you call a function, R will look for it in all active namespaces, following a particular order. To see the order of the namespaces, you can use `search`:

```
search()
```

Note that the global environment is first in this list – meaning that the functions that you define always will be preferred to functions in packages.

All this being said, note that it is bad practice to give your functions and variables the same names as common functions. Don't name them `mean`, `c`, or `sqrt`. Nothing good can ever come from that sort of behaviour.

Nothing.

6.1.6 Sourcing other scripts

If you want to reuse a function that you have written in a new script, you can of course copy it into that script. But if you then make changes to your function, you will quickly end up with several different versions of it. A better idea can therefore be to put the function in a separate script, which you then can call in each script where you need the function. This is done using `source`. If, for instance, you have code that defines some functions in a file called `helper-functions.R` in your working directory, you can run it (thus defining the functions) when the rest of your code is run by adding `source("helper-functions.R")` to your code.

Another option is to create an R package containing the function, but that is beyond the scope of this book. Should you choose to go down that route, I highly recommend reading *R Packages* by Wickham and Bryan.

6.2 More on pipes

We have seen how the pipe operator `|>` can be used to chain functions together. But there are also other pipe operators that are useful. In this section we'll look at some of them, and see how you can create functions using pipes.

6.2.1 *Ce ne sont pas non plus des pipes*

Although `|>` is the pipe operator that we use the most, there are situations where other pipe operators are more appropriate. The `magrittr` package provides a number of other pipes that are useful in certain situations.

One example is when you want to pass variables rather than an entire dataset to the next function. This is needed for instance if you want to use `cor` to compute the correlation between two variables, because `cor` takes two vectors as input instead of a data frame. Previously, we solved this by using `with`:

```
library(dplyr)
airquality |>
      filter(Temp > 80) |>
      with(cor(Temp, Wind))
```

Another option is to use the `%$%` pipe, which passes on the names of all variables in your data frame instead of the actual data frame:

```
library(magrittr)
airquality |>
      filter(Temp > 80) %$%
      cor(Temp, Wind)
```

If you want to modify a variable using a pipe, you can use the *compound assignment* pipe `%<>%`. The following three lines all yield exactly the same result:

```
x <- 1:8;   x <- sqrt(x);       x
x <- 1:8;   x |> sqrt() -> x;   x
x <- 1:8;   x %<>% sqrt;        x
```

As long as the first pipe in the pipeline is the compound assignment operator `%<>%`, you can combine it with other pipes:

```
x <- 1:8
x %<>% subset(x > 5) |> sqrt()
x
```

Sometimes, you want to do something in the middle of a pipeline, like creating a plot, before continuing to the next step in the chain. The *tee* operator `%T>%` can be used to execute a function without passing on its output (if any). Instead, it passes on the output to its left. Here is an example:

```
airquality |>
      filter(Temp > 80) %T>%
      plot() %$%
      cor(Temp, Wind)
```

Note that if we'd used an ordinary pipe `|>` instead, we'd get an error:

```
airquality |>
      filter(Temp > 80) |>
      plot() %$%
      cor(Temp, Wind)
```

The reason is that `cor` looks for the variables `Temp` and `Wind` in the plot object, and not in the data frame. The tee operator takes care of this by passing on the data from its left side.

When using the tee operator with `ggplot`, you need to wrap `ggplot` in curly brackets and in a call to `print`:

```
library(ggplot2)
airquality |>
      filter(Temp > 80) %T>%
      {print(ggplot(., aes(Temp, Wind)) + geom_point())} %$%
      cor(Temp, Wind)
```

6.2.2 Writing functions with pipes

If you will be reusing the same pipeline multiple times, you may want to create a function for it. Let's say that you have a data frame containing only `numeric` variables, and that you want to create a scatterplot matrix (which can be done using `plot`) and compute the correlations between all variables (using `cor`). As an example, you could do this for `airquality` as follows:

```
airquality %T>% plot |>  cor()
```

To define a function for this combination of operators, we write:

```
plot_and_cor <- function(x) { x %T>% plot |>  cor() }
```

If you only use `magrittr` pipes in your function, you can write this in a shorter form, without the `function(...)` part:

```
plot_and_cor <- . %T>% plot %>% cor
```

We can now use this function just like any other:

```
# With the airquality data:
airquality |> plot_and_cor
plot_and_cor(airquality)

# With the bookstore data:
age <- c(28, 48, 47, 71, 22, 80, 48, 30, 31)
purchase <- c(20, 59, 2, 12, 22, 160, 34, 34, 29)
bookstore <- data.frame(age, purchase)
bookstore |> plot_and_cor
```

∼

Exercise 6.3 Write a function that takes a data frame as input and uses pipes to print the number of `NA` values in the data, remove all rows with `NA` values and return a summary of the remaining data.

(Click here to go to the solution.)

Exercise 6.4 Pipes are operators, that is, functions that take two variables as input and can be written without parentheses (other examples of operators are `+` and `*`). You can define your own operators just as you would any other function. For instance, we can define an operator called `quadratic` that takes two numbers `a` and `b` as input and computes the quadratic expression $(a + b)^2$:

```
`%quadratic%` <- function(a, b) { (a + b)^2 }
2 %quadratic% 3
```

Create an operator called `%against%` that takes two vectors as input and draws a scatterplot of them.

(Click here to go to the solution.)

6.3 Checking conditions

Sometimes you'd like your code to perform different operations depending on whether or not a certain condition is fulfilled. Perhaps you want it to do something different if there is missing data, if the input is a `character` vector, or if the largest value in a `numeric` vector is greater than some number. In Section 2.11.3 you learned how to filter data using conditions. In this section, you'll learn how to use conditional statements for a number of other tasks.

6.3.1 `if` and `else`

The most important functions for checking whether a condition is fulfilled are `if` and `else`. The basic syntax is

```
if(condition) { do something } else { do something else }
```

The condition should return a single `logical` value, so that it evaluates to either `TRUE` or `FALSE`. If the condition is fulfilled, i.e., if it is `TRUE`, the code inside the first pair of curly brackets will run, and if it's not (`FALSE`), the code within the second pair of curly brackets will run instead.

As a first example, assume that you want to compute the reciprocal of x, $1/x$, unless $x = 0$, in which case you wish to print an error message:

```
x <- 2
if(x == 0) { cat("Error! Division by zero.") } else { 1/x }
```

Now try running the same code with `x` set to `0`:

```
x <- 0
if(x == 0) { cat("Error! Division by zero.") } else { 1/x }
```

Alternatively, we could check if $x \neq 0$ and then change the order of the segments within the curly brackets:

```
x <- 0
if(x != 0) { 1/x } else { cat("Error! Division by zero.") }
```

You don't have to write all of the code on the same line, but you must make sure that the `else` part is on the same line as the first `}`:

```
if(x == 0)
{
    cat("Error! Division by zero.")
} else
{
    1/x
}
```

You can also choose not to have an `else` part at all. In that case, the code inside the curly brackets will run if the condition is satisfied, and if not, nothing will happen:

```
x <- 0
if(x == 0) { cat("x is 0.") }

x <- 2
if(x == 0) { cat("x is 0.") }
```

Finally, if you need to check a number of conditions one after another, in order to list different possibilities, you can do so by repeated use of `if` and `else`:

```
if(x == 0)
{
    cat("Error! Division by zero.")
} else if(is.infinite((x)))
{
    cat("Error! Division by infinity.")
} else if(is.na((x)))
{
    cat("Error! Division by NA.")
} else
{
    1/x
}
```

6.3.2 `&` & `&&`

Just as when we used conditions for filtering in Sections 2.11.3 and 5.8.2, it is possible to combine several conditions into one using `&` (AND) and `|` (OR). However, the `&` and `|` operators are vectorised, meaning that they will return a vector of `logical` values whenever possible. This is not desirable in conditional statements, where the condition must evaluate to a single value. Using a condition that returns a vector results in an error message:

```
if(c(1, 2) == 2) { cat("The vector contains the number 2.\n") }
if(c(2, 1) == 2) { cat("The vector contains the number 2.\n") }
```

Usually, if a condition evaluates to a vector, it is because you've made an error in your code. Remember, if you really need to evaluate a condition regarding the elements in a vector, you can collapse the resulting `logical` vector to a single value using `any` or `all`.

Some texts recommend using the operators `&&` and `||` instead of `&` and `|` in conditional statements. These work almost like `&` and `|`, but check the conditions on the left-hand and right-hand sides sequentially. `&` and `|` always evaluate all the conditions that you're combining, while `&&` and `||` don't: `&&` stops as soon as it encounters a `FALSE`, and `||` stops as soon as it encounters a `TRUE`. Consequently, you can put the conditions you wish to combine in a particular order to make sure that they can be evaluated. For instance, you may

want first to check that a variable exists, and then check a property. This can be done using `exists` to check whether or not it exists – note that the variable name must be written within quotes:

```
# "a" is a variable that doesn't exist

# Using && works:
if(exists("a") && a > 0)
{
    cat("The variable exists and is positive.")
} else { cat("a doesn't exist or is negative.") }

# But using & doesn't, because it attempts to evaluate a>0
# even though a doesn't exist:
if(exists("a") & a > 0)
{
    cat("The variable exists and is positive.")
} else { cat("a doesn't exist or is negative.") }
```

6.3.3 `ifelse`

It is common that you want to assign different values to a variable depending on whether or not a condition is satisfied:

```
x <- 2

if(x == 0)
{
    reciprocal <- "Error! Division by zero."
} else
{
    reciprocal <- 1/x
}

reciprocal
```

In fact, this situation is so common that there is a special function for it: `ifelse`:

```
reciprocal <- ifelse(x == 0, "Error! Division by zero.", 1/x)
```

`ifelse` evaluates a condition and then returns different answers depending on whether the condition is `TRUE` or `FALSE`. It can also be applied to vectors, in which case it checks the condition for each element of the vector and returns an answer for each element:

```
x <- c(-1, 1, 2, -2, 3)
ifelse(x > 0, "Positive", "Negative")
```

6.3.4 `switch`

For the sake of readability, it is usually a good idea to try to avoid chains of the type `if() {} else if() {} else if() {} else {}`. One function that can be useful for this is `switch`, which lets you list a number of possible results, either by position (a number) or by name:

```
position <- 2
switch(position,
       "First position",
       "Second position",
       "Third position")

name <- "First"
switch(name,
       First = "First name",
       Second = "Second name",
       Third = "Third name")
```

You can for instance use this to decide what function should be applied to your data:

```
x <- 1:3
y <- c(3, 5, 4)
method <- "nonparametric2"
cor_xy <- switch(method,
       parametric = cor(x, y, method = "pearson"),
       nonparametric1 = cor(x, y, method = "spearman"),
       nonparametric2 = cor(x, y, method = "kendall"))
cor_xy
```

6.3.5 Failing gracefully

Conditional statements are useful for ensuring that the input to a function you've written is of the correct type. In Section 6.1.3 we saw that our `average` function failed if we applied it to a `character` vector:

```
average <- function(x)
{
       avg <- sum(x)/length(x)
       return(avg)
}

average(c("Lady Gaga", "Tool", "Dry the River"))
```

By using a conditional statement, we can provide a more informative error message. We can check that the input is `numeric` and, if it's not, stop the function and print an error message, using `stop`:

```
average <- function(x)
{
      if(is.numeric(x))
      {
            avg <- sum(x)/length(x)
            return(avg)
      } else
      {
            stop("The input must be a numeric vector.")
      }
}

average(c(1, 5, 8))
average(c("Lady Gaga", "Tool", "Dry the River"))
```

~

Exercise 6.5 Which of the following conditions are `TRUE`? First think about the answer, and then check it using R.

```
x <- 2
y <- 3
z <- -3
```

1. `x > 2`
2. `x > y | x > z`
3. `x > y & x > z`
4. `abs(x*z) >= y`

(Click here to go to the solution.)

Exercise 6.6 Fix the errors in the following code:

```
x <- c(1, 2, pi, 8)

# Only compute square roots if x exists
# and contains positive values:
if(exists(x)) { if(x > 0) { sqrt(x) } }
```

(Click here to go to the solution.)

6.4 Iteration using loops

We have already seen how you can use functions to make it easier to repeat the same task over and over. But there is still a part of the puzzle missing – what if, for example, you wish to apply a function to each column of a data frame? What if you want to apply it to data from a number of files, one at a time? The solution to these problems is to use *iteration*. In this section, we'll explore how to perform iteration using *loops*.

6.4.1 `for` loops

`for` loops can be used to run the same code several times, with different settings, e.g., different data, in each iteration. Their use is perhaps best explained by some examples. We create the loop using `for`, give the name of a *control variable* and a vector containing its values (the control variable controls how many iterations to run), and then write the code that should be repeated in each iteration of the loop. In each iteration, a new value of the control variable is used in the code, and the loop stops when all values have been used.

As a first example, let's write a `for` loop that runs a block of code five times, where the block prints the current iteration number:

```
for(i in 1:5)
{
    cat("Iteration", i, "\n")
}
```

This is equivalent to writing:

```
cat("Iteration", 1, "\n")
cat("Iteration", 2, "\n")
cat("Iteration", 3, "\n")
cat("Iteration", 4, "\n")
cat("Iteration", 5, "\n")
```

The upside is that we didn't have to copy and edit the same code multiple times – and as you can imagine, this benefit becomes even more pronounced if you have more complicated code blocks.

The values for the control variable are given in a vector, and the code block will be run once for each element in the vector – we say that we *loop over the values in the vector*. The vector doesn't have to be `numeric` – here is an example with a `character` vector:

```
for(word in c("one", "two", "five hundred and fifty five"))
{
    cat("Iteration", word, "\n")
}
```

Of course, loops are used for so much more than merely printing text on the screen. A common use is to perform some computation and then store the result in a vector. In this case, we must first create an empty vector to store the result in, e.g., using `vector`, which creates an empty vector of a specific type and length:

```
squares <- vector("numeric", 5)

for(i in 1:5)
{
    squares[i] <- i^2
}
squares
```

In this case, it would have been both simpler and computationally faster to compute the squared values by running `(1:5)^2`. This is known as a *vectorised* solution and is very important in R. We'll discuss vectorised solutions in detail in Section 6.5.

When creating the values used for the control variable, we often wish to create different sequences of numbers. Two functions that are very useful for this are `seq`, which creates sequences, and `rep`, which repeats patterns:

```
seq(0, 100)
seq(0, 100, by = 10)
seq(0, 100, length.out = 21)

rep(1, 4)
rep(c(1, 2), 4)
rep(c(1, 2), c(4, 2))
```

Finally, `seq_along` can be used to create a sequence of indices for a vector of a data frame, which is useful if you wish to iterate some code for each element of a vector or each column of a data frame:

```
seq_along(airquality) # Gives the indices of all columns of the data
                      # frame
seq_along(airquality$Temp) # Gives the indices of all elements of the
                           # vector
```

Here is an example of how to use `seq_along` to compute the mean of each column of a data frame:

```
# Compute the mean for each column of the airquality data:
means <- vector("double", ncol(airquality))

# Loop over the variables in airquality:
for(i in seq_along(airquality))
{
      means[i] <- mean(airquality[[i]], na.rm = TRUE)
}

# Check that the results agree with those from the colMeans function:
means
colMeans(airquality, na.rm = TRUE)
```

The line inside the loop could have read `means[i] <- mean(airquality[,i], na.rm = TRUE)`, but that would have caused problems if we'd used it with a `data.table` or `tibble` object; see Section 5.9.4.

Finally, we can also change the values of the data in each iteration of the loop. Some machine learning methods require that the data is *standardised*, i.e., that all columns have mean 0 and standard deviation 1. This is achieved by subtracting the mean from each variable and then dividing each variable by its standard deviation. We can write a function for this that uses a loop, changing the values of a column in each iteration:

```
standardise <- function(df, ...)
{
      for(i in seq_along(df))
      {
          df[[i]] <- (df[[i]] - mean(df[[i]], ...))/sd(df[[i]], ...)
      }
      return(df)
}

# Try it out:
aqs <- standardise(airquality, na.rm = TRUE)
colMeans(aqs, na.rm = TRUE) # Non-zero due to floating point
                            # arithmetics!
sd(aqs$Wind)
```

~

Exercise 6.7 Practice writing `for` loops by doing the following:

1. Compute the mean temperature for each month in the `airquality` dataset using a loop rather than an existing function.
2. Use a `for` loop to compute the maximum and minimum value of each column of the `airquality` data frame, storing the results in a data frame.
3. Make your solution to the previous task reusable by writing a function that returns the maximum and minimum value of each column of a data frame.

(Click here to go to the solution.)

Exercise 6.8 Use `rep` or `seq` to create the following vectors:

1. `0.25 0.5 0.75 1`
2. `1 1 1 2 2 5`

(Click here to go to the solution.)

Exercise 6.9 As an alternative to `seq_along(airquality)` and `seq_along(airquality$Temp)`, we could create the same sequences using `1:ncol(airquality)` and `1:length(airquality$Temp)`. Use `x <- c()` to create a vector of length zero. Then create loops that use `seq_along(x)` and `1:length(x)` as values for the control variable. How many iterations are the two loops run? Which solution is preferable?

(Click here to go to the solution.)

Exercise 6.10 An alternative to standardisation is *normalisation*, where all `numeric` variables are rescaled so that their smallest value is 0 and their largest value is 1. Write a function that normalises the variables in a data frame containing `numeric` columns.

(Click here to go to the solution.)

Exercise 6.11 The function `list.files` can be used to create a vector containing the names of all files in a folder. The `pattern` argument can be used to supply a regular expression describing a file name pattern. For instance, if `pattern = "\\.csv$"` is used, only `.csv` files will be listed.

Create a loop that goes through all `.csv` files in a folder and prints the names of the variables for each file.

(Click here to go to the solution.)

6.4.2 Loops within loops

In some situations, you'll want to put a loop inside another loop. Such loops are said to be *nested*. An example is if we want to compute the correlation between all pairs of variables in `airquality` and store the result in a matrix:

```
cor_mat <- matrix(NA, nrow = ncol(airquality),
                  ncol = ncol(airquality))
for(i in seq_along(airquality))
{
    for(j in seq_along(airquality))
    {
        cor_mat[i, j] <- cor(airquality[[i]], airquality[[j]],
                             use = "pairwise.complete")
    }
}

# Element [i, j] of the matrix now contains the correlation between
# variables i and j:
cor_mat
```

Once again, there is a vectorised solution to this problem, given by `cor(airquality, use = "pairwise.complete")`. As we will see in Section 6.6, vectorised solutions like this can be several times faster than solutions that use nested loops. In general, solutions involving nested loops tend to be fairly slow; but, on the other hand, they are often easy and straightforward to implement.

6.4.3 Keeping track of what's happening

Sometimes each iteration of your loop takes a long time to run, and you'll want to monitor its progress. This can be done using printed messages or a progress bar in the Console panel, or sound notifications. We'll showcase each of these using a loop containing a call to `Sys.sleep`, which pauses the execution of R commands for a short time (determined by the user).

First, we can use `cat` to print a message describing the progress. Adding `\r` to the end of a string allows us to print all messages on the same line, with each new message replacing the old one:

```
# Print each message on a new same line:
for(i in 1:5)
{
    cat("Step", i, "out of 5\n")
    Sys.sleep(1) # Sleep for 1 second
}

# Replace the previous message with the new one:
for(i in 1:5)
{
    cat("Step", i, "out of 5\r")
    Sys.sleep(1) # Sleep for one second
}
```

Adding a progress bar is a little more complicated, because we must first start the bar by using `txtProgressBar` and then update it using `setTxtProgressBar`:

```
sequence <- 1:5
pbar <- txtProgressBar(min = 0, max = max(sequence), style = 3)
for(i in sequence)
{
    Sys.sleep(1) # Sleep for 1 second
    setTxtProgressBar(pbar, i)
}
close(pbar)
```

Finally, the `beepr` package[43] can be used to play sounds, with the function `beep`:

```
install.packages("beepr")

library(beepr)
# Play all 11 sounds available in beepr:
for(i in 1:11)
{
    beep(sound = i)
    Sys.sleep(2) # Sleep for 2 seconds
}
```

6.4.4 Loops and lists

In our previous examples of loops, it has always been clear from the start how many iterations the loop should run and what the length of the output vector (or data frame) should be. This isn't always the case. To begin with, let's consider the case where the length of the output is unknown or difficult to know in advance. Let's say that we want to go through the `airquality` data to find days that are extreme in the sense that at least one variable attains its maximum on those days. That is, we wish to find the indices of the maximum of each variable, and store them in a vector. Because several days can have the same temperature or wind speed, there may be more than one such maximal index for each variable. For that reason, we don't know the length of the output vector in advance.

In such cases, it is usually a good idea to store the result from each iteration in a `list` (Section 5.2), and then collect the elements from the list once the loop has finished. We can create an empty list with one element for each variable in `airquality` using `vector`:

```
# Create an empty list with one element for each variable in
# airquality:
max_list <- vector("list", ncol(airquality))

# Naming the list elements will help us see which variable the maximal
# indices belong to:
names(max_list) <- names(airquality)

# Loop over the variables to find the maxima:
for(i in seq_along(airquality))
{
      # Find indices of maximum values:
      max_index <- which(airquality[[i]] == max(airquality[[i]],
                                                na.rm = TRUE))

      # Add indices to list:
      max_list[[i]] <- max_index
}

# Check results:
max_list

# Collapse to a vector:
extreme_days <- unlist(max_list)
```

(In this case, only the variables `Month` and `Days` have duplicate maximum values.)

6.4.5 `while` loops

In some situations, we want to run a loop until a certain condition is met, meaning that we don't know in advance how many iterations we'll need. This is more common in numerical optimisation and simulation, but sometimes it also occurs in data analyses.

When we don't know in advance how many iterations are needed, we can use `while` loops. Unlike `for` loops that iterate a fixed number of times, `while` loops keep iterating as long as some specified condition is met. Here is an example where the loop keeps iterating until `i` squared is greater than 100:

```
i <- 1

while(i^2 <= 100)
{
      cat(i,"squared is", i^2, "\n")
      i <- i +1
}

i
```

The code block inside the loop keeps repeating until the condition `i^2 <= 100` no longer is satisfied. We have to be a bit careful with this condition – if we set it in such a way that it is possible that the condition *always* will be satisfied, the loop will just keep running and running, creating what is known as an *infinite loop*. If you've accidentally created an infinite loop, you can break it by pressing the Stop button at the top of the Console panel in RStudio.

In Section 5.3.3 we saw how `rle` can be used to find and compute the lengths of runs of equal values in a vector. We can use nested `while` loops to create something similar. `while` loops are a good choice here, because we don't know how many runs are in the vector in advance. Here is an example, which you'll study in more detail in Exercise 6.12:

```r
# Create a vector of 0's and 1's:
x <- rep(c(0, 1, 0, 1, 0), c(5, 1, 4, 2, 7))

# Create empty vectors where the results will be stored:
run_values <- run_lengths <- c()

# Set the initial condition:
i <- 1

# Iterate over the entire vector:
while(i < length(x))
{
    # A new run starts:
    run_length <- 1
    cat("A run starts at i =", i, "\n")

    # Check how long the run continues:
    while(x[i+1] == x[i] & i < length(x))
    {
          run_length <- run_length + 1
          i <- i + 1
    }

    i <- i + 1

    # Save results:
    run_values <- c(run_values, x[i-1])
    run_lengths <- c(run_lengths, run_length)
}

# Present the results:
data.frame(run_values, run_lengths)
```

~

Exercise 6.12 Consider the nested `while` loops in the run length example above. Go through the code and think about what happens in each step. What happens when `i` is `1`? When it is `5`? When it is `6`? Answer the following questions:

1. What does the condition for the outer while loop check? Why is it needed?
2. What does the condition for the inner while loop check? Why is it needed?
3. What does the line `run_values <- c(run_values, x[i-1])` do?

(Click here to go to the solution.)

Exercise 6.13 The *control statements* `break` and `next` can be used inside both `for` and `while` loops to control their behaviour further. `break` stops a loop, and `next` skips to the next iteration of it. Use these functions to modify the following piece of code so that the loop skips to the next iteration if `x[i]` is `0`, and breaks if `x[i]` is `NA`:

```
x <- c(1, 5, 8, 0, 20, 0, 3, NA, 18, 2)

for(i in seq_along(x))
{
      cat("Step", i, "- reciprocal is", 1/x[i], "\n")
}
```

(Click here to go to the solution.)

Exercise 6.14 Using the `cor_mat` computation from Section 6.4.2, write a function that computes all pairwise correlations in a data frame and uses `next` to only compute correlations for `numeric` variables. Test your function by applying it to the `msleep` data from `ggplot2`. Could you achieve the same thing without using `next`?

(Click here to go to the solution.)

6.5 Iteration using vectorisation and functionals

Many operators and functions in R take vectors as input and handle them in a highly efficient way, usually by passing the vector on to an optimised function written in the C programming language[44]. So if we want to compute the squares of the numbers in a vector, we don't need to write a loop:

```
squares <- vector("numeric", 5)

for(i in 1:5)
{
    squares[i] <- i^2
}
squares
```

Instead, we can simply apply the `^` operator, which uses fast C code to compute the squares:

```
squares <- (1:5)^2
```

These types of functions and operators are called *vectorised*. They take a vector as input and apply a function to all its elements, meaning that we can avoid slower solutions utilising loops in R[45]. Try to use vectorised solutions rather than loops whenever possible – it makes your code both easier to read and faster to run.

A related concept is *functionals*, which are functions that contain a `for` loop. Instead of writing a `for` loop, you can use a functional, supplying data, a function that should be applied in each iteration of the loop, and a vector to loop over. This won't necessarily make your loop run faster, but it does have other benefits:

- *Shorter code*: functionals allow you to write more concise code. Some would argue that they also allow you to write code that is easier to read, but that is obviously a matter of taste.
- *Efficient*: functionals handle memory allocation and other small tasks efficiently, meaning that you don't have to worry about creating a vector of an appropriate size to store the result.
- *No changes to your environment*: because all operations now take place in the local environment of the functional, you don't run the risk of accidentally changing variables in your global environment.
- *No left-overs*: `for` leaves the control variable (e.g., `i`) in the environment, functionals do not.
- *Easy to use with pipes*: because the loop has been wrapped in a function, it lends itself well to being used in a `|>` pipeline.

Explicit loops are preferable when:

- You think that they are easier to read and write.

- Your functions take data frames or other non-vector objects as input.
- Each iteration of your loop depends on the results from previous iterations.

In this section, we'll see how we can apply functionals to obtain elegant alternatives to (explicit) loops.

6.5.1 A first example with `apply`

The prototypical functional is `apply`, which loops over either the rows or the columns of a data frame[46]. The arguments are a dataset, the margin to loop over (`1` for rows, `2` for columns) and then the function to be applied.

In Section 6.4.1 we wrote a `for` loop for computing the mean value of each column in a data frame:

```
# Compute the mean for each column of the airquality data:
means <- vector("double", ncol(airquality))

# Loop over the variables in airquality:
for(i in seq_along(airquality))
{
      means[i] <- mean(airquality[[i]], na.rm = TRUE)
}
```

Using apply, we can reduce this to a single line. We wish to use the `airquality` data, loop over the columns (margin `2`), and apply the function `mean` to each column:

```
apply(airquality, 2, mean)
```

Rather elegant, don't you think?

Additional arguments can be passed to the function inside `apply` by adding them to the end of the function call:

```
apply(airquality, 2, mean, na.rm = TRUE)
```

~

Exercise 6.15 Use `apply` to compute the maximum and minimum value of each column of the `airquality` data frame. Can you write a function that allows you to compute both with a single call to `apply`?

(Click here to go to the solution.)

6.5.2 Variations on a theme

There are several variations of `apply` that are tailored to specific problems:

- `lapply`: takes a function and vector/list as input and returns a list.
- `sapply`: takes a function and vector/list as input and returns a vector or matrix.
- `vapply`: a version of `sapply` with additional checks of the format of the output.
- `tapply`: for looping over groups, e.g., when computing grouped summaries.
- `rapply`: a recursive version of `tapply`.
- `mapply`: for applying a function to multiple arguments; see Section 6.5.7.
- `eapply`: for applying a function to all objects in an environment.

We have already seen several ways to compute the mean temperature for different months in the `airquality` data (Sections 2.12 and 5.7.7, and Exercise 6.7). The `*apply` family offer several more:

```
# Create a list:
temps <- split(airquality$Temp, airquality$Month)

lapply(temps, mean)
sapply(temps, mean)
vapply(temps, mean, vector("numeric", 1))
tapply(airquality$Temp, airquality$Month, mean)
```

There is, as that delightful proverb goes, more than one way to skin a cat.

∼

Exercise 6.16 Use an `*apply` function to simultaneously compute the monthly maximum and minimum temperature in the `airquality` data frame.

(Click here to go to the solution.)

Exercise 6.17 Use an `*apply` function to simultaneously compute the monthly maximum and minimum temperature and windspeed in the `airquality` data frame.

Hint: start by writing a function that simultaneously computes the maximum and minimum temperature and windspeed for a data frame containing data from a single month.

(Click here to go to the solution.)

6.5.3 `purrr`

If you feel enthusiastic about ~~skinning cats~~ using functionals instead of loops, the tidyverse package `purrr` is a great addition to your toolbox. It contains a number of specialised alternatives to the `*apply` functions. More importantly, it also contains certain shortcuts that come in handy when working with functionals. For instance, it is fairly common to define a short function inside your functional, which is useful for instance when you don't want the function to take up space in your environment. This can be done a little more elegantly with `purrr` functions using a shortcut denoted by `~`. Let's say that we want to standardise all variables in `airquality`. The `map` function is the `purrr` equivalent of `lapply`. We can use it with or without the shortcut, and with or without pipes (we mention the use of pipes now because it will be important in what comes next):

```
# Base solution:
lapply(airquality, function(x) { (x-mean(x))/sd(x) })

# Base solution with pipe:
airquality |> lapply(function(x) { (x-mean(x))/sd(x) })

# purrr solution:
library(purrr)
map(airquality, function(x) { (x-mean(x))/sd(x) })

# We can make the purrr solution less verbose using a shortcut:
map(airquality, ~(.-mean(.))/sd(.))

# purr solution with pipe and shortcut:
airquality |> map(~(.-mean(.))/sd(.))
```

Where this shortcut really shines is if you need to use multiple functionals. Let's say that we want to standardise the `airquality` variables, compute a `summary` and then extract columns `2` and `5` from the summary (which contains the 1st and 3rd quartile of the data):

```
# Impenetrable base solution:
lapply(lapply(lapply(airquality,
                     function(x) { (x-mean(x))/sd(x) }),
              summary),
       function(x) { x[c(2, 5)] })

# Base solution with pipe:
airquality |>
       lapply(function(x) { (x-mean(x))/sd(x) }) |>
       lapply(summary) |>
       lapply(function(x) { x[c(2, 5)] })

# purrr solution:
airquality |>
       map(~(.-mean(.))/sd(.)) |>
       map(summary) |>
       map(~.[c(2, 5)])
```

Once you know the meaning of `~`, the `purrr` solution is a lot cleaner than the base solutions.

6.5.4 Specialised functions

So far, it may seem like `map` is just like `lapply` but with a shortcut for defining functions, which is more or less true. But `purrr` contains a lot more functionals that you can use, each tailored to specific problems.

For instance, if you need to specify that the output should be a vector of a specific type, you can use:

- `map_dbl(data, function)` instead of `vapply(data, function, vector("numeric", length))`,
- `map_int(data, function)` instead of `vapply(data, function, vector("integer", length))`,

- `map_chr(data, function)` instead of `vapply(data, function, vector("character", length))`,
- `map_lgl(data, function)` instead of `vapply(data, function, vector("logical", length))`.

If you need to specify that the output should be a data frame, you can use:

- `map_dfr(data, function)` instead of `sapply(data, function)`.

The `~` shortcut for functions is available for all these `map_*` functions. In case you need to pass additional arguments to the function inside the functional, just add them at the end of the functional call:

```
airquality |> map_dbl(max)
airquality |> map_dbl(max, na.rm = TRUE)
```

Another specialised function is the `walk` function. It works just like `map`, but it doesn't return anything. This is useful if you want to apply a function with no output, such as `cat` or `read.csv`:

```
# Returns a list of NULL values:
airquality |> map(~cat("Maximum:", max(.), "\n"))

# Returns nothing:
airquality |> walk(~cat("Maximum:", max(.), "\n"))
```

~

Exercise 6.18 Use a `map_*` function to simultaneously compute the monthly maximum and minimum temperature in the `airquality` data frame, returning a vector.

(Click here to go to the solution.)

6.5.5 Exploring data with functionals

Functionals are great for creating custom summaries of your data. For instance, if you want to check the data type and number of unique values of each variable in your dataset, you can do that with a functional:

```
library(palmerpenguins)
penguins |> map_dfr(~(data.frame(unique_values = length(unique(.)),
                                 class = class(.))))
```

You can of course combine `purrr` functionals with functions from other packages, e.g., to replace `length(unique(.))` with a function from your favourite data manipulation package:

```
# Using uniqueN from data.table:
library(data.table)
peng <- as.data.table(penguins)
peng |> map_dfr(~(data.frame(unique_values = uniqueN(.),
                             class = class(.))))

# Using n_distinct from dplyr:
library(dplyr)
penguins |> map_dfr(~(data.frame(unique_values = n_distinct(.),
                                 class = class(.))))
```

When creating summaries, it can often be useful to be able to loop over both the elements of a vector and their indices. In `purrr`, this is done using the usual `map*` functions, but with an `i` (for index) in the beginning of their names, e.g., `imap` and `iwalk`:

```
# Returns a list of NULL values:
imap(airquality, ~ cat(.y, ": ", median(.x), "\n", sep = ""))

# Returns nothing:
iwalk(airquality, ~ cat(.y, ": ", median(.x), "\n", sep = ""))
```

Note that `.x` is used to denote the variable, and that `.y` is used to denote the *name* of the variable. If `i*` functions are used on vectors without element names, indices are used instead. The names of elements of vectors can be set using `set_names`:

```
# Without element names:
x <- 1:5
iwalk(x, ~ cat(.y, ": ", exp(.x), "\n", sep = ""))

# Set element names:
x <- set_names(x, c("exp(1)", "exp(2)", "exp(3)", "exp(4)", "exp(5)"))
iwalk(x, ~ cat(.y, ": ", exp(.x), "\n", sep = ""))
```

~

Exercise 6.19 Write a function that takes a data frame as input and returns the following information about each variable in the data frame: variable name, number of unique values, data type, and number of missing values. The function should, as you will have guessed, use a functional.

(Click here to go to the solution.)

Exercise 6.20 In Exercise 6.11 you wrote a function that printed the names and variables for all `.csv` files in a folder given by `folder_path`. Use `purrr` functionals to do the same thing.

(Click here to go to the solution.)

6.5.6 Keep calm and carry on

Another neat feature of `purrr` is the `safely` function, which can be used to wrap a function that will be used inside a functional and makes sure that the functional returns a result even if there is an error. For instance, let's say that we want to compute the logarithm of all variables in the `msleep` data:

```
library(ggplot2)
msleep
```

Note that some columns are `character` vectors, which will cause `log` to throw an error:

```
log(msleep$name)
log(msleep)
lapply(msleep, log)
map(msleep, log)
```

Note that the error messages we get from `lapply` and `map` here don't give any information about which variable caused the error, making it more difficult to figure out what's gone wrong.

If first we wrap `log` with `safely`, we get a list containing the correct output for the numeric variables, and error messages for the non-numeric variables:

```
safe_log <- safely(log)
lapply(msleep, safe_log)
map(msleep, safe_log)
```

Not only does this tell us where the errors occur, but it also returns the logarithms for all variables that `log` actually could be applied to.

If you'd like your functional to return some default value, e.g., `NA`, instead of an error message, you can use `possibly` instead of `safely`:

```
pos_log <- possibly(log, otherwise = NA)
map(msleep, pos_log)
```

6.5.7 Iterating over multiple variables

A final important case is when you want to iterate over more than one variable. This is often the case when fitting statistical models that should be used for prediction, as you'll see in Section 8.2.5. Another example is when you wish to create plots for several subsets in your data. For instance, we could create a plot of `body_mass_g` versus `flipper_length_mm` for each combination of `species` and `sex` in the `penguins` data. To do this for a single combination, we'd use something like this:

```
library(ggplot2)
library(dplyr)
library(palmerpenguins)

penguins |> filter(species == "Chinstrap",
                   sex == "male") |>
              ggplot(aes(body_mass_g, flipper_length_mm)) +
                  geom_point() +
                  ggtitle("Chinstrap, male")
```

To create such a plot for all combinations of `species` and `sex`, we must first create a data frame containing all unique combinations, which can be done using the `distinct` function from `dplyr`:

```
combos <- penguins |> distinct(species, sex) |> na.omit()
all_species <- combos$species
all_sexes <- combos$sex
```

`map2` and `walk2` from `purrr` loop over the elements of two vectors, `x` and `y`, say. They combine the first element of `x` with the first element of `y`, the second element of `x` with the second element of `y`, and so on – meaning that they won't automatically loop over all combinations of elements. That is the reason why we use `distinct` above to create two vectors where each pair (`x[i]`, `y[i]`) corresponds to a combination. Apart from the fact that we add a second vector to the call, `map2` and `walk2` work just like `map` and `walk`:

```
# Print all pairs:
walk2(all_species, all_sexes, ~cat(.x, .y, "\n"))

# Create a plot for each pair:
combos_plots <- map2(all_species, all_sexes, ~{
                    penguins |> filter(species == .x,
                                       sex == .y) |>
                     ggplot(aes(body_mass_g, flipper_length_mm)) +
                            geom_point() +
                            ggtitle(paste(.x, .y, sep =", "))})

# View some plots:
combos_plots[[1]]
combos_plots[[6]]

# Save all six plots in a pdf file, with one plot per page:
pdf("all_combos_plots.pdf", width = 8, height = 8)
combos_plots
dev.off()
```

The base function `mapply` could also have been used here. If you need to iterate over more than two vectors, you can use `pmap` or `pwalk`, which work analogously to `map2` and `walk2`.

∼

Exercise 6.21 Using the `gapminder` data from the `gapminder` package, create scatterplots of `pop` and `lifeExp` for each combination of `continent` and `year`. Save each plot as a separate `.png` file.

(Click here to go to the solution.)

6.6 Measuring code performance

There are probably as many ideas about what good code is as there are programmers. Some prefer readable code; others prefer concise code. Some prefer to work with separate functions for each task, while others would rather continue to combine a few basic functions in new

ways. Regardless of what you consider to be *good code*, there are a few objective measures that can be used to assess the quality of your code. In addition to writing code that works and is bug-free, you'd like your code to be:

- *Fast*: meaning that it runs quickly. Some tasks can take seconds or weeks, depending on what code you write for them. Speed is particularly important if you're going to run your code many times.
- *Memory efficient*: meaning that it uses as little of your computer's memory as possible. Software running on your computer uses its memory – its RAM – to store data. If you're not careful with RAM, you may end up with a full memory and a sluggish or frozen computer. Memory efficiency is critical if you're working with big datasets that take up a lot of RAM to begin with.

In this section we'll have a look at how you can measure the speed and memory efficiency of R functions. A caveat is that while speed and memory efficiency are important, the most important thing is to come up with a solution that works in the first place. You should almost always start by solving a problem, and then worry about speed and memory efficiency, not the other way around. The reason for this is that efficient code often is more difficult to write, read, and debug, which can slow down the process of writing it considerably.

Note also that speed and memory usage is system-dependent. The clock frequency and architecture of your processor and speed and size of your RAM will affect how your code performs, as will what operating system you use and what other programs you are running at the same time. That means that if you wish to compare how two functions perform, you need to compare them on the same system under the same conditions.

As a side note, a great way to speed up functions that use either loops or functionals is parallelisation. We cover that topic in Section 12.2.

6.6.1 Timing functions

To measure how long a piece of code takes to run, we can use `system.time` as follows:

```
rtime <- system.time({
      x <- rnorm(1e6)
      mean(x)
      sd(x)
})

# elapsed is the total time it took to execute the code:
rtime
```

This isn't the best way of measuring computational time though and doesn't allow us to compare different functions easily. Instead, we'll use the `bench` package, which contains a function called `mark` that is very useful for measuring the execution time of functions and blocks of code. Let's start by installing it:

```
install.packages("bench")
```

In Section 6.1.1 we wrote a function for computing the mean of a vector:

```
average <- function(x)
{
      return(sum(x)/length(x))
}
```

Is this faster or slower than `mean`? We can use `mark` to apply both functions to a vector multiple times, and measure how long each execution takes:

```
library(bench)
x <- 1:100
bm <- mark(mean(x), average(x))
bm # Or use View(bm) if you don't want to print the results in the
   # Console panel.
```

`mark` has executed both function `n_itr` times each, and measured how long each execution took to perform. The execution time varies – in the output, you can see the shortest (`min`) and median (`median`) execution times, as well as the number of iterations per second

(`itr/sec`). Be a little wary of the units for the execution times so that you don't get them confused – a millisecond (ms, 10^{-3} seconds) is 1,000 microseconds (µs, 1 µs is 10^{-6} seconds), and 1 microsecond is 1,000 nanoseconds (ns, 1 ns is 10^{-9} seconds).

The result here may surprise you – it appears that `average` is faster than `mean` ! The reason is that `mean` does a lot of things that `average` does not: it checks the data type and gives error messages if the data is of the wrong type (e.g., `character`), and then traverses the vector twice to lower the risk of errors due to floating point arithmetics. All of this takes time and makes the function slower (but safer to use).

We can plot the results using the `ggbeeswarm` package:

```
install.packages("ggbeeswarm")

plot(bm)
```

It is also possible to place blocks of code inside curly brackets, `{ }` , in `mark` . Here is an example comparing a vectorised solution for computing the squares of a vector with a solution using a loop:

```
x <- 1:100
bm <- mark(x^2,
    {
        y <- x
        for(i in seq_along(x))
        {
            y[i] <- x[i]*x[i]
        }
        y
    })
bm
plot(bm)
```

Although the above code works, it isn't the prettiest, and the `bm` table looks a bit confusing because of the long expression for the code block. I prefer to put the code block inside a function instead:

```
squares <- function(x)
{
        y <- x
        for(i in seq_along(x))
        {
            y[i] <- x[i]*x[i]
        }
        return(y)
}

x <- 1:100
bm <- mark(x^2, squares(x))
bm
plot(bm)
```

Note that `squares(x)` is faster than the original code block:

```
bm <- mark(squares(x),
    {
        y <- x
        for(i in seq_along(x))
        {
            y[i] <- x[i]*x[i]
        }
        y
    })
bm
```

Functions in R are *compiled* the first time they are run, which often makes them run faster than the same code would have outside of the function. We'll discuss this further next.

6.6.2 Measuring memory usage (and a note on compilation)

`mark` also shows us how much memory is allocated when running different code blocks, in the `mem_alloc` column of the output[47].

Unfortunately, measuring memory usage is a little tricky. To see why, restart R (yes, really, this is important!), and then run the following code to benchmark `x^2` versus `squares(x)`:

```
library(bench)

squares <- function(x)
{
        y <- x
        for(i in seq_along(x))
        {
            y[i] <- x[i]*x[i]
        }
        return(y)
}

x <- 1:100
bm <- mark(x^2, squares(x))
bm
```

Judging from the `mem_alloc` column, it appears that the `squares(x)` loop not only is slower, but it also uses more memory. But wait! Let's run the code again, just to be sure of the result:

```
bm <- mark(x^2, squares(x))
bm
```

This time, both functions use less memory, and `squares` now uses *less* memory than `x^2`. What's going on?

Computers can't read code written in R or most other programming languages directly. Instead, the code must be translated to *machine code* that the computer's processor uses, in a process known as *compilation*. R uses *just-in-time compilation* of functions and loops[48], meaning that it translates the R code for new functions and loops to machine code *during execution*. Other languages, such as C, use ahead-of-time compilation, translating the code *prior to execution*. The latter can make the execution much faster, but some flexibility is lost, and the code needs to be run through a compiler ahead of execution, which also takes time. When doing the just-in-time compilation, R needs to use some of the computer's memory,

which causes the memory usage to be greater the first time the function is run. However, if an R function is run again, it has already been compiled, meaning R doesn't have to allocate memory for compilation.

In conclusion, if you want to benchmark the memory usage of functions, make sure to run them once before benchmarking. Alternatively, if your function takes a long time to run, you can compile it without running it using the `cmpfun` function from the `compiler` package:

```
library(compiler)
squares <- cmpfun(squares)
squares(1:10)
```

~

Exercise 6.22 Write a function for computing the mean of a vector using a `for` loop. How much slower than `mean` is it? Which function uses more memory?

(Click here to go to the solution.)

Exercise 6.23 We have seen three different ways of filtering a data frame to only keep rows that fulfil a condition: using base R, `data.table`, and `dplyr`. Suppose that we want to extract all flights from 1 January from the `flights` data in the `nycflights13` package:

```
library(data.table)
library(dplyr)
library(nycflights13)
# Read about the data:
?flights

# Make a data.table copy of the data:
flights.dt <- as.data.table(flights)

# Filtering using base R:
flights0101 <- flights[flights$month == 1 & flights$day == 1,]
# Filtering using data.table:
flights0101 <- flights.dt[month == 1 & day == 1,]
# Filtering using dplyr:
flights0101 <- flights |> filter(month ==1, day == 1)
```

Compare the speed and memory usage of these three approaches. Which has the best performance?

(Click here to go to the solution.)

41. Do you *really*?↩
42. `base` is automatically loaded when you start R and contains core functions such as `sqrt`.↩
43. Arguably the best add-on package for R.↩
44. Unlike R, C is a low-level language that allows the user to write highly specialised (and complex) code to perform operations very quickly.↩
45. The vectorised functions often use loops, but loops written in C, which are much faster.↩
46. Actually, over the rows or columns of a matrix - `apply` converts the data frame to a `matrix` object.↩
47. But only if your version of R has been *compiled with memory profiling*. If you are using a standard build of R, i.e., have downloaded the base R binary from R-project.org, you should be good to go. You can check that memory profiling is enabled by checking that `capabilities("profmem")` returns `TRUE`. If not, you may need to reinstall R if you wish to enable memory profiling.↩
48. Since R 3.4.↩