Chapter 1

Programming

Even if R's and our focus is on statistical matters, R can be used for programming. Programming here means to use concepts that are encountered in other traditional programming languages, such as loops or branching, to perform complex and repeated operations that can depend on input data and produce output data.

Using R for statistical calculations already falls within this definition, but is generally on a high level of abstraction and the detailed implementation may be not apparent. However, with knowledge of the traditional programming toolbox, it is possible to adapt and extend existing R code or to write more advanced and suitable solutions that one needs. For people without programming experience the knowledge can also be handy when they want to move on to other languages such as python or Java.

Strictly, R is a scripting language because it is "interpreted": Instead of directly giving instructions to the computer processor, another program reads the script code that we write and translates it into instructions that are sent to the processor.

R provides all the common programming constructs needed to control the flow of a program execution: As you now, R scripts are executed beginning with the first line, then the next line, until the end of the file. This is not very flexible, but the program execution can be *c*ontrolled to follow any desired pattern.

Mainly three constructs are used to that purpose: *branching*, *loops*, and *functions*.

Object orientation is one other concept that is generally important in programming. R itself features object orientation but we do not treat this topic in this chapter.

1.1 Branching

Branching, or conditional statements, means that a condition is checked before a certain section of code is run. These conditions need to result in either `TRUE` or `FALSE`. You can imagine a river that splits into two where the water flows down the one branch if the condition is `TRUE` and the other branch if its `FALSE`.

The branching is done putting `if else` statements in the code. `if` is followed by the condition in brackets, then by curly brackets that surround the code that should be executed depending on whether the condition is true.

```
x <- sample(1:3, 1)
print(x)
```

```
## [1] 3

if(x == 1){
  print("x is 1")
}
```

If statements can optionally be followed by `else if` or `else` statements. `else if` statements work similarly to if statements.`else` statements are executed none of the preceeding `if` or `else if` conditions result in `TRUE`:

```
if(x == 1){
  print("condition 1 is true, x is 1")
} else if(x == 2) {
  print("condition 2 is true, x is 2")
} else{
  print("Condition 1 and 2 are not true, x must be 3")
}

## [1] "Condition 1 and 2 are not true, x must be 3"
```

Note that `else` and `else if` statements need to be on the same line as the closing curly bracket of the previous code block.

Branching is very important in programming because it allows programs to be flexible. For example, an R can check if a package that should be loaded is already installed, electric cars can play sounds if the battery level becomes low, social media platforms can check if uploaded pictrues are appropriate, and countless more. Of course, all these examples involve much more than simple *if - else* checks.

Own experimentation

What happens if you run?

```
if(log(-1) > 0){
  print("?")
}
```

Try to fix the problem by using `is.na()`.

1.2 Loops

Loops are another important mean to control the program execution flow. They allow to blocks of code to be repeated. Similar to *if*s they need to be tied to a condition because one wants to avoid loops to run forever. The two most common loop constructs are`for` and `while` loops.

`while` loops reapeats a block of code until the associated condition evaluates to `FALSE`. They are useful to in situations where the exact number of repetitions is not known beforehand:

```
x <- 10
while(x >= 5){
 print(x)
 x <- sample(1:10, 1)
}

## [1] 10
## [1] 9
```

This loop could run for a very ong time if one is unlucky, but this scenorio is unlikely.

`for` loops in R traverse over the elements of a vector. In the background, R checks after every repetiotion if the end of the vector is already reached:

```
for(i in 1:10){
  print(i)
}

## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

This example prints 1 to 10 to the console.

In general, loops in R should be avoided as much as possible. Instead, the provided functions that use vectors as inputs should be used if available because it is usually much faster and requires much less code. Also consider using the `apply()` family of functions instead of loops.

However, sometimes you may feel more comfortable with loops than with alternatives. Loops are as perfomant as `apply()` functions and can make code easier understand and write in some situations.

To demonstrate the differences in performance between the loop variants and vectorized functions, we write a little program to calculate the time it takes to calculate the sum of a large vector.

```
v = sample(100, 1000000, replace = T)

startTime <- Sys.time() # save current time point
i <- 1 # i stands for index
cs <- 0 # cs: current sum, start with 0

# to get the total sum of v, we retrieve each element in the loop
# and add it to the current sum
while(i < length(v)){ # run until i >= length of v

  cs <- cs+v[i] # add the number at index i
  i <- i +1 # increase the index to progress
}
endTime <- Sys.time() # save the current time point

print(paste0("Sum is: ", cs)) # print the current sum (use paste0 to )

## [1] "Sum is: 50443536"

print(paste0("Time for while loop: ", round(endTime - startTime,3), " seconds"))

## [1] "Time for while loop: 1.783 seconds"

startTime <- Sys.time()
cs <- 0 # set cs to 0 again
for(i in v){ # in the for loop, we automatically get the next element i in  v
  cs <- cs+i # add i to current sum
```

```
}
endTime <- Sys.time()
print(paste0("Sum is: ", cs))

## [1] "Sum is: 50443570"

print(paste0("Time for for loop: ", round(endTime - startTime,3), " seconds"))

## [1] "Time for for loop: 0.536 seconds"

startTime <- Sys.time()
cs <- sum(v)
endTime <- Sys.time()
print(paste0("Sum is: ", cs))

## [1] "Sum is: 50443570"

print(paste0("Time for for sum: ", round(endTime - startTime,3), " seconds"))

## [1] "Time for for sum: 0 seconds"
```

Two other type of loops are the `dowhile()` loop and the `repeat` loop. These are not as commonly used and they will not be explained here.

Own experimentation

Use the help to read about `next` and `break` (run ?next) and what they do in a loop.
What do you think are advantages and disadvantages of using `break` and `next` statements inside loops?

1.3 Functions

By now, you already have encountered a large number of functions in R, and probably got a good idea of why they are useful. They allow to encapsulate specific functionality and to make it reusable in varying contexts. Of course, R also allows you to define your own functions.

The defining feature of functions are their brackets. Functions take zero or more arguments in round brackets that follow the function name. These arguments allow to pass data from where the function is called to the code inside the function. This means that the code inside the functions sees the value of the objects that were given to the function when calling it.

Functions are stored in objects in R. In their definition, the object name is followed by the equal sign, the special keyword `function` and then the a comma-separated list of arguments in round brackets. The code of the function follows after that enclosed by curly brackets. This section is called the function body:

```
myFunction <- function(arg1, arg2){
  print("Hello, this is my function. The weighted sum of arguments is: ")
  print(arg1*2+arg2)
}
```

To execute the function, write the name, followed by the comma-separated argument list in brackets:

```
number <- 4
myFunction(4,number)

## [1] "Hello, this is my function. The weighted sum of arguments is: "
## [1] 12
```

Now the `myFunction` sees arg1 as 4 and arg2 as 12.

These arguments are just normal objects and can be used inside the function as such. However, modifying them wont alter the value of the original object that was passed to the function. The function just operates on a copy of the original object:

```
number <- 4
myFunction2 <- function(number){
  number <- 5 # last used value will be returned (v)
}
print(myFunction2())

## [1] 5

number # what is number?

## [1] 4
```

Note that 5 and 4 were printed, and not two times 5.

1.3.1 Anonymous functions

In R, functions are objects comparable to objects that just hold values. Because of that, it is straightforward to use functions in a similar way as objects. Specifically, functions are only executed when followed by () brackets, and otherwise treatetd similarly to objects:

```
afunction <- function(number){
  print("Hello")
}
bfunction <- afunction # use function as object
bfunction() # execute functions with ()

## [1] "Hello"
```

This allows for a lot of flexibility, because one can pass functions to other functions or one can directly specify function bodies without function names, so called *anonymous* functions:

```
cfunction <- function(x){
  print(paste0("Value of x is: ", x))
}
sapply(1:3, cfunction) # pass funtion as an argument

## [1] "Value of x is: 1"
## [1] "Value of x is: 2"
## [1] "Value of x is: 3"
## [1] "Value of x is: 1" "Value of x is: 2" "Value of x is: 3"

sapply(1:3, function(x){ x }) # use anonymous function

## [1] 1 2 3
```

1.3.2 Return values

Functions can return values that can be assigned to objects. In R, it is sufficient to just write the name of the value or object that should be returned:

```
myRetFunction <- function(){
  astring <- "blue"
  astring # return astring
}
color <- myRetFunction()
color

## [1] "blue"
```

This function without arguments returns the character sequence "blue".

In general, the return value of a function is the result of the last executed expression. This is way the above example works. It is possible to explicitly return values by using the `return()` statement, which excepts the return value as an argument:

```
lazySum <- function(x){
  # demonstrate explicit returns using return ()
  if(weekdays(Sys.time()) == "Sunday")
  {
    return("I dont work on Sundays")
  }
  else
  {
    return(sum(x))
  }

  "I don't know"
}
```

Explicit `return()` statements are useful if not the whole function body should be executed.

In R, it is possible to refer indicate the arguments of functions by their name when calling the function:

```
myFunction(arg2 = 2, arg1 = 1) # indicate arguments by name

## [1] "Hello, this is my function. The weighted sum of arguments is: "
## [1] 4
```

If no names are provided, the arguments values or objects are assigned to the arguments according to the order that they are listed. Eeven if not always necessary, indicating which argument is meant by using the '=' syntax may make an R script much easier to understand.

It is also possible to provide default values for function arguments when defining functions. This makes it optional to specify the value for this argument. Default values should preferable used for the last arguments in the function definition. This way, it is possible to leave out the argument with the default value when calling the function, even when not specifying all other arguments by name:

```
# provide a default argument
myDefArgFunction <- function(arg1, arg2, arg3=2){
  print(sum(arg1+arg2+arg3))
}
myDefArgFunction(2, 3) # only needs two arguments
```

```
## [1] 7
```

1.3.3 Variable scope

The objects created in R functions are local to those function, meaning that they cannot be used outside:

```
f <- function(){
  obj <- "object"
}
f()
print(obj)

## Error in print(obj):  object 'obj' not found

# we get an error!
```

Objects that are available where a function is called from are also accessible inside of the the function:

```
x <- 10
f <- function(){
  print(x)
}
f()

## [1] 10
```

Such objects from parent contexts are also called *global* objects. They can not, however, be altered using `<- or =`, `x` will not change to 77 in the following example:

```
x <- 10
f <- function(){
  x <- 77
}
f()
x

## [1] 10
```

It is possible to do so using the «- operator, but this is not recommended in most circumstances. The reason for is that the code inside a function should in general only be related through the arugments and the return value to whathever other code calling it. Changing global objects from within a function violates this principle and can it harder to understand the code or find problems i case of unexpected behavior.

Two other type of loops are the `dowhile()` loop and the `repeat` loop. These are not commonly used and they will not be explained here.

Own experimentation

Write a function testNA that takes two arguments: The first is another function, and the second a numerical value. The function should return `TRUE` if the function that is passed as the first arguments returns NA for the numerical value in the second argument. Otherwise it should return `FALSE`.
Test your function with `log(-1)` and `log(1)`.