1 Programming in R

There is a huge corpus of predefined functions in R, but it is also possible to write your own R functions. In this way you may execute the same commands as often as you want while changing parameter values. Functions are R objects as any other object.

1.1 Writing your own R functions

An R function is a sequence of instructions involving parameters and returning an R object at the end. In general, functions are always developed in an R script (or in Jupyter) and not directly in the console.

To declare a function, we use the function `function`. The general syntax is the following:

```
NameOfTheFunction <-  function(arg1,...,argk){
     sequence of instructions
     output <- ...
     return(output)
}
```

Here `arg1, ..., argk` are the names of the arguments (or parameters) of the function. You can use as many arguments as you want, all separated by a comma. The instruction `return(output)` at the end of the instructions returns the object `output`. Notice that you can use several arguments, but only a single object is returned at the end.

To indicate the beginning and the end of the body of instructions, delimit them by curly braces { and }. You can omit the curly braces only if the body of the function consists of a single instruction.

When executing the function in the console (or in Jupyter), the function `function` creates an object that is assigned to the variable `NameOfTheFunction`. Now you can use this function `NameOfTheFunction` by calling its name and adding arguments in paratheses.

Here comes a little example:

```
BMI <- function(weight,height) {
    res <- weight/height^2
    return(res)
}
```

The function's name is `BMI`. The function takes two arguments, `height` and `weight`, and it returns the corresponding body mass index.

To be able to use the function `BMI` in the console, we have to submit the function to R, that is we have to execute it once in the console (or in Jupyter).

Now we can use the function `BMI` in the current session like any other function in R:

```
BMI(90,1.85)

[1] 26.29657
```

You may assign default values to the arguments of the function. This is very simple as you can see from the example:

```
BMI <- function(weight=60,height=1.75) {
    res <- weight/height^2
    return(res)
}
```

As we have changed the definition of the function `BMI`, we have to submit it again to the console. Now, when we call the function without any arguments, i.e. with empty parentheses, the default values are used:

```
BMI()

[1] 19.59184
```

If you want to use a different value for the height, you may just type:

```
BMI(height=1.88)

[1] 16.97601
```

Or you use the less economic writing:

```
BMI(60,1.88)

[1] 16.97601
```

1.2 Object `list`

A list is a collection of objects that are **not necessarily of the same type**. This is useful for functions that return a single object only. Indeed, most `R` functions return lists.

A list can be created by using the function `list()`. Example: Create a list called `useless`:

```
v <- 10:1
x <- pi
word <- "Wow!"
useless <- list(vec = v, nombr = x, fn_BMI = BMI, mot = word)
useless

$vec
 [1] 10  9  8  7  6  5  4  3  2  1

$nombr
[1] 3.141593
```

```
$fn_BMI
function (weight = 60, height = 1.75)
{
    res <- weight/height^2
    return(res)
}

$mot
[1] "Wow!"
```

Let's check the type of the object `useless`:

```
typeof(useless)

[1] "list"
```

The elements of a list typically have names, that are defined when creating the list within the call of `list`. We use the function `names()` to get the names of an existing list and also to modify them:

```
names(useless)

[1] "vec"    "nombr"  "fn_BMI" "mot"
```

To access the elements of a list use the symbol $:

```
useless$mot

[1] "Wow!"
```

To access the third element of the vector `useless$vec` do:

```
useless$vec[3]

[1] 8
```

To use the function `useless$fn` do:

```
useless$fn_BMI(100,1.5)

[1] 44.44444
```

1.3 Control structures

In general, any more involved computations require the control of the flow of the program, that is the control of the direction to go within a program. Very often, different values of

parameters or variables require different actions. It is also frequent that some instructions may be executed several times. For all this, we have to use control structures in the program. As in any other programming languages, `R` integrates in its language the structure `if else`, and the loops `for` and `while`.

1.3.1 The structure `if else`

The structure `if else` is used to treat different cases differently. Depending on a logical `condition` that can be verified or not, either the instructions `action1` are performed or the instructions `action2`. The general syntax of the structure `if else` is

```
if (condition) {action1} else {action2}
```

or in the particular case where `action2` consists in doing nothing, the instruction simplifies to

```
if (condition) {action1}
```

Here is another elementary example:

```
age <- 17
if (age < 18) {x <- 'minor'} else {x <- 'adult'}
x

[1] "minor"
```

In practice, the condition `condition` is often the result of logical operations using the logical operators. The syntax of the elementary logical conditions is the following:

```
age < 18   # age strictly lower than 18

[1] TRUE

age <= 18  # age lower than or equal to 18

[1] TRUE

age == 18  # Attention: age = 18 is not a condition but an assignement

[1] FALSE

age != 18  # age not equal to 18

[1] TRUE
```

Certainly, conditions can be more complex by combining elementary conditions:

```
cond1 && cond2  # is TRUE if both cond1 AND cond2 are TRUE.
cond1 || cond2  # is TRUE if at least cond1 is TRUE OR cond2 is TRUE.
!cond1          # Is TRUE if cond1 is FALSE.
```

Here a last example for the structure `if else`:

```
BMI <- function(weight,height) {
    val <- weight/height^2
    if (val>=18.5 && val<=25 ){
        say <- "You have a healthy weight. Everything's fine!"
    }
    else{
        say <- "Take care of your health!"
    }
    return(list(BMI=val, comment=say))
}

BMI(58,1.70)

$BMI
[1] 20.0692

$comment
[1] "You have a healthy weight. Everything's fine!"

BMI(48,1.70)

$BMI
[1] 16.609

$comment
[1] "Take care of your health!"
```

1.3.2 The `for` loop

To repeat the same instructions by varying the value of a parameter, we use the `for` loop. More precisely, in the `for` loop the `instructions` are repeated consecutively for all elements `v` of the vector `vecteur`. The general syntax is

`for (v in vecteur) {instructions}`

Here are two basic examples using the `for` loop:

```
for (i in 1:3){print(i)}

[1] 1
[1] 2
[1] 3
```

```
vec <- 2:5
for (k in 1:length(vec)){
  vec[k] <- vec[k]^k
  }
vec

[1]   2   9  64 625
```

Whenever possible, avoid loops in R as they are expensive in terms of computing time. In general it is much faster to use vectorized operations. In this sens, the last example is a very bad example for the `for` loop, since the same computation is done much faster by the following instruction

```
vec <- 2:5
vec^(1:length(vec))

[1]   2   9  64 625
```

An example where the `for` loop is indispensable is the Fibonacci sequence, which is the sequence of integers where each term is the sum of the two preceding terms. The initial terms are 0 and 1. Here is an R function to compute the first `m` terms of the Fibonacci sequence:

```
fibonacci <- function(m){
  # for the for loop we need m>=3 :
  if (m<3) {m=3}

  # create a vector of length m whose 2 first elements are
  # the values of the first 2 terms of the Fibonacci sequence:
  Fib_seq <- 0:(m-1)
  for (k in 3:m){
    # compute the k-th term of the sequence:
    Fib_seq[k] <- Fib_seq[k-1]+Fib_seq[k-2]
  }
  return(Fib_seq)
}

fibonacci(25)

 [1]     0     1     1     2     3     5     8    13    21    34    55
[12]    89   144   233   377   610   987  1597  2584  4181  6765 10946
[23] 17711 28657 46368
```

1.3.3 The `while` loop

Iterative mathematical methods often require the execution of the same instructions until convergence. In this case, the number of iterations is not known in advance. Thus, the `for`

loop is inappropriate and we rather use the `while` loop that relies on a logical condition `condition` that is evaluated at each iteration and the `instructions` are executed as long as the `condition` is verified. The general syntax is

```
while (condition) {instructions}
```

Here is a simple example:

```
a <- 2
while (a < 100)
  {a <- a^2 }
a

[1] 256
```

Let's consider again the Fibonacci sequence. We can write a function to compute the minimal number of terms to exceed a given value:

```
howmany_fibonacci <- function(M){
  if (M<=1) {howmany <- 'Call the function for a value > 1'}
  else{
      # initialisation of the first 2 terms of the sequence:
      term_kmoins1 <- 0
      term_k <- 1
      # current length of the sequence:
      k <- 2

      # repeat the following instructions while the last term
      # is lower than the threshold  M:
      while (term_k < M){
        # compute the next term of the sequence:
        term_kplus1 <- term_k+term_kmoins1
        # update the length of the sequence:
        k <- k+1
        # update the last 2 terms of the sequence:
        term_kmoins1 <- term_k
        term_k <- term_kplus1
      }
      howmany <- k
  }

  return(howmany)
}

howmany_fibonacci(10)

[1] 8
```

```
howmany_fibonacci(10^6)
```

```
[1] 32
```

When using the `while` loop, make sure that the condition will be verified at a moment. Otherwise, the program will run indefinitely.