1 Introduction

Some facts R is a programming language and therefore ruled by a set of

- Syntax (the form, HOW to write)
- Semantic (the meaning, WHAT to do)

The best way to learn is to practice conversation!

There are (mainly) two kinds of programming language:

- compiled languages. In this case, your "text" (or set of commands) is translated/compiled once by a compiler into a file which can be directly understood by the processor many times
- interpreted languages. Here, the commands are given directly to a software that translates on the fly your instructions.

As a rule of thumb compiled languages are often faster to execute on a computer, but the compilation can be cumbersome and time consuming. On the other hand interpreted languages avoid the compilation process and makes it easier to "try and see" without any running time optimisation.

2 Data types

Without going into details, we review here what are the different ways to handle variables and/or parameters in R. Working with such structures or objects will be detailed in a following section.

2.1 Basic data types

R has three basic data types:

- Numeric; corresponds to any number values whether integer or real numbers (floating point). This can be strange to anyone who has done a bit of programmation before.

- Character; corresponds to any chain of alphanumerical characters delimited by quotes:

  "This is a character"
  "This 1 too"
  "a"

- Boolean; may take the value TRUE or FALSE. There are used mainly to check consistency of variables (do you get what you were expecting?), as a test for loop (Do I need to go through the loop once again?) or when defining functions (useOtherMethod=TRUE).
Table 1: Example of both a numeric and a boolean vector of size 5

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.9</td>
<td>0.6</td>
<td>0.3</td>
<td>0.95</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

2.2 Advanced data structures and objects

These kinds of structures help the user organize one’s data. We can consider different ways of representing it.

Scalars are used to represent a single piece of information (shall it be a numeric 5, a character "character" or a boolean TRUE or FALSE). This kind of structure is useful for instance when considering a counter (in a loop) or a test, but rather unused for large data matrices.

Vectors are used to represent a set of data as a column. All of the values inside this vector have to be of the same kind. For instance, we could get only the values of a vector which are higher than a certain threshold; this would return a boolean vector, as illustrated in Table 1

Exercise 1. Try to create a numeric vector and automatically compute the associated boolean vector as in Table 1.

Matrices and more generally multidimensional array are used to gather pieces of information of the same kind in a multidimensional manner. A matrix is a two dimensional structure with \( m \) rows and \( n \) columns (\( m \) and \( n \) being defined by the user) while an array can have more than two dimensions.

\[
M = \begin{pmatrix}
M(1,1) & \cdots & M(1,n) \\
\vdots & \ddots & \vdots \\
M(m,1) & \cdots & M(m,n)
\end{pmatrix}
\]

More complex structures can be used and are defined as data objects.

Lists are (ordered) sets of information of different kinds associated to a key

\[
\text{myStudy} <- \text{list(first.name="John", last.name="Doe", student.no=123, courses.no=2, courses.names=c("MATH-680", "MATH-330"), courses.grades=c(1.0,1.7) )}
\]

creates a list with different argument names and values. The last name of the student can be access by either the name of the parameter or as a proper variables or by its position in the list

\[
\text{myStudy}["last.name"]
\]
\[
\text{myStudy}$last.name
\]
\[
\text{myStudy}[2]
\]

Data frames can be seen as a generalization of lists in a matrix form. In such objects different vectors of the same size and of (potentially) different types are put next to each other. It is also common use to name the columns. For instance consider the case where you are in charge of a database of many students of the previous form, you could stack them together in form of a table with column names given by the attributes name (see Table 2)

<table>
<thead>
<tr>
<th>Student number</th>
<th>First name</th>
<th>Last name</th>
<th>Nb Courses</th>
<th>Courses names</th>
<th>Grades</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>John</td>
<td>Doe</td>
<td>2</td>
<td>{MATH-680,MATH-330}</td>
<td>{1.0, 1.7}</td>
</tr>
<tr>
<td>124</td>
<td>Foo</td>
<td>Bar</td>
<td>1</td>
<td>{XX-11}</td>
<td>{NA}</td>
</tr>
</tbody>
</table>

In this case, we have also used the student number (which should be unique) as identifier for a particular row of the table (this identifier is not necessarily mandatory but becomes handy in many cases).
3 Basic operators

R comes with a rather exhaustive package of functions one can use to work with the diverse data one has. We try to give here the basics needed to start working with R. More details can be found online on the R package page. We will also see in a later section how one can use third party packages which implement more advanced functions for particular purposes.

Two important commands that we use often when working with R (and with any other programming languages!!) are the comment # and help ? functions. Whatever is written after a # character and before a line break will be ignored (bla=5 #this will be ignore) while the question mark asks for help about a function, for instance ?c. Don’t be shy, use this valuable resource before looking (asking?) anywhere else.

**Arithmetic operations** are defined as binary operators operating on two variables. Most of the basic operations are implemented using rather natural notations.

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Add +</th>
<th>Sub −</th>
<th>Mult *</th>
<th>Div /</th>
<th>Int. div %/%</th>
<th>Modulo %%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>NOT !</td>
<td>OR</td>
<td>( or</td>
<td></td>
<td>)</td>
<td>AND &amp; (&amp;&amp;)</td>
</tr>
<tr>
<td>Useful</td>
<td>Exp. ^</td>
<td>sin</td>
<td>cos</td>
<td>exp</td>
<td>sqrt</td>
<td>% * %</td>
</tr>
</tbody>
</table>

Most of the calculus can be done using such basic operators (note that the last one stands for the matrix multiplication operator).

**Logical operators** are used to control ending conditions in loops for instance or to access only a subset of indices useful for further processing. We have access to all comparison possible as well as boolean operations.

**Creating and assigning variables** is an important part of any programming language. In order to process your data you will need to assign them to a variable.

The assignment can be done using either the assignment operator <- or the assign() function or, in the latest release (will not work on earlier version of R) using the classical = operators. Hence

```r
a <- 5
assign("a",5)
a = 5
```

are three equivalent ways to assign the value 5 to the scalar variable a. Now a is known in the workspace and can be understood by the interpreter as having the value 5 (as a proof, trying displaying its value in the R interpreter)

Note that this assignment works for any scalar (in a wide sense) and myFirstString="Characters" would work exactly fine.

Creating vectors rely on the built-in function c(). It stands for concatenate as it works by concatenating the values passed as argument; these values have to be of the same kind. myFirstVector=c(1,2,3).

**Exercise 2.** *Is the following set of operations equivalent to the previous vector assignment and creation? Why? What is the difference?*

```r
aScalar = 1
anotherScalar <- 3
assign("myFirstVector", c(aScalar, 2, anotherScalar))
```

Array and matrices are created using the matrix() and array() functions. They work by giving them a vector of values as first argument and a vector of dimension (integers) as second argument. For instance

```r
myMatrix=matrix(1:20, c(4,5))
```

creates an array with 4 rows and 5 columns containing the integers from 1 to 20 ordered in a column first basis (1 to 4 in the first column then 5 to 8 in the second and so on)

You could also define a three dimensional matrix by adding a third component to the vector of dimensions. threeDArray=matrix(1:20, c(2,2,5)) creates a stack of 5 2×2 matrices. The first one will contain the numbers 1 to 4 and the last one the numbers 17 to 20.
As we said earlier, data.frames are important when programming with R as they allow to deal with large databases with different kind of information and clear representation (table like representation). A data frame can be seen as a list of lists. A row (one list) corresponds to an individual (for example the student John Doe in the previous section) while a column can be seen as a particular characteristic recorded in the database (for instance, the column of all student IDs).

```r
IDs = c(123, 124, 125) # creates a vector of IDs
names = c("John Doe", "Jane Doe", "Foo Bar") # Our students' names
majors = c("Math", "Biosciences", "Neurosciences") # What the students study

myDataFrame <- data.frame(IDs, names, majors, birthyears)
```

Exercise 3. Write the same data.frame as the example but using a concatenation of lists. Each list should be a single student.

Hint: See the commands `cbind` and `rbind` for this task.

In general data frames will not be created but loaded from an external file. This allows to read a complete database and store it in a single variable in a single line of code.

4 Working with R

4.1 Some useful functions

First of all, we describe as a function any operator of the R language which requires \( n \) parameters \((n \geq 0)\) and returns a certain value (shall it be of the type vector, data.frame or anything else). Note that R allows the user to pass the parameters in a rather random sequence as long as the name of the parameters are recalled. (Use the question mark for help `?aCertainFunctionIDontRememberTheArguments`) For instance, the help for the `matrix` function returns the following usage:

```r
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

All functions have the same format for their definitions: a name and a set of assignment as parameters separated by commas. The assignments give us both the names of the variables and their default values (these default values will be used if they are not passed as a parameter). As we just said above, the order of the parameter is not important as long as the parameters' names are recalled therefore

```r
# Assume x is a 150 dimensional vector already defined
myMatrix = matrix(x, 10, 15, FALSE, NULL)
myMatrix = matrix(x, nrow=10, ncol=15)
myMatrix = matrix(x, ncol=15, nrow=10)
```

are three equivalent ways to create a matrix of size \( 10 \times 15 \) using the vector \( x \) as original data.

Variable information are important whenever one wants to do a bit of programming. Here is some examples of functions used within R

- `dim()` returns a vector with the dimensionality of the variable \( x \). Note that this function can also be used as a left hand side of an assignment: `dim(x) <- someValue`.
- `nrow()` and `ncol()` return the number of rows and columns of a vector, array of data frame.
- `colnames()` and `row.names()` return the names of the different columns and rows (note the period in the second function...). They can also be used for assignment.
- `length()` returns an integer with the number of data points in the argument (for instance, 10 for a vector of size 10 or 16 for a \( 4 \times 4 \) matrix)

Exercise 4. 1. Write a sequence of statement to reproduce the `length()` function using the `dim()` function.

2. You are given a 15 dimensional vector \( z \). Convert it to a \( 3 \times 5 \) dimensional matrix `myMatrix` and assign names to its columns and rows.
Variables creation and access are particularly important. We have already seen how we can create variables using the matrix(), array(), c(), data.frames() or list() functions. Do not hesitate to ask for the inline help (?) for details about these functions.

Once a variable is created, it is important for you to be able to access a particular value at a given position (in the array, say). For this, you will (mainly) need the brackets [idx] (for array, matrices and vectors) or double-brackets [[idx]] (for lists and data.frames) notations.

If the data are multidimensional, the value at position p, q, r etc... can be accessed by separating each row and column index by commas. For instance, if m is a two dimensional matrix, the value at position (x,y) can be accessed with m[x,y]. Similarly, if one needs to process a complete column of a matrix (this does not work with data.frames), it can be accessed by omitting one of the argument, for instance m[,y] returns the complete column number y.

When using data.frames, calling a column can be done either by its number or by its name. In the case of the database of students as in the previous section, one can access the column of IDs by either myDB["IDs"] or using the column number myDB[[IDcolNb]].

Finally, a complete column can also be access using the $ sign. This sign works as a completion of variables, accessing sub parts of a complex structure (it is somewhat similar to the period for people working on structures in C programming). As an example, the IDs can be recovered from the database by writing myDB$IDs if the column’s name is "IDs". Hence the $ sign can be very useful in making the code more readable.

Variable overview functions are useful mainly for debugging a code. When one wants to work it is mandatory to check often if the operation does as expected. For this task one needs to know what is being processed and how the data are being handled/represented. The head() and tail() functions display the 10 first (resp. last) elements of a given table.

summary() str() are also interesting to get useful information about a table of data. However, they act very differently depending on the data passed as an input. It will basically try to summarize the data and get statistics out of them. The best way to understand these two functions is to try them.

Index generation can be very useful to select just a certain subset of the whole database. The function seq() creates a vector starting at a certain value, ending at another value, and all having a specified increment (or decrement in the case of a negative increment...)

someIdx = seq(from=1, to=10, by=2) # returns the vector (1,3,5,7,9)
otherIdx = seq(12, -10,-3) # otherIdx = (12, 9, 6, 3, 0, -3, -6, -9)

Set operations are often needed to compare biological populations. A population can be considered as containing or not certain species. Therefore, they can be represented by considering only the set of found individual in a sample. Comparing such sets can become very handy in such cases. (Note however, that it is not the smartest we can do). A certain number of built-in functions are available to the user without using any third party packages. setequal(A,B) returns a boolean TRUE if the two sets A and B contain exactly the same elements (i.e. A ⊂ B and B ⊂ A). Other set operators include intersect() setdiff() union() for the intersection, set difference (elements that are in a first given set but not in the second one) and union of sets.

Exercise 5. What is the value of the variables result1 and result2? Is this only due to a random smart guess?

firstSet = c(2,3,5,7) # vector of the first four prime integers
secondSet = c(2,4,6,8,10,12) # vector containing only multiples of 2
thirdSet = c(2,4,8,16,32,64) # vector containing only powers of 2
result1 = setequal( union(firstSet,secondSet), c(setdiff(firstSet,secondSet), ...
... intersect(firstSet,secondSet), setdiff(secondSet,firstSet)))
result2 = setequal( union(thirdSet,secondSet), c(setdiff(thirdSet,secondSet), ...
... intersect(thirdSet,secondSet), setdiff(secondSet,thirdSet)))
Exercise 6. The Jaccard index, often used as a similarity measure between two sets $A$ and $B$, is defined as

$$J(A, B) := \frac{|A \cap B|}{|A \cup B|}$$

Assume you are given two data matrices $X$ of size $mx \times nx$ and $Y$ of size $my \times ny$ you wish to compare. A row on such a matrix corresponds to an individual (for instance a DNA sequence, or a particular student) and a column to a characteristic (for instance the frequency of 4-mer AATG). A colleague gave you a method to compute a boolean vector $v_x$ out of your matrix $X$ (resp. $v_y$ out of $Y$) of size $nx$ (resp. $ny$) telling you whether the feature represented in a given row is relevant for the characterization of your sample or not.

Implement the Jaccard index between these boolean vectors, it can be used as a measure of similarity between the two original samples $X$ and $Y$. What are its characteristics (boundedness? particular cases of perfect match and perfect dismatch?)

Some functions are also available to check whether an element is part of a set or not. `a %in% aVector` returns true if the value of the variable `a` is contained in the vector (or set) `aVector`.

`is.element()` can also be used to find out whether an element is part of a larger set or not. Sometimes, the `match()` function is more useful as it returns the indices of all occurrences of a given value.

Working with vectors and arrays is very important to process code much faster. The `cbind()` and `rbind()` are mandatory whenever one wants to add information to a database (shall it be a new element or a new feature/characteristic). Moreover, the `sapply()` is particularly handy when dealing with large database as it avoids to implement loops. However this can be seen as a more advanced technique and should be learned later.

Printing `print()`

Graph generation `hist()` `boxplot()`

4.2 Working in a workspace

is unavoidable in R (and actually in any other programming language, whether interpreted or compiled). A workspace contains all the variables with their values used so far (since the startup of the interpreter or the last cleaning of the workspace). The `ls()` function lists all the variables in the workspace. It is really interesting when one tries to remember a certain variable name, say. Sometimes, variables are created just as temporary useful link to more complicated or advanced processing. These can be (and should be whenever dealing with large database) deleted from the workspace and hence free some memory from the system using the `rm()` command. You can specify a list of variables you wish to delete as an argument to that function.

Exercise 7. Using both functions `rm()` and `ls()` try to get rid of the whole R workspace (save before, if you need!)

The workspace contains all variables used so far. When you start dealing with data.frames, the code you are writting is getting more and more complicated due to the cumbersome notations to access certain rows or columns. To make everything more readable you can temporarily attach the substructure of a dataframe to the workspace using the `attach()` function. For instance if a dataframe `df` has the fields `IDs`, `ages` and `first.names` typing `attach(df)` in your interpreter will create ghost variables (ghost in the sense that they do not really exist in the system's memory, but are just names pointing to the underlying variables) `IDs`, `ages` and `first.names` which you can work with. This process can be inverted with the `detach(df)` command.

Whenever you start your interpreter, you are working in a Working Directory. You can customize this as you wish (and actually should customize it to avoid bad surprises). However if you are unsure where you are currently working in your harddrive the `getwd()` returns a string with the path to the current directory. This WD can be changed using the `setwd()` function.

Finally if you think you have been working really well, or just if you think your computer could crash anytime (keep this possibility in mind!) you might want to save you current workspace in order to keep track
of the processing you did so far. `save.image()` creates an image of your workspace while `savehistory()` creates a file containing all the commands you have passed in the interpreter.

A few words about R-Studio

4.3 Installing R

5 Defining functions and scripts

So far we have seen the functions needed to work with R. However, by practicing (if you haven’t practiced yet, you should now! Remember, learn the language by practicing your conversation) you (might) have noticed that it can get really cumbersome working directly in the interpreter. Whenever you start doing many times (where “many” means any number greater or equal to 2) the same task, you should think about saving the sequence of instructions in a file and calling this file as many times as you need.

5.1 Functions and scripts

A script is only a set of instructions you can call any time. It is somehow loaded sequentially whenever you call it. Calling a script does not create any environment or whatsoever. It just interprets the instructions saved in a separate files. When using scripts you need to make sure the variables’ names do not overlap with the one in your main workspace. The following set of instructions can be saved in a separate file (for instance called `myFirstRScript.R`)

```r
# This is an example of script
aScalar <- 5
aVector <- 1:5 # the nbOne:nbTwo notation is equivalent to seq(nbOne,nbTwo, 1)
finalVector = aScalar*aVector
areTheyEqual = setequal(finalVector, c(5,10,15,20,25))
```

This file contains only basic operations but can still be called in the interpreter by typing `source("myFirstScript.R")`. At the end, even an empty workspace should now contain four variables: the scalar `aScalar`, two numeric vectors `aVector` and `finalVector` and a boolean scalar `areTheyEqual` which should be `TRUE`, if everything went okay. The danger here is that one of the variables is already assigned in your workspace; in this case, it will be overwritten regardless of its previous value. To summarize, the use of a script via the `source()` function is equivalent to a copy paste of the lines in the separated file.

A way to avoid this effect is by using functions instead of script. A function is defined (in a separated file) by using the `function()` command. Save the following set of instructions in a separate file (for instance `separateFunction.R`) to learn how to write and call a function in R; it implements a basic addition.

```r
myFirstRFunction = function( aFirstNumericParameter=0, aSecondParameter=0 ) {
  result = aFirstNumericParameter+aSecondParameter # sums the two elements
  result
} # end of myFirstFunction
```

Now you need to make your interpreter aware of this function by "sourcing" it: `source("separateFunction.R")`. Now you can use this function without problem:

```r
a = 1
b = 2
c = myFirstFunction(a,b)
c == 3 # this should return TRUE
d = myFirstfunction(a) # the second parameter is omitted
d == a # this should return TRUE
```

Be aware that the use of that function is not in real time. It uses the function as it was when you sourced it. If you decide to modify the separate file, do not forget to save and "source" is once again.
5.2 Loops and conditions

If you need to add sequentially individuals to a database you can use the two main loops `for` and `while`. The difference between these two loops is that the first one can be used when the number of iterations is known in advance while the second one is used when the stopping criteria is evaluated `on-the-fly`.

```r
for(myCounter in someSet) { # myCounter is a running variable
    # someSet can either be generated with 1:10 or using some already defined vector
    some intructions
} # end of the for loop
```

In the case that `someSet=1:10`, the previous loop is equivalent to

```r
myCounter=1
while(myCounter <= 10){
    some instructions
    myCounter = myCounter+1 # do not forget to increment your loop in this case!
} # end of the while loop
```

Similarly, conditional statements can be done using the usual `if(condition){...}else{...}` set of commands.