An Introduction to R

Vittorio Perduca (Université de Paris)

September 2020

Contents

Chapter 1

Welcome

This document is a tutorial for self-learning the basic use of `R`. It was prepared for the students enrolled in the first year of the AIRE Life Sciences Master Program of the Centre de Recherche Interdisciplinaire at the Université de Paris.

This document was written in with the `R` package `bookdown` (Xie, 2020), (Xie, 2016) and is under continuous development, please report any issues to `vittorio.perduca at u-paris.fr`.

Chapter 2

Introduction

R is a statistical language developed from the 1990s at the University of Auckland, New Zealand. Its main implementation is the open source software (free and modifiable) R. R is normally used from the graphical user interface (GUI) and development interface RStudio.

R is an interpreted language (likePython) that we use from the command line:

```
print ('Hello world!')
```

```
## [1] "Hello world!"
```

Alternatively, you can run a script, that is a series of commands found in a file with the extension .R.

The R user community is very active in the scientific world (statistics, data science, bioinformatics, social sciences, ...) and more and more in companies. One of the advantages of R is the wealth of *packages* developed by users and developers that can be installed to increase its capabilities in many areas of statistics.

In addition, R has a very complete documentation. You can access help by typing ? followed by the *function* you are interested in:

```
? rnorm
```

You can also find a lot of information on the web: by searching on Google for a problem related to an R task, you almost always find an answer (often in threads opened on Cross Validated).

2.1 How to use this document

Although basic objects and commands are covered, this introduction is not a complete reference to the `R` language, so you will have to use the help and search for information on the web. Before moving on to the exercises at the end of each chapter, you are encouraged to type the commands and understand the result: the best way to learn a language is to write and debug lots of lines of code!

2.2 Useful references

A complete reference to `R` is the official introduction that can be found on the CRAN website: https://cran.r-project.org/doc/manuals/R-intro.html. For French readers, an excellent reference is Vincent Goulet's book *Introduction à la programmation R*, which can be downloaded at https://cran.r-project.org/doc/contrib/Goulet_introduction_programmation_R.pdf

2.3 Installing R and Rstudio

1. Download the `R` distribution that is appropriate for your machine from https://cran.r-project.org/ and install it by double-clicking on the installation file.
2. Dowanload and install RStudio Desktop from https://rstudio.com/products/rstudio/download/.

2.4 Getting started with RStudio

The RStudio workspace is generally divided into four pans: starting from the top left and going clockwise we find:

1. a script editor
2. the environment listing the variables in memory and the history of commands that were executed
3. a window for graphical outputs, the help, and the file and package managers.
4. the `R` console, ie the command line.

Typically we type the code in the editor and then run it in the console. To do this we place the cursor in the line we want to execute and we send the command to the console using the combination `cmd Return` in macOS and`ctrl Return` under Linux and Windows.

2.5 Packages

To install a package from the console:

```
install.packages("dplyr")
# download and install the dplyr package used for data manipulation
```

Once a package is installed, it will have to be loaded into memory each time a new session is opened:

```
library(dplyr) # quotes are not needed here
# require(dplyr) # equivalent function
```

Chapter 3

Language basics

3.1 Assignments, arithmetic operations

There are two types of R commands: expressions and assignments.

Expression

```
cos(pi)
```

```
## [1] -1
```

Assignments and expressions

```
x <- 1 + 2 # or x = 1 + 2
x
```

```
## [1] 3
```

```
y = 4
x == y
```

```
## [1] FALSE
```

Using ; we can type two commands on the same line before executing them:

```
e <- exp (1); log (e)
```

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

Some examples of arithmetic and Boolean operators:

```
3*4
```

```
## [1] 12
```

```
12/3
```

```
## [1] 4
```

```
2^3
```

```
## [1] 8
```

```
sqrt (16)
```

```
## [1] 4
```

```
1 == 2
```

```
## [1] FALSE
```

```
1!= 1
```

```
## [1] FALSE
```

```
FALSE & TRUE # and
```

```
## [1] FALSE
```

```
FALSE | TRUE # or
```

```
## [1] TRUE
```

3.2 Mode, length and class

In R, everything is an *object*. The *mode* specifies what an object can contain. The main modes are:

- `numeric`: real numbers
- `character`: character strings
- `logical`: logical values true / false
- `list`: list, collection of objects
- `function`: function

`numeric`,`character`, and `logical` objects are *simple* objects that can contain data of only one type. On the contrary, `list` mobjects are special objects that can contain other objects.

You can access the mode of an object with the `mode ()` function:

```
age = c(33,28, 33) # The concatenation function c() allows to create vectors
mode(age)
```

```
## [1] "numeric"
```

```
names <- c('Daniel', 'Jehanne', 'Romain')
mode(names)
```

```
## [1] "character"
```

```
my.list <- list(Names = names, Age = age)
mode(my.list)
```

```
## [1] "list"
```

```
mode(is.integer(pi))
```

```
## [1] "logical"
```

```
mode
```

```
## function (x)
## {
##     if (is.expression(x))
##         return("expression")
##     if (is.call(x))
##         return(switch(deparse(x[[1L]])[1L], `(` = "(", "call"))
##     if (is.name(x))
##         "name"
##     else switch(tx <- typeof(x), double = , integer = "numeric",
##         closure = , builtin = , special = "function", tx)
## }
## <bytecode: 0x7fb44d8eb5b0>
## <environment: namespace:base>
```

Besides the mode, an object also has a *length*, defined as the number of elements it contains:

```
length(age)
```

```
## [1] 3
```

```
length(names)
```

```
## [1] 3
```

```
length(my.list)
```

```
## [1] 2
```

The *class* of an object specifies its behavior and therefore its way of interacting with operations and functions. An important example are *data frames*: special lists whose elements all have the same length. The class of a data frame is different from that of generic lists and data frames have an indexing system that does not exist for other lists:

```
class(my.list)
```

```
## [1] "list"
```

```
my.data.frame = data.frame(names, age)
mode(my.data.frame)
```

```
## [1] "list"
```

```
class(my.data.frame)
```

```
## [1] "data.frame"
```

```
my.data.frame[1,2] # to extract the 1st element from the 2nd "column"
```

```
## [1] 33
```

```
# Try the following command:
# my.list[1,2]
```

A special object is the missing value `NA`. By default, its mode is `logical`, however`NA` is neither `TRUE` nor`FALSE`. To test if a value is missing we will use the `is.na ()` function:

```
NA == NA # not what we want
```

```
## [1] NA
```

```
is.na(NA)
```

```
## [1] TRUE
```

```
is.na(mean(c(1,4, NA)))
```

```
## [1] TRUE
```

Chapter 4

Vectors, matrices and functions

4.1 Vectors

The R basic object is the *vector* (a scalar is considered as a vector of length one). The most used function to create a vector is the concatenation:

```
price <- c(150, 162, 155, 157); price
```

```
## [1] 150 162 155 157
```

Indexing is done through brackets:

```
price[1] # Unlike in Python, the first index is always 1!!
```

```
## [1] 150
```

```
price[c(1.3)]
```

```
## [1] 150
```

```
price[-(1:2)] # to extract all elements except the 1st and 2nd
```

```
## [1] 155 157
```

One can also use a boolean indexing vector, the extracted elements are obviously those corresponding to the `TRUE` values. For example to extract prices greater than 156:

```
price > 156 # the boolean vector
```

```
## [1] FALSE  TRUE FALSE  TRUE
```

```
price[price > 156]
```

```
## [1] 162 157
```

An alternative is given by the `which ()` function which returns the indices whose elements satisfy a logical condition:

```
which(price > 155)
```

```
## [1] 2 4
```

```
price[which(price > 156)]
```

```
## [1] 162 157
```

You can use the indexing to change an element:

```
price [1] <- 0; price
```

```
## [1]   0 162 155 157
```

It is possible to give labels to the elements of a vector and extract elements based on them:

```
names(price)
```

```
## NULL
```

```
# NULL is a special object with NULL mode that reads "no container"
names(price) <- c('model.1', 'model.2', 'model.3', 'model.4')
price
```

```
## model.1 model.2 model.3 model.4
##       0     162     155     157
```

```
price['model.3']
```

```
## model.3
##     155
```

In a vector, all the elements must have the same mode:

```
x <- c(1,2, 'a', 'b'); x
```

```
## [1] "1" "2" "a" "b"
```

```
mode(x)
```

```
## [1] "character"
```

To generate the vector of the first n integers we use the syntax `1:n`

```
1:10
```

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

```
2:6
```

```
## [1] 2 3 4 5 6
```

To generate more general sequences we use the `seq()` function:

```
seq(from = 2, to = 20, by = 2) # or more simply seq(2,20,2)
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

We can create a vector of repeated elements with `rep()`:

```
rep(1, len = 3) # same thing as rep (1,3)
```

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

```
rep(NA, 4)
```

```
## [1] NA NA NA NA
```

4.2 Matrices

A matrix is a vector with a `dim` attribute of length two. All the elements of a matrix therefore have the same mode. To create a matrix:

```
M <- matrix(2:7, nrow = 2, ncol = 3); M
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    3    5    7
```

```
matrix (2:7, nrow = 2, ncol = 3, byrow = TRUE)
```

```
##      [,1] [,2] [,3]
## [1,]    2    3    4
## [2,]    5    6    7
```

By default `matrix ()` fills the new matrix one column after another. Indexing is done through brackets:

```
M[2,] # 2nd line
```

```
## [1] 3 5 7
```

```
M[, 3] # 3rd column
```

```
## [1] 6 7
```

```
M[2.3]
```

```
## [1] 3
```

```
M[3]
```

```
## [1] 4
```

```
M[, -2] # to extract all columns except the 2nd
```

```
##      [,1] [,2]
## [1,]    2    6
## [2,]    3    7
```

To vertically (resp. horizontally) merge two matrices we use `rbind()` (resp. `cbind()`):

```
cbind (M, -M)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    2    4    6   -2   -4   -6
## [2,]    3    5    7   -3   -5   -7
```

```
rbind(M, 2 * M)
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    3    5    7
## [3,]    4    8   12
## [4,]    6   10   14
```

4.3 Operations on numerical vectors and matrices

Element wise operations:

```
v <- c(3,4,1,6)
v + 2
```

```
## [1] 5 6 3 8
```

```
v * 2
```

```
## [1]  6  8  2 12
```

```
v * v
```

```
## [1]  9 16  1 36
```

```
v / 2
```

```
## [1] 1.5 2.0 0.5 3.0
```

```
v / v
```

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

```
v + v^2
```

```
## [1] 12 20  2 42
```

```
sqrt(M)
```

```
##          [,1]     [,2]     [,3]
## [1,] 1.414214 2.000000 2.449490
## [2,] 1.732051 2.236068 2.645751
```

```
M * M
```

```
##      [,1] [,2] [,3]
## [1,]    4   16   36
## [2,]    9   25   49
```

```
# Try the following command:
# M + v
```

Transpose, multiplication, inverse:

```
t(M)
```

```
##      [,1] [,2]
## [1,]    2    3
## [2,]    4    5
## [3,]    6    7
```

```
N <- M[, -3]
N %*% diag(1,2) # row by column product matrix
```

```
##      [,1] [,2]
## [1,]    2    4
## [2,]    3    5
```

```
# diag (1,2) builds the 2x2 diagonal matrix where all the
# diagonal elements are equal to 1, ie the 2x2 identity matrix
solve(N)
```

```
##      [,1] [,2]
## [1,] -2.5    2
## [2,]  1.5   -1
```

```
solve(N) %*% N # checking if solve(N) is the inverse of N
```

```
##      [,1]          [,2]
## [1,]    1 1.776357e-15
## [2,]    0 1.000000e+00
```

The transpose of a vector is a row matrix:

```
V <- t(v)
dim(V)
```

```
## [1] 1 4
```

```
t(V)
```

```
##      [,1]
## [1,]    3
## [2,]    4
## [3,]    1
## [4,]    6
```

Pay attention to the following examples:

```
v %*% t(v) # v is considered a column vector!
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    9   12    3   18
## [2,]   12   16    4   24
## [3,]    3    4    1    6
## [4,]   18   24    6   36
```

```
t(v) %*% v # ditto
```

```
##      [,1]
## [1,]   62
```

```
diag(1,4) %*% v # ditto
```

```
##      [,1]
## [1,]    3
## [2,]    4
## [3,]    1
## [4,]    6
```

```
v %*% v # v is both considered a row-vector and a column-vector
```

```
##      [,1]
## [1,]   62
```

4.4 Factors

A factor is a vector used to represent qualitative variables, ie a variable with discrete values. Its values, or categories, are called the `levels` in `R`.

```
city <- c('paris', 'lyon', 'lyon', 'paris', 'nantes')
fact.city <- as.factor(city); fact.city
```

```
## [1] paris  lyon   lyon   paris  nantes
## Levels: lyon nantes paris
```

```
class(fact.city)
```

```
## [1] "factor"
```

```
levels(fact.city)
```

```
## [1] "lyon"   "nantes" "paris"
```

A factor has the `numeric` mode. The reason for this counter-intuitive fact is that the elements of a factor are represented as integers corresponding to the lexicographic order of their values:

```
mode(fact.city)
```

```
## [1] "numeric"
```

```
as.numeric(fact.city)
```

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

4.5 User-defined functions

Example:

```
my.function <- function(x, y = 10) {# the default value of y is 10
  z = x-y
  return(z)
}
my.function(2)
```

```
## [1] -8
```

```
my.function(2,4)
```

```
## [1] -2
```

```
my.function(y = 1, x = 4)
```

```
## [1] 3
```

Any variable defined in a function is *local* and does not appear in the workspace: try to run

```
z
```

Chapter 5

Lists and data frames

5.1 Lists

Lists are special vectors that can store elements of any mode (including other lists).

```
age = c(33,28, 33)
names <- c('Daniel', 'Jehanne', 'Romain')
my.list <- list(Names = names, Age = age)
```

Like any other vector, a list is indexed by the `[...]` operator, however, note that the result will be a list containing as unique element the desired item:

```
my.list[1]
```

```
## $Names
## [1] "Daniel"  "Jehanne" "Romain"
```

```
mode(my.list[1])
```

```
## [1] "list"
```

To get the desired item directly, we therefore use the `[[...]]` operator or the `$` operator followed by the name of the element (if available):

```
my.list[[1]]
```

```
## [1] "Daniel"  "Jehanne" "Romain"
```

```
my.list$age
```

```
## NULL
```

The elements of a list can have different lengths:

```
city <- c('paris', 'lyon', 'lyon', 'paris', 'nantes')
my.list$city <- city
my.list
```

```
## $Names
## [1] "Daniel"  "Jehanne" "Romain"
##
## $Age
## [1] 33 28 33
##
## $city
## [1] "paris"  "lyon"   "lyon"   "paris"  "nantes"
```

5.2 Data frames

The most widely data containers is the data frame, a special list of class `data.frame` in which all elements have the same length. For this reason, a data frame is represented in the form of a two-dimensional array whose columns are its elements. Typically, in a data frame the columns represent the **variables** and the rows the **observations**. Unlike matrices, the elements of a data frame can have different modes.

```
id <- c('id.453', 'id.452', 'id.455', 'id.459', 'id.458', 'id.456', 'id.450', 'id. 451
age <- c (19, 45, 67, 53, 17, 30, 27, 35)
smoker <- c (TRUE, FALSE, TRUE, TRUE, FALSE, TRUE, TRUE, TRUE)
sex <- c ('f', 'f', 'h', 'h', 'f', 'h', 'f', 'f')
my.db <- data.frame (Id = id, Age = age, Smoker = smoker, Sex = sex); my.db
```

```
##        Id Age Smoker Sex
## 1  id.453  19   TRUE   f
## 2  id.452  45  FALSE   f
## 3  id.455  67   TRUE   h
## 4  id.459  53   TRUE   h
## 5  id.458  17  FALSE   f
```

```
## 6  id.456  30   TRUE   h
## 7  id.450  27   TRUE   f
## 8 id. 451  35   TRUE   f
```

```
dim(my.db); nrow(my.db); ncol(my.db)
```

```
## [1] 8 4

## [1] 8

## [1] 4
```

```
names(my.db)
```

```
## [1] "Id"     "Age"    "Smoker" "Sex"
```

A data frame being a list, we can extract a column using the $ operator preceded by the name of the data frame and followed by the name of the column (or variable), or use the operator [...]

```
my.db$Sex # a column of characters is automatically transformed into a factor
```

```
## [1] f f h h f h f f
## Levels: f h
```

```
my.db[, 2]
```

```
## [1] 19 45 67 53 17 30 27 35
```

```
my.db$Age[my.db$Smoker == FALSE] # simple example of selection
```

```
## [1] 45 17
```

The columns are directly accessible in the workspace (without having to type the name of the data frame and the $) after having *attached* the data frame:

```
attach(my.db)
Age
```

```
## [1] 19 45 67 53 17 30 27 35
```

To display only the first six lines:

```
head(my.db)
```

```
##       Id Age Smoker Sex
## 1 id.453  19   TRUE   f
## 2 id.452  45  FALSE   f
## 3 id.455  67   TRUE   h
## 4 id.459  53   TRUE   h
## 5 id.458  17  FALSE   f
## 6 id.456  30   TRUE   h
```

Similarly, `tail()` creates a data frame with the last six columns.

5.3 Importing and exporting data

Importing data is a fundamental step in data analysis. To load the data stored in a file (texte, .csv, Excel, ...) into the workspace (ie into memory), you can use the basic function `read.table()`. The three most important arguments are:

- `file`: name (and path) of the file, in quotes
- `header`: are the elements of the first row the names of the columns?
- `sep`: character separating the columns

`read.table()` returns a data frame:

```
url1 <- 'https://raw.githubusercontent.com/vittorioperduca/Introduction-to-R/master/da
d1 <- read.table(url1,
                 # the first line contains the name of the variables
                 header = TRUE,
                 # values are separated by ;
                 sep = ';')

head (d1)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
url2 <- 'https://raw.githubusercontent.com/vittorioperduca/Introduction-to-R/master/data/heart.tx
d2 <- read.table(url2,
                 header = TRUE,
                 # variables are separated by a tabulation
                 sep = '\t')
dim(d2); names(d2)
```

```
## [1] 270  13
```

```
##  [1] "age"          "sexe"         "type_douleur" "pression"     "cholester"
##  [6] "sucre"        "electro"      "taux_max"     "angine"       "depression"
## [11] "pic"          "vaisseau"     "coeur"
```

For data stored in the `.Rda` or `.Rdata` format, the import is done with `load()` with the argument `file = filename`. For instance download the `Iris.Rda` file at github.com/vittorioperduca/Introduction-to-R/blob/master/data/Iris.Rda to your working directory and then try the following:

```
iris_path <- 'data/Iris.Rda' # replace with the file path
load(iris_path)
```

If you want to load `.Rda` or `.Rdata` files directly from an url, don't forget to use the `url()` function (this was not necessary in `read.table()`).

Data can be exported either to a text file (or .csv, Excel ...) using `write.file()` or to `.rda` and `.Rdata` files at using `save()`. In both cases, the two main arguments are

- `x` = data to save
- `file` = the name of the file (in quotes).

If the dataset is stored (or must be saved up) locally, it is necessary to know (and be able to modify) the working directory:

```
# getwd() # try on your machine!
# setwd('~/Documents') # to move to the Documents directory
```

Remember that in Linux and macOS machines, `~/` is a shortcut for `/Users/username`. For Windows machines, the address syntax is slightly different. For example we use `\` instead of `/`.

5.4 Exercice

Download the text file raw.githubusercontent.com/vittorioperduca/Introduction-to-R/master/data/hepatitis.txt to your working directory.

1. Import the dataset in `R`. Warning: missing data were coded with a `?`, read the documentation of `read.table ()`.
2. Find the number of observations, display the names of the variables and the first six observations. Check that the value of `STEROID` for the fourth observation is missing using the appropriate function.
3. Calculate the mean value of `ALBUMIN` in women and men.
4. Create a variable `NSYMP` counting the number of times a variable is equal to`yes` between `FATIGUE` and`MALAISE`. Pay attention to the format of these two variables!

Bibliography

Xie, Y. (2016). *bookdown: Authoring Books and Technical Documents with R Markdown*. Chapman and Hall/CRC, Boca Raton, Florida. ISBN 978-1138700109.

Xie, Y. (2020). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.20.