Introduction to R

Version 1

Charalambos Themistocleous

2016

Contents

1 An informal introduction to R **5**

1.1 Download R and RStudio 5

1.2 Getting help in R 6

1.3 R packages 7

1.4 The R environment 8

2 Expressions **9**

2.1 Variables 9

2.2 Constant 11

2.3 Numbers 13

2.4 Operators 15

2.5 Functions 17

3 Objects **22**

3.1 Vectors 22

3.1.1 Select values from vectors 24

3.1.2 Arithmetic operations with vectors 26

3.1.3 Character vectors 30

3.1.4 Vectors with logical values 31

3.2 Matrices 38

3.2.1 Selecting Values in matrices 41

3.2.2 Mathematical operations with matrices 43

3.3 Arrays 48

3.4 Lists 51

3.5 Data frames 54

3.5.1 Creating dataframes 54

3.5.2 Presentation of the data frame 58

3.5.3 Select values in dataframes 60

3.5.4 Changing values in data frames 67

3.5.5 Change of column names 69

4 Missing Values **71**

5 Functions attach () and detach () **74**

6 Importing data in R **75**

6.1 Import data from the R input window 75

6.2 Import data from text files 76

6.3 Save files 79

7 Workflow in R 80

1 An informal introduction to R

This chapter aims to provide a quick introduction to R, its syntax, and vocabulary. If you are familiar with R, you may skip it, if not you may want to read it as it will help you understand the code we will be using, before we jump into our main topics related to Natural Language Processing, Probability Theory, and Machine Learning.

R is a programming language and a statistical analysis software. It originates from the S programming language developed by Rick Becker, John Chambers, and Allan Wilks at Bell Laboratories (Chambers, 2008). It is a domain-specific language focused on statistics, mathematics, and machine learning. According to Chambers (2002b, p. 4), the creators of R aimed at the development of a programming language that provides reliable calculations of data enabling researchers to get the most from the analysis. R allows the processing of data and their representation in plots and tables. R has be extented with numerous packages, which facilated its adoption in the academic community. Nevertheless, as a programming language, R can be used to create any type of application. Also, one can utilize R to programmatically interact with the operating system, such as to create, delete and rename files or folders.

1.1 Download R and RStudio

R is available for free on the R website: http://www.r-project.org/. After, downloading R, we need to download RStudio, which is an integrated development environment (IDE) that facilates coding. RStudio can be downloaded from http://www.rstudio.com.

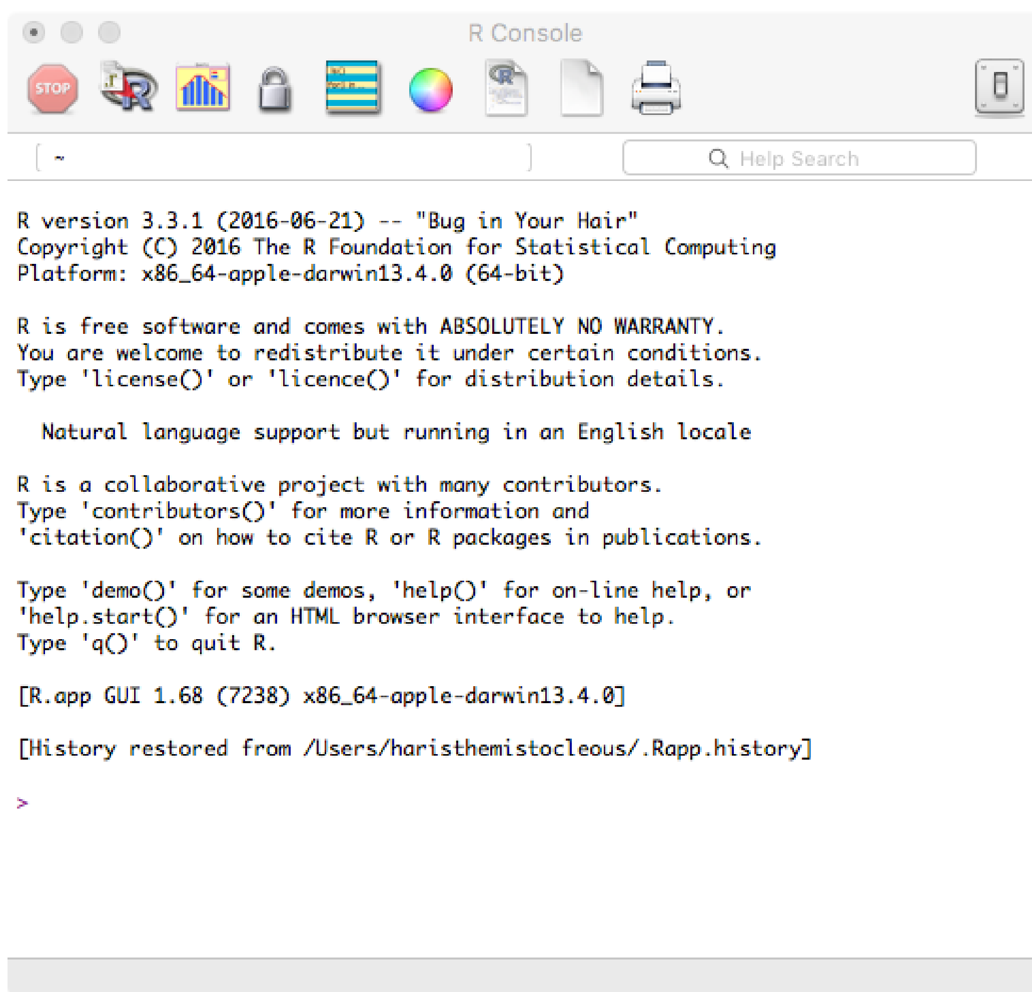


Figure 1: R IDE on MacOS Sierra.

1.2 Getting help in R

The basic help function in R is the question mark (?) followed by the name of a function e.g., ?plot; this will open the help window in your computer with the relevant information. Table 1 summarizes some of the most common help functions.

Table 1: Help functions in R.

Help function	Interpretation
?function or help(function)	e.g., ?plot or help(plot), it provides help for the plot() function.
apropos(term)	it provides all the functions that contain a term.
example(plot)	provides examples, if available, for a certain function.

1.3 R packages

R consists of the main system—the base—and its packages. The base includes:

- functions for data management and storage,
- a number of operators for performing calculations,
- basic data analysis tools,
- chart creation functions,
- all components that are necessary in a programming language, such as statements, methods, control flow, etc .

The base can be expanded with several packages, which enable the user to access code created by other users. The number of R packages is really impressive. At the time of writing this document there were 9915 available packages. The R editor and the RStudio have options on their menus to automatically install packages. Packages can be installed by using the install.packages() function. For example, the following code, install the ggplot2 package in your system:

R code: 1.3.0.

```
install.packages ("ggplot2")
```

Several packages have been grouped into categories, a.k.a., Task Views:

Task Views	
Bayesian	Bayesian Inference
DifferentialEquations	Differential Equations
Distributions	Probability Distributions
ExperimentalDesign	Design of Experiments (DoE) & Analysis of Experimental Data
MachineLearning	Machine Learning & Statistical Learning
MetaAnalysis	Meta-Analysis
Multivariate	Multivariate Statistics
NaturalLanguageProcessing	Natural Language Processing
ReproducibleResearch	Reproducible Research
SocialSciences	Statistics for the Social Sciences
TimeSeries	Time Series Analysis
WebTechnologies	Web Technologies and Services
…	

Table 2: Task views in R

1.4 The R environment

In R every interaction with the environment is performed using commands, which can be written directly in the terminal, in Rstudio, or in our favorite text editor, such as Emacs. To access R from the terminal simply type R. To exit R, type q().

However, instead of the terminal, it is much more convinient to use RStudio for writing and executing R code. It is also a good idea to take some time and familiarize yourself with Rstudio. To start writing code, select from the File menu in Rstudio New > R Script.

2 Expressions

An expression in R is a combination of

1. values
2. variables
3. constants
4. numbers
5. operators
6. functions

Specific reference is made hereinafter.

2.1 Variables

The variable is a symbolic name, which can take different values. The variable can store a value, which can numeric, alphanumeric, logic, etc. The symbols x and y in 2.1.1 are variables. The number to the right is the value assigned to the variable.

Example 2.1.1

(i) $x = 30$

(ii) $y = 45$

Note that R can assign values to a variable with the symbol = like most other programming languages but the symbol <- is more common and it is the one that we are using here. Note that the symbol -> to reverse yield, examples are provided in 2.1.2.

Example 2.1.2

a = 20 + sqrt (9) # returns the value of the transaction 20 + sqrt (9) in the variable a.

b <- 20 + sqrt (9) # returns the value of the transaction 20 + sqrt (9) in the variable b.

20 + sqrt (9) -> c # attributes the value of the transaction 20 + sqrt (9) in the variable c.

Also, the informal style manual for R provided by Google (Chambers, 2008, p. 16) suggests the use of the symbol (<-) for associating values to a variable. For example, 2.1.3 generates a vector from a 1 to 20.

Example 2.1.3

a <- 1:20 # generates a vector from 1 to 20

To call a variable use its name. So when asked a variable environment R given the result (4).

R code: 2.1.0.

```
a
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

We can assign in a variable, data consisting of characters, logical values, dates, etc.

R code: 2.1.1.

```
name <- "Maria"
```

2.2 Constant

a constant is a value that cannot be altered by the program during normal execution. For example, the constant *letters* includes the letter of the English alphabet.

R code: 2.2.0.

```
letters # produces the series of letters of the English
   alphabet.
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
[2] "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

The assignment of a constant to another variable is as follows:

R code: 2.2.1.

```
latin.letters <- letters # performance series of letters
   to the variable a,
                 producing the same result as the previous
                    one.
latin.letters
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
[2] "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

We can use variables to perform mathematical operations.

R code: 2.2.2.

```
a <- 3
b <- 4
c <- a + b
c
[1] 7
```

Caution! You may use any name for a variable, yet, when one prepares an extensive statistical analysis, it is good to provide names to variables that make sense, so as to avoid ambiguity. For instance names, such as *a, b, c* that make no sense is better to be avoided and prefer more clear names:

R code: 2.2.3.

```
length <- 30
width <- 40
area  <- length * width
```

Note! If you are familiar with another programming language such as C, C++, C# or Java, you might have noticed that the type of the variable, like *int* for integers, *string* for characters, etc., is not required in R.

2.3 Numbers

R employs standard notation to declare numbers and operations. For scientific notation uses the letter e to specify a range in the power of four.

R code: 2.3.0.

```
1/10000
[1] 1e-04
```

To read the scientific notation 1e-04 simply add 4 zeros before 1, i.e., 0.0001. To convert the scientific notation to standard, you can use the function format as follows: format(1e-04, scientific=FALSE). The imaginary numbers are defined using i as a suffix.

R code: 2.3.1.

```
12i * 3
[1] 0 + 36i
```

Also, the symbol for infinity is *inf*:

R code: 2.3.2.

```
1/0
[1] Inf
-1 / 0
[1] -Inf
```

In some cases, certain components of a vector may be unknown as in the case of missing values. This can happen in an experiment or in questionnaire some participants did not provide an answer. In R, the missing values are indicated by the symbol NA (not available). Generally, R converts any act of missing values in TO. The is.na function (x) gives a logical vector, ie. A vector which contains two values, the True (TRUE value) and the False value (FALSE) equal to the size of x. Where is the vector TO attributed the True value.

R code: 2.3.3.

```
x <- c (1: 3, NA)
nas <- is.na (x)
```

There is also another kind of values derived by numerical calculation. These values are called Not a Number and are denoted by NaN as in the following cases:

R code: 2.3.4.

```
0/0
[1] NaN
Inf - Inf
[1] NaN
```

2.4 Operators

Operators are expressions that are used in arithmetic and logic. The Table 5.2 presents the key operators in R.

You may use R as calculator as in the following:

R code: 2.4.0.

operator	Description
$+$	addition
$-$	subtraction
$*$	multiplication
$/$	division
or $**$	power
xx() parentheses	

Table 3: Basic arithmetic operators in R.

```
1 + 5
```

R code: 2.4.1.

```
[1] 6
```

The result 6 follows a number in square brackets [1]; the number in the brackets defines the position of the data in the output to assist with the reading of the data. For example, number [11] in the second raw of the output in **??** says that the numbering continues from the eleventh position.

R code: 2.4.2.

```
series <- 1:15 # create vector with the numbers 1-15.
# series of vector slope
```

```
> series
[1] 1 2 3 4 5 6 7 8 9 10
[11] 11 12 13 14 15
```

Other simple actions you can perform in R are the following:

R code: 2.4.3.

```
2-3 # removal
[1] -1
2 * 3 # proliferation
[1] 6
2/3 division #
[1] 0.6666667
2 ^ 3 power #
[1] 8
(3 * 3) ^ 0.5 # square root
[1] 3
```

In the last example, the use of brackets allows control of the order of operations. Otherwise we would have other answers.

2.5 Functions

R provides a large number of mathematical functions that allow us to carry out more complex mathematical operations:

R code: 2.5.0.

```
sqrt (3 * 3) # square root
[1] 3
```

Caution! Functions are case-sensitive in R, so we need to pay attention and distinguish between capital and small letters: the sqrt(), Sqrt(), and SQRT() are considered by R as different functions. This distinction applies to variable names as well. So the variables *aVariable* and *avariable* are different variables. Table **??** shows the most important of arithmetic operations including the R shown in the following table.

In R, a function can take as an input other functions. For example, one can take the square root of the absolute value of the vector sum of -30, -20, 34, 45, -21 using a single expression which encapsulates different operations:

R code: 2.5.1.

```
sqrt (abs (sum (c (-30, -20,34,45, -21))))
```

A function in R is defined as follows:

R code: 2.5.2.

Mode	Description
abs(x)	absolute value
sqrt (x)	square root
ceiling(x)	strong ylosi upwards: ceiling (3.475) results: 4
floor(x)	floor stand (3.475) results: 3
trunc(x)	standard decimal trunc (5.99) results: 5
round (x , digits = n)	round (3.475, digits = 2) results: 3.48
signif (x , digits = n)	signif (3.475, digits = 2) result: 3.5
cos (x), sin- (x), tan (x)	sine, cosine, tangent. Also acos (x), cosh (x), acosh (x), etc.
log (x, y = base)	natural logarithm
log10 (x)	common logarithm
exp (x)	exp (3)
eˆx	in force 3 4 stated as 3 ˆ 4

Table 4: Mathematical functions in R.

```
myfunction <- function (arg1, arg2, ...) {
  expressions
  return (object)
}
```

For instance, if we want to write a function that calculates the area of a rectangular, we simply write the following (see **??**):

R code: 2.5.3.

```
area <- function (length, width) {
e = length * width
return (paste ("The␣area␣is" e))
}
```

We can apply our function as follows:

R code: 2.5.4.

```
area (5, 6)
"The␣area␣is␣30"
```

The first number corresponds to the length and the second width of the rectangular. To provide these values in reverse order, we must explicitly state the values as follows:

R code: 2.5.5.

```
area(width = 6, Length = 5)
"To␣area␣is␣30"
```

We can use the same function with multiple arguments as well.

R code: 2.5.6.

```
a <- c (20, 30, 50, 23)
b <- c (10, 14, 19, 9)

area (a, b)
```

We get the following response:

[1] ”The area is 200” ”The area is 420” ”The area is 950” ”The area is 207” In this case the R computes for each pair of values in the vectors a and b the size 20 * 10 30 * 14 etc.

When using functions, the code is more efficient, the programming is faster and easier, and our code becomes more readable.

3 Objects

Beyond the basic expressions, we considered in the preceding section, R has a number of objects, namely vectors, matrices, arrays, lists, and dataframes. These objects enable the creation, the processing, and the storage of items. We can also use them to perform mathematical operations with the use of objects of R.

3.1 Vectors

Vectors are quantities that are characterized by one dimension. This means that a vector is simply a series of data, such as a row or a column in a data table. The following rows are vectors:

1. 1, 2, 5.3, 6, -2, 4
2. "Dog", "cat", "mouse", "fish"
3. TRUE, TRUE, TRUE, FALSE, TRUE, FALSE

These vectors can be stored in the R variables, so we can define three variables: mynumbers, mypets, and timesiamcorrect to store the vectors.

R code: 3.1.0.

```
mynumbers <-  c (1,  2, 5.3,  6, -2,  4) #  numeric vector.
```

```
mypets <-  c ( "dog", "cat",  "mouse", "fish␣of␣glass")  #
   vector of characters.
timesiamcorrect   <- c  (TRUE, TRUE, TRUE,  FALSE, TRUE,
   FALSE) # logical vector.
```

Each constituent is written within the function c()—c stands for *concatenate*—separated by commas. Note, that items included in the vector *mypets* are within quotation marks “” because the items are strings; without the quotations R will read them as variable names.

If you write the name of the variable, then R presents the contents of the variable:

R code: 3.1.1.

```
mynumbers
[1] 1.0 2.0 5.3 6.0 -2.0 4.0

mypets
[1] "dog" "cat" "mouse" "smelt␣of␣glass"
```

R code: 3.1.2.

```
timesiamcorrect
[1] TRUE TRUE TRUE FALSE TRUE FALSE
```

1	2	3	4	5	6	Serial number
1.0	2.0	5.3	6.0	-2.0	4.0	value

3.1.1 Select values from vectors

To refer to the items inside a vector, we use subvectors. Take for example the vector *mynumbers* that we defined above. R assigns to each item in this vector a serial number, which serves as the address of the item in the memory, as shown in following:

To take the 2nd number, we type the name of the variable followed by the position of the value in square brackets: mynumbers[2].

To extract the values at the 2nd and 5th position of the vector *mynumbers*, we write the positions in rectangular brackets [] as vectors, that is with the c() function (see. **??**):

R code: 3.1.3.

```
mynumbers[c (2,5)] # the 2nd and 5th price vector mynumbers
[1] 2 -2
```

(A) To select the items in the 2nd, 3rd and 4th position we can write [2:4], : means 2 to 4. The following expressions are equivalent.

R code: 3.1.4.

```
vector.a [2: 4]
[1] 2.0 5.3 6.0
```

or

R code: 3.1.5.

```
vector.a [c (2,3,4)]
[1] 2.0 5.3 6.0
```

(B) to select all other values except those which are in 3rd and 5th position of the vector *mynumbers*, we add the minus symbol before the number.

R code: 3.1.6.

```
mynumbers[c(-3, -5)]
[1] 1 2 6 4
```

In the same way, we can select items from character vectors. To, select all the items except the "dog" and "mouse" from vector *mypets*, we type the following.

R code: 3.1.7.

```
> mypets[c (-1, -3)]
[1] "cat" "fish"
```

(C) To select items that satisfy a condition and exclude items that do not satisfy a condition, we can employ logical operators (greater (>), smaller (<), equal to (==), not equal (! =) etc.). For example, to select all numbers in *mynumbers* who are greater than five, we will have to provide the following command mynumbers[mynumbers > 5]:

R code: 3.1.8.

```
mynumbers[mynumbers > 5]
[1] 5.3 6.0
```

Caution! The square brackets perform a different function from parentheses. Parentheses are used in R for determining the order of operations, and in functions e.g., c(1, 3, 4). The square brackets are utilized for accessing items in vectors and other objects.

3.1.2 Arithmetic operations with vectors

In R, we employ vectors to perform calculations. Let us take for instance the following vectors:

R code: 3.1.9.

```
vector.a <- c (1,2,3,4,5)
```

```
vector.b <- c (6,7,8,9,10)
vector.c <- vector.a + vector.b # Add vectors
```

vector.c contains the sum of the first value of the *vector.a* and *vector.b*, the second value of vector.a and vector.b, etc., so, the output of *vector.c* is the following:

R code: 3.1.10.

```
[1] 7 9 11 13 15
```

If the vectors have a different number of components, R will perform the addition but it will throw the following error message "In vector.a + vector.b: longer object length is not a multiple of shorter object length". This means that result might not be the one we want. To understand the output, let us change *vector.b* by adding two more numbers at the end of the vector, as in **??**.

R code: 3.1.11.

```
vector.b = c (6, 7, 8, 9, 10, 11, 12)
```

and try to add $vector.a + vector.b$, the result will be the following:

R code: 3.1.12.

```
vector.a <- c (1, 2, 3, 4, 5)
vector.b <- c (6, 7, 8, 9, 10, 11, 12)
vector.c <- vector.a + vector.b
Warning message:
In vector.a + vector.b :
  longer object length is not a multiple of shorter object
     length
> vector.c
[1]  7  9 11 13 15 12 14
```

Note that in the output 7 9 11 13 15 12 14, the numbers 11 and 12 were added to the first two values of vector.a (i.e., 1 and 2). So, we need to pay attention to this kind of messages. Overall, R performs all possible operations with numeric vectors (such as subtraction, multiplication, and division).

R code: 3.1.13.

```
vector.a <- c (1,2,3,4,5)
vector.b <- c (6,7,8,9,10)
vector.d <- a - b # Remove
```

The result obtained is shown in refsubtractionresults.

R code: 3.1.14.

```
vector.d
[1] -5 -5 -5 -5 -5
```

Multiplication of vectors is performed as follows:

R code: 3.1.15.

```
vector.e = vector.a * vector.b # Multiplication
```

The result of multiplication is shown in **??**.

R code: 3.1.16.

```
vector.e
[1] 6 14 24 36 50
```

Finally, the division result shown in **??**.

R code: 3.1.17.

```
vector.f = vector.a / vector.b # Division
vector.f
[1] 0.1666667 0.2857143 0.3750000 0.4444444 0.5000000
```

Dividing by zero (0) provides the following results:

R code: 3.1.18.

```
vector.a <- c (0, 1, 2, 3, 4)
vector.b <- c (6, 0, 8, 9, 10)
vector.f = vector.a / vector.b
vector.f

[1] 0.0000000 Inf 0.2500000 0.3333333 0.4000000
```

3.1.3 Character vectors

Operations can be performed with vectors containing characters, as in the following:

R code: 3.1.19.

```
vector.a <- c ( "one", "two", "three")
vector.b <- c ( "one", "two", "three")
vector.c <- c (vector.a, vector.b)
```

This creates a new vector comprised of the values of vector.a vector.b, as shown below:

R code: 3.1.20.

```
vector.c
"one" "two" "three" "one" "two" "three"
```

Note that in this case the two vectors are brought together; in other words they have been concatenated.

3.1.4 Vectors with logical values

To perform logical operations, e.g., see **??**, we employ logical operators, see the Table 5.

R code: 3.1.21.

```
logic.a <- c (TRUE, FALSE, TRUE)
logic.b <- c (TRUE, FALSE, FALSE)
logic.c <- logic.a + logic.b
logic.c
[1] 0 2 1
```

R code: 3.1.22.

```
vector.a <- c (1:10) # Generates a vector with numbers
   from 1 to 10.
```

operator	Description
$<$	smaller than
$<=$	smaller or less than
$>$	bigger than
$>=$	greater or equal than
$==$	equal to
$! =$	other than
$!x$	not x
$x\|y$	x or y
$x\&y$	x and y
isTRUE (x)	verify that the X is TRUE

Table 5: Logical operators in R.

```
vector.a
[1]  1  2  3  4  5  6  7  8  9  10
```

R code: 3.1.23.

```
vector.a < 4 #  this shows which values are less than 4.
[1] TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE
   FALSE
```

To show values correspond to TRUE, you can use the output in the following way:

R code: 3.1.24.

```
vector.a [c (TRUE, TRUE, TRUE, FALSE, FALSE, FALSE, FALSE,
   FALSE, FALSE, FALSE)]
```

or

R code: 3.1.25.

```
vector.a [c (T, T, T, F, F, F, F, F, F, F)]
[1] 1 2 3
```

R code: 3.1.26.

```
vector.a [(vector.a> 7) | (vector.a <4)] # This selects
   values greater than 7 and less than 4.
[1] 1 2 3 8 9 10
```

Let us show how to display the differences between two vectors containing letters of the English alphabet. To achieve this let us first create a vector that includes letters “e” to “o” and compare it with another vector that contains random letters. This is done in the following way.

R code: 3.1.27.

```
englishletters <- letters[5:15]  # This will select  the
    letters of
the English alphabet in positions 5-15  and add them to
    the variable
englishletters
```

Note, that we can also create vectors by manually typing them but this is much easier. The output is the following:

R code: 3.1.28.

```
englishletters
[1] "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
```

To select random letters from the constant *letters* we will use the function sample().

R code: 3.1.29.

```
randomenglishletters <-  sample(letters [5:15])  # letters
   of the  English alphabet
from position 5-15 in random order
```

The output of randomenglishletters is [1] ”k” ”h” ”g” ”j” ”e” ”o” ”n” ”m” ”f” ”l” ”i”. Now, to compare the two vectors, namely *randomenglishletters* and *englishletters* we use the logical operator (==) in the following way:

TRUE	FALSE
10	1

R code: 3.1.30.

```
randomenglishletters == englishletters
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
   TRUE FALSE
```

TRUE indicates the positions where the letters are identical and FALSE indicates the positions that they differ. The table() function shows the sum of all the TRUE and FALSE correspondences.

table(englishletters == randomenglishletters)

The sum() can be employed to show how many letters are identical

R code: 3.1.31.

```
sum(englishletters == randomenglishletters)
[1] 1
```

To view how many letters differ you may change the operator (==) to (!=) as sum(englishletters != randomenglishletters). The result in **??** displays the number of identical letters in the two vectors. To see which letters are identical we have to express it in the following way:

R code: 3.1.32.

```
 englishletters[ englishletters  == randomenglishletters]
"g"
```

What this expression says is the following, report from vector *englishletters*, the letter that are identical with those in vector *randmenglishletters*. Note that this expression is particularly important and will be used many times when we try to find data in other R objects as well, such as tables and dataframes. The which() function can be employed to show the location of the letter ”g”. This is expressed as follows:

R code: 3.1.33.

```
which(englishletters  == "g")
[1]  3
```

To declare a vector of characters as a factor, for instance, as an independent variable in an experiment, we use the function factor(). That is, to create the independent variable *gender* with two levels: *girls* and *boys* consisting of 25 girls and 25 boys we first create a character vector the rep() function in rep(”girls”, 25) and rep(”boys”, 35) simply repeats the strings “girls” and “boys” 25 times. The function factor() converts the character vector to a factor vector.

R code: 3.1.34.

```
gender <- c (rep ( "girls", 25), rep ( "boys", 25))
fgender <- factor (gender)
```

Factors R are ordered sets of items and there is a specific relationship between these items. For instance, fgender does not simply have 50 items but there is a certain relationship in these items, namely they are grouped into two levels.

R code: 3.1.35.

```
levels (fgender)
[1] "Boys" "Girls"
```

The summary () provides a report of how many ingredients exist in each level of the factor *fgender*:

R code: 3.1.36.

```
summary (fgender)
boys girls
25 25
```

Beyond numbers and characters, R can also represent dates. The as.Date () function converts a character vector into a date vector:

R code: 3.1.37.

```
date <- as.Date (c ( "2011-05-30", "2012-06-07"))
```

To number the days between 2011-05-30 and 2012-06-07 you can simply subtract them:

R code: 3.1.38.

```
days <- date[2] - date[1]
```

The result is the following:

R code: 3.1.39.

```
days
Time difference of 374 days
```

3.2 Matrices

Matrices are basic objects, they represent a collection of data, which are organized in rows and columns, as Table A below (see 1).

$$A = \begin{bmatrix} 43 & 44 & 33 \\ 34 & 33 & 34 \end{bmatrix} \tag{1}$$

The columns in a table must be in the same category (characters, numbers, etc.) and have the same length. The general form for the creation of a matrix is the following:

R code: 3.2.0.

```
mymatrix <- matrix (vector , nrow = r, ncol = c, byrow =
   FALSE ,
dimnames = list(row.names, column.names))
```

1. byrow = TRUE: this option indicates that the table must be filled with rows whereas the standard option for this is byrow = FALSE, which indicates that the matrix must be full of columns.
2. the dimnames is used to provide names for the columns and the rows.

There are many ways to create a table. One can start with a vector and organize the vector into columns and rows as in (43).

R code: 3.2.1.

```
vector.a <- 1:20 # create a vector with 20 numbers.
# Create a numerical table 4 x 5
matrix.a <-matrix (vector.a, nrow = 4, ncol = 5)
The result is the following:
```

R code: 3.2.2.

```
matrix.a
```

```
[, 1] [2] [3] [4] [5]
[1] 1 5 9 13 17
[2] 2 6 10 14 18
[3] 3 7 11 15 19
[4] 4 8 12 16 20
```

The creation of a table with only two rows can be formulated as follows:

R code: 3.2.3.

```
matrix.b <-matrix (vector.a, nrow = 2)
```

The number of rows of the table are determined by the operator *nrow*; the operator *ncol* can define the number of columns. The result of the matrix.b is the following:

R code: 3.2.4.

```
matrix.b
```

```
[, 1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
[1] 1 3 5 7 9 11 13 15 17 19
[2] 2 4 6 8 10 12 14 16 18 20
```

In matrices, the operator *dimnames* can be employed to provide names for columns and rows. This is accomplished in the following way:

R code: 3.2.5.

```
# Explicit definition
Cells <- c (35,46,23,67,31,33,25,55)
row.names <- c ( "Row1" "Row2" "Row3" "Row4")
column.names <- c ( "Height", "Series")
heights <- matrix (cells, nrow = 4, ncol = 2, byrow =
   TRUE, dimnames = list (row.names, column.names))
```

The result obtained is displayed in the following table:

R code: 3.2.6.

```
heights
```

3.2.1 Selecting Values in matrices

To select items from a matrix we employ square brackets in a way which very similar to the one we used to select items in vectors. They only difference is that

	Height	Series
Row1	35	46
Row2	23	67
Row3	31	33
Row4	25	55

we have to define two dimensions this time not one. Specifically, to select an item from the second row of the fifth column we define that in square brackets as [2,5]. The general form that we define the selection is the following.

R code: 3.2.7.

```
table [row, column]
```

That is first we provide the number of row(s) and then the number of column(s). Some examples are provided below.

R code: 3.2.8.

```
matrix.a [, 2] # all rows of the second column of  matrix.a
matrix.a [4,] # 4th row of the table matrix.a
matrix.a [1: 3,1: 3] # rows 1-3 of columns 1-3.
matrix.a [c (2,3), 1: 3] #  rows 2 and 3 of columns 1 to 3.
matrix.a [1: 3,] # rows 1 to 3 of all columns in matrix.a
matrix.a [-1: -3,] # take all rows and columns exempt from
   1 to 3.
```

The absence of a number before or after the comma indicates the selection of all items in the position, namely all columns or all rows, consequently [,] selects the whole table!

The minus sign ($-$) selects all data other than those specified in the minuses.

3.2.2 Mathematical operations with matrices

Performing mathematical operations in R, such as matrix addition, matrix multiplication, etc., is extremely easy. In **??**, two tables were created with 20 random numbers each consisting of four rows and five columns.

R code: 3.2.9.

```
random.numbers.a <- runif (20, 1,10)
random.numbers.v <- runif (20, 1,10)
matrix.a <-matrix (random.numbers.a, nrow = 4, ncol = 5)
matrix.v <-matrix (random.numbers.v, nrow = 4, ncol = 5)
```

The runif function (20, 1, 10) 20 generates random numbers from a uniform distribution on interval between 1 and 10.

R code: 3.2.10.

```
matrix.a
```

```
[, 1] [2] [3] [4] [5]
[1] 2.620871 3.729884 4.221497 8.687757 6.738848
[2] 9.664402 8.828212 1.163462 5.865550 5.827878
[3] 9.642593 4.101409 3.427884 1.068980 6.605213
[4] 7.396233 5.121680 8.193945 7.627636 6.460386
```

R code: 3.2.11.

```
matrix.v
[, 1] [2] [3] [4] [5]
[1] 6.847156 2.088490 9.285601 6.876831 4.970435
[2] 9.420954 3.830850 3.578612 3.118894 3.574255
[3] 2.121085 6.380238 6.322597 7.422863 9.066266
[4] 3.373487 7.329575 5.864051 1.524884 2.604839
```

The addition of these two tables can be made as follows:

R code: 3.2.12.

```
> matrix.a + matrix.v
[, 1] [2] [3] [4] [5]
[1] 9.468027 5.818373 13.507099 15.564588 11.709284
[2] 19.085356 12.659062 4.742074 8.984444 9.402132
[3] 11.763678 10.481647 9.750481 8.491842 15.671479
[4] 10.769720 12.451255 14.057996 9.152520 9.065225
```

Multiplication and division table matrix.a with Table matrix.v carried out as follows:

R code: 3.2.13.

```
matrix.a * matrix.v
[, 1] [2] [3] [4] [5]
[1] 17.94551 7.789824 39.19914 59.744236 33.49501
[2] 91.04788 33.819557 4.16358 18.294028 20.83032
[3] 20.45276 26.167965 21.67313 59.88462 7.934889
[4] 24.95110 37.539739 48.04971 11.631258 16.82827
```

R code: 3.2.14.

```
> matrix.division <- matrix.a / matrix.v
> round(matrix.division, 2)
[, 1] [2] [3] [4] [5]
[1] 0.38 1.79 0.45 1.26 1.36
[2] 1.03 2.30 0.33 1.88 1.63
[3] 4.55 0.64 0.54 0.14 0.73
[4] 2.19 0.70 1.40 5.00 2.48
```

The round(matrix.division, 2) function rounds the values in two decimal places.

R includes basic functions that enable us to permit quick calculations with matrices. For example, to average values in all columns of a matrix, we can use

the function colMeans(x), to calculate the sum of all rows, we can employ the rowSums() function.

R code: 3.2.15.

```
colMeans (matrix.a) # get the mean values of  columns.
[1] 7.331025 5.445296 4.251697 5.812481 6.408081
```

R code: 3.2.16.

```
colSums (matrix.a) # get the sum values of columns.
[1] 29.32410 21.78118 17.00679 23.24992 25.63233
```

The Table **??** common mathematical operations in R objects.

Finally, to transpose a matrix, we employ the function t(), which returns the matrix transposed. The following shows an example of transposing a matrix in R.

R code: 3.2.17.

```
matrix.a
[, 1] [2] [3] [4] [5]
[1] 2.620871 3.729884 4.221497 8.687757 6.738848
[2] 9.664402 8.828212 1.163462 5.865550 5.827878
```

Function	Description
colMeans (x)	It provides the mean values of columns in matrices, tables, and dataframes.
colSums (x)	It provides the sums of the columns in matrices, tables, and dataframes.
cor (x, y)	It provides the correlation between two vectors, the x and y.
max (x)	It provides the maximum value of a vector x.
mean (x)	It provides the mean of a vector x.
median (x)	It provides the median of a vector x.
min (x)	It provides the minimum value of a vector x.
order (x)	It returns another vector arithmetic that contains the vector data x in ascending order.
pmax (x, y, z)	It receives one or more vectors (or matrices) as arguments and returns a single vector providing the 'parallel' maxima of the vectors.
pmin (x, y, z)	It receives one or more vectors (or matrices) as arguments and returns a single vector providing the 'parallel' minima of the vectors. .
quantile (x)	It provides the quantiles of a vector or matrix
range (x)	It provides the minimum and the maximum value of a vector or matrix.
rowMeans (x)	It provides the means of rows in a dataframe or matrix.
rowSums (x)	It provides the sums of the rows of a dataframe or matrix.
sort(x)	It sorts a vector or factor into ascending or descending order.
sum (x)	It returns the sum of a vector.
var(x) and cov	provide the variance and covariance of one or two vectors.

Table 6: Mathematical operators.

```
[3] 9.642593 4.101409 3.427884 1.068980 6.605213
[4] 7.396233 5.121680 8.193945 7.627636 6.460386
```

R code: 3.2.18.

```
t(matrix.a)
[, 1] [2] [3] [4]
[1] 2.620871 9.664402 9.642593 7.396233
[2] 3.729884 8.828212 4.101409 5.121680
[3] 4.221497 1.163462 3.427884 8.193945
[4] 8.687757 5.865550 1.068980 7.627636
[5] 6.738848 5.827878 6.605213 6.460386
```

3.3 Arrays

array ()

An array is a vector that is stored with some additional properties, such as dimensions (specified using the operator dim") and names for these dimensions (specified using the operator "dimnames"). A sequence is identical to a table, but may have more dimensions. In operator "dim" is defined as a numeric vector containing non-negative values, the effect of these prices should correspond to the length of the series.

R code: 3.3.0.

```
array.a <- array (1:48, dim = c (3,4,2))
array.a

, , 1

[, 1] [2] [3] [4]
[1] 1 7 13 19
[2] 2 8 14 20
[3] 3 9 15 21
[4] 4 10 16 22
[5] 5 11 17 23
[6] 6 12 18 24

, , 2
```

```
[, 1] [2] [3] [4]
[1] 25 31 37 43
[2] 26 32 38 44
[3] 27 33 39 45
[4] 28 34 40 46
[5] 29 35 41 47
[6] 30 36 42 48
```

An alternative way of creating an array is by modifying a vector.

R code: 3.3.1.

```
vector. a <- 1:48 # create a vector.
```

R code: 3.3.2.

```
dim (vector.a) <- c ( 6,4,2) # create two rows with three
   numbers rows and four columns.
```

Now array.a and vector.a are the same, which can be tested by using the function identical() in the following:

R code: 3.3.3.

```
identical(array.a, vector.a)
TRUE
```

To select values in array, we employ square brackets, like in vectors and matrices. However, as arrays can be multidimensional, we have to define three values in the square bracket, which correspond to the following: 1. row, 2. column, and 3. matrix.

R code: 3.3.4.

```
array.a [1 ,,]
array.a [1 ,,]
[, 1] [2]
[1] 1 25
[2] 31 7
[3] 13 37
[4] 19 43
```

The definition of the operator ”dimnames” is optional and it is employed to set the names in rows. An example is provided in the following:

R code: 3.3.5.

```
dimnames (array.a) <- list (c ( "A", "B", "C"), c ( "S1",
   "S2", "S3", "S4"), c ( "Group␣A" "Group␣B"))
```

```
array.a
,, Group A

S1 S2 S3 S4
A 1 4 7 10
B 2 5 8 11
C 3 6 9 12

,, Group B

S1 S2 S3 S4
A 13 16 19 22
B 14 17 20 23
C 15 18 21 24
```

3.4 Lists

list () Lists are collections of objects or components. The list allows the assembly of components of different categories, vectors, tables, sets, lists, etc., which are not necessarily connected.

R code: 3.4.0.

```
my.vector <- 1:10
list.a <- list (
number.a = 1,
```

```
character.title = "corpora␣for␣analysis"
vector.a = my.vector,
array.a = array (1:48, dim = c (3,4,2)),
matrix.a = matrix (my.vector, nrow = 4, ncol = 5)
)
list.a

$ number.a
[1] 1

$ character.title
[1] "corpora␣for␣analysis"

$ vector.a
[1] 1 2 3 4 5 6 7 8 9 10

$ array.a
,, 1

[, 1] [2] [3] [4]
[1] 1 4 7 10
[2] 2 5 8 11
[3] 3 6 9 12

,, 2

[, 1] [2] [3] [4]
[1] 13 16 19 22
[2] 14 17 20 23
```

```
[3] 15 18 21 24


$ matrix.a
[, 1] [2] [3] [4] [5]
[1] 1 5 9 3 7
[2] 2 6 10 4 8
[3] 3 7 1 5 9
[4] 8 2 4 6 10
$
```

The following three functions provide information about the items in the list

- names (x) function returns the names of the components of the list.
- length (x) function returns the number of items in the list.
- unlist (x) functions converts the list into single vectors.

To select items from lists, we employ square brackets of this form [[]]. In the square brackets we place a numeric index or the name of the object. This is shown in **??**.

R code: 3.4.1.

```
list.a [[3]] # The third component of the list.
[1] 1 2 3 4 5 6 7 8 9 10
```

```
list.a [[ "character.title"]] # Refer to character.title
   component.
[1] "corpora␣for␣analysis"
```

Also, the $ symbol followed by the name of the object can employed to access items in a list.

R code: 3.4.2.

```
list.a $ character.title
[1] "corpora␣for␣analysis"
$
```

3.5 Data frames

Dataframes are tables which can accept values of different types, such as numbers, characters, logical values (TRUE or FALSE), dates etc. Unlike tables, dataframes do not require columns to have the same length. Dataframes are very similar in their properties with worksheets, such as those in Microsoft Excel or Libreoffice Calc, and therefore they are very suitable to store data, which are imported from these programs.

3.5.1 Creating dataframes

Table 7 is a dataframe with data of the participants in an experimental design.

	Height	Series
AA	Sex	Age
1	man	22
2	man	22
3	man	24
4	man	25
5	man	23
6	Woman	22
7	woman	23
8	woman	23
9	woman	22
10	woman	23

Table 7: Data Frame participants information on experimental design.

We can create a dataframe manually by inserting the values in R or as it is often the case we can import a table into R for analysis. The latter is what happens, when we get results from an experiment or data from other applications. First, we will see how to create manually a dataframe in R manually and then how we import data into dataframes from other sources. To create a dataframe we employ the function data.frame(), as shown below.

R code: 3.5.0.

```
# Vector with values for the serial number.
vector.id <- 1:10
# Vector with values for the sex of participants: five men
```

```
    and five women.
factor.gender <- gl (2, 5, labels = c ( "man", "woman"))
# Vector with the ages of the participants:
vector.age <- c (22, 22, 24, 25, 23, 22, 23, 23, 22, 23)
# Construction data frame:
dataframe.data <- data.frame (
ID = vector.id,
Sex = factor.gender,
Age = vector.age
)
```

The dataframe has been created by a vector containing the serial numbers of items, a factor containing the gender with two levels “man” and “woman”, and a vector containing the age of the participants. Note that when you prepare a table in R or in Excel / Calc is good to add a column with the serial numbers of item. The serial number works as an element for data identification and can help us to order the table to its original state if the layout has been changed. This column can have the name id and the data can be a series of number from 1 to the length of the table.

To create the factor for gender we employed the function gl(), which has the following structure:

R code: 3.5.1.

```
gl (n, k, length = n * k, labels = 1: n, ordered = FALSE)
```

Specifically, the parameters of gl() are explicitly defined as follows:

R code: 3.5.2.

```
gl (n = 2, k = 5, labels = c ( "male", "female"))
 [1] male   male   male   male   male   female female
    female female female
Levels: male female
```

In this case n = 2, k = 5 the like. The n is a number of levels of the factor; k is the of items assigned to each level. Therefore, the operator length has a predetermined value n * k. Next, we provide the labels of the factor sex: c(”man”, ”woman”).

Finally, the three vectors vector.id, factor.gender, and vector.age are used to generate a dataframe titled *dataframe.data*. After we create the dataframe, we can remove the objects we employed to construct the dataframe from memory, by deleting them as follows:

R code: 3.5.3.

```
rm (vector.id)
rm (factor.gender)
rm (vector.age)

or in one line: rm(vector.id, factor.gender, vector.age)
```

3.5.2 Presentation of the data frame

To see the contents of a dataframe simply write the name of the dataframe in the console:

R code: 3.5.4.

```
dataframe.data
```

Alternatively, the View() function can be used to present the data in a tabular form:

R code: 3.5.5.

```
View(dataframe.data)
```

The dim() function indicates the dimensions of the data frame.

R code: 3.5.6.

```
dim (dataframe.data)
```

The names() function shows the variables are within dataframe.data data.

R code: 3.5.7.

```
names (dataframe.data)
The result is the following:
[1] "ID" "Sex" "Age"
```

The str() function shows the structure of an object. For dataframe.data data frame the result of the operation is as follows:

R code: 3.5.8.

```
str (dataframe.data)
'Data.frame': 10 obs. of 3 variables:
$ ID: int 1 2 3 4 5 6 7 8 9 10
$ Sex: Factor w / 2 levels "man", "woman": 1 1 1 1 1 2 2 2
   2 2
$ Age: num 22 22 24 25 23 22 23 23 22 23
```

Specifically, the str() function reports the that *dataframe.data* is a dataframe with 10 rows and 3 columns. For each column, it provides the class of the data, e.g., integer, number, and factor. In general, to ask R about the class of an object, we employ the class() function.

R code: 3.5.9.

```
class (dataframe.data)
[1] "data.frame"
```

Function	Description
length (object)	It returns the number of elements in an object.
str (object)	It displays the structure of an object.
class (object)	It returns the class or type of the object.
names (object)	It provides names for an object.
c(object, object, ...)	It forms a vector of objects.
cbind (object, object, ...)	It combines objects by columns
rbind (object, object, ...)	It combines objects bu rows.
objectname	It returns the data of the object.
ls ()	It lists the items of memory.
rm(object)	It removes items from memory.
new <- edit(object)	It creates a copy of an object with a new name for editing.
fix(object)	It opens an interactive editing window in R with the data of the object insic

Table 8: Useful functions for data analysis.

The Table **??** presents useful functions of R for data analysis.

3.5.3 Select values in dataframes

Selecting values in dataframes is very similar to the way we select items in other objects, such as vectors and matrices. We define the position of the object in square brackets, by providing an index of the rows in the left part and in the right part an index of the columns, separated by a comma:

[rows,columns]

1, 1	1, 2	1, 3	1, 4
2, 1	2, 2	2, 3	2, 4
3, 1	3 2	3, 3	3, 4
4, 1	4, 2	4, 3	4, 4

Table **??** provides indexes of each cell in a 4×4 table.

To select the value of the cell in the address [2, 2] in the dataframe, we provide the name of the dataframe and the position in square brackets:

R code: 3.5.10.

```
dataframe.data [2, 2]
```

The R output is the following:

R code: 3.5.11.

```
[1] man
Levels: man woman
```

Therefore, to select rows from 1 to 2, we type: dataframe.data [1: 2,]. Since, we do not define a column index after the comma, R we select all columns. Consequently, the selection dataframe.data[, 1: 2], will return all rows in columns from 1 to 2, since in this case only the column index is specified. Finally, dataframe.data[c (1,3,4),2] returns rows 1, 3, and 4 of column 2.

The minus sign (-) before the index of the row or column is used to exclude values. So to select all data in a table except the data in the third row, we type the following:

R code: 3.5.12.

```
dataframe.data [-3,] # option all data except row 3.
```

Another way to refer to columns of a dataframe is by using the name of the column. So, first we type the name of the dataframe and after that the dollar sign ($), and then the name of the column, as in **??**.

R code: 3.5.13.

```
dataframe.data$Sex
dataframe.data$Age
```

The function levels () shows the levels of a factor.

R code: 3.5.14.

```
levels (dataframe.data$Sex)
```

Of course, we can assign the column to a variable.

R code: 3.5.15.

```
a <- dataframe.data$Sex
b <- dataframe.data$Age
```

To select values from a dataframe based on conditions, we employ logical operators. For example, the selection of all participants whose age is 23 in the dataframe, we use the (==) operator:

R code: 3.5.16.

```
dataframe.data [dataframe.data $ Age == 23,]

ID Sex Age
5 of 5 man 23
7 7 23 woman
8 8 woman 23
Woman 10 10 23
```

Similarly, to select all "male" participants, we provide the following command.

R code: 3.5.17.

```
dataframe.data [dataframe.data $ Sex == "man" ,]
```

Note that because in this case the value is not number but a character, we need to type the value in double or single quotes). Other logical operators that are commonly employed for data selection are the operators less than (<) and greater than (>), as in the example **??** (see Table 5 for more information on logical operators).

R code: 3.5.18.

```
over23 <- dataframe.data [dataframe.data $ age <23,]
```

In this case, the new table is assigned in the variable over23.

R code: 3.5.19.

```
new.data

ID Sex Age
1 1 man 22
2 2 man 22
6 6 22 woman
9 9 woman 22
```

More complicated data selections are possible by combining conditions. For instance, to select participants whose age is less than or equal to 25 or higher than 22 we can use the logical OR operator (|) to connect these two conditions as in [dataframe.data $ age >= 25 | dataframe.data $ age == 22,]:

R code: 3.5.20.

```
dataframe.data [dataframe.data $ age >= 25 |
    dataframe.data $ age == 22,]
ID Sex Age
1 1 man 22
2 2 man 22
4 4 man 25
6 6 22 woman
9 9 woman 22
```

Another useful logical operation for combining conditions is the Logical AND operator, which combines each element of the first vector with the corresponding element of the second vector as in **??**. This selection returns all female participants whose age is less than 24.

R code: 3.5.21.

```
dataframe.data [dataframe.data $ Age <24 & dataframe.data
   $ Sex == "woman", ]
ID Sex Age
6 6 22 woman
7 7 23 woman
8 8 woman 23
9 9 woman 22
Woman 10 10 23
```

Logical operators can be employed along with the subset () function, which provides additional options, for selecting data. For instance, subset(dataframe.data, age $<$ 24 & sex == ”woman”, select = aa: age) selects the data of participants from columns AA to Age, whose Age is less than 24 and their Sex is “woman”.

R code: 3.5.22.

```
ID Sex Age
6 6 22 woman
7 7 23 woman
8 8 woman 23
9 9 woman 22
10 10 woman 23
```

The choice of three random sequences from the table obtained by the sample () function (see. (91)).

R code: 3.5.23.

```
sample.table <-
   dataframe.data[sample(1:nrow(dataframe.data),
   3,replace=F),]
sample.table
```

```
ID Sex Age
```

```
3 3 man 24
4 4 man 25
6 6 22 woman
```

The head () function shows the first five rows of a dataframe and the tail() function displays the last five rows of a dataframe.

R code: 3.5.24.

```
head (data)
This function is equivalent to the following option
      93
Data [1: 5]
```

R code: 3.5.25.

```
tail (mydata)
```

Notice, that there is some flexibility in how to structure the commands in R, as the same results can be achieved in many ways.

3.5.4 Changing values in data frames

We provided a relatively lengthy description on data selection because it is the foundation for many other operations. In this section, we discuss the way we

change values in a dataframe. Let us begin with the following example. We found that there was a mistake in the records and all the participants whose age is equal or less than 22 are in fact 19 years old. So, to change all these values we can either type them one by one or we can use a condition. The latter is of course more efficient in terms of effort and time, especially when the data are big. To change the values, we need to select the values that we want to change and then using the assignment operator to assign a new value as in the following example:

R code: 3.5.26.

```
dataframe.data $ Age [dataframe.data $ Age <= 22] <- 19
```

The result of the change is the following:

R code: 3.5.27.

```
dataframe.data
ID Sex Age
1 1 man 19
2 2 man 19
3 3 man 24
4 4 man 25
5 of 5 man 23
6 6 19 woman
7 7 23 woman
```

```
8 8 woman 23
9 9 woman 19
Woman 10 10 23
```

Often when we make changes we want to store them in a new table instead of changing the original dataframe. This can be achieved very easily by making a copy of the dataframe in the following way.

R code: 3.5.28.

```
changed.data <- dataframe.data
changed.data$Age[changed.data$Age <= 22] <- 19
```

3.5.5 Change of column names

The function names () is used when need to change the column-names of the dataframe; names() returns the names of the columns as in the following example:

R code: 3.5.29.

```
names(dataframe.data)
[1] "ID" "Sex" "Age"
```

To change of the names of columns, we have to assign the new values in the following way:

R code: 3.5.30.

```
names (dataframe.data) [2: 3] <- c ("Gender", "Age")
```

```
\subsubsection{Sorting data}
One can sort the rows or columns  of a data frame in
   various ways. The
following  example sorts  the \emph{dataframe.data}  based
   on  Age and
stores the results in a new dataframe.
```

```
\begin{rcode}{R Code}{rlabel}
order.d <- dataframe.data [order (dataframe.data$Age)]
order.d
AA Sex Age
1 1 man 22
2 2 man 22
6 6 22 woman
9 9 woman 22
5 of 5 man 23
7 7 23 woman
8 8 woman 23
Woman 10 10 23
```

```
3 3 man 24
4 4 man 25
```

The minus symbol can be used to sort Age in decreasing order, alternatively we can define this [order(dataframe.data$Age, decreasing= TRUE),].

R code: 3.5.31.

```
revorder.d <- dataframe.data [order (-dataframe.data$Age),]
revorder.d
AA Sex Age
4 4 man 25
3 3 man 24
5 of 5 man 23
7 7 23 woman
8 8 woman 23
Woman 10 10 23
1 1 man 22
2 2 man 22
6 6 22 woman
9 9 woman 22
```

4 Missing Values

In R, missing values are indicated by the symbol NA. The undefined or unrepresentable values, especially in floating-point calculations are represented by the

symbol NaN (not a number). To check for missing values, we employ the is.na () function. Its outcome is a data frame in which the missing values are identified as TRUE.

If we change the missing values 2, 3, and 9 to NA and use the is.na() function the output will be the following.

R code: 4.0.0.

```
  dataframe.data
  $ Age [c (2,3,9)]  <- NA # Change values of  rows
  2, 3, and 9 in column Age and check for NA values.
 is.na (dataframe.data)

ID Sex Age
[1,] FALSE FALSE FALSE
[2,] FALSE FALSE TRUE
[3,] FALSE FALSE TRUE
[4,] FALSE FALSE FALSE
[5,] FALSE FALSE FALSE
[6,] FALSE FALSE FALSE
[7,] FALSE FALSE FALSE
[8,] FALSE FALSE FALSE
[9,] FALSE FALSE TRUE
[10,] FALSE FALSE FALSE
```

Another useful function for detecting NA values the complete.cases() function.

R code: 4.0.1.

```
dataframe.data [complete.cases (dataframe.data),]
ID Sex Age
1 1 man 22
4 4 man 25
5 of 5 man 23
6 6 22 woman
7 7 23 woman
8 8 woman 23
Woman 10 10 23
```

The same effect can be obtained with the help of na.omit function().

R code: 4.0.2.

```
na.omit (dataframe.data)
Instead, the selection of data with missing values is the
   following:

dataframe.data [! complete.cases (dataframe.data),]
ID Sex Age
2 2 man NA
3 3 man NA
9 9 NA woman
```

5 Functions attach () and detach ()

The attach () converts column names into variables, so that it is possible to refer to them explicitly using the name of the dataframe:

R code: 5.0.0.

```
attach (dataframe.data)
```

In this way, it is possible to refer to the statement of column names as follows:

R code: 5.0.1.

```
Sex
Age
```

However, the use of the attach function () can cause problems when there are other objects with the same name in R memory. To avoid the risk of ambiguity, it is better to explicitly set the column name using the symbol $ (dataframe.data$Age) The opposite of the attach function () is the detach () function.

R code: 5.0.2.

```
detach (data)
```

In this section we introduced the R objects. We have seen how to create vectors, tables, sets, lists, and dataframes. Also, we have seen how to choose values of these objects, delete or change values. In the next section we will see how to import data in R.

6 Importing data in R

Instead of typing the data into variables and creating the dataframe using these elements, we usually import data from other applications such as Excel or Calc. There are three options.

1. Type data directly in an R input window.
2. Import data from external files of text.
3. Import data Excel or Calc tables.

6.1 Import data from the R input window

One way to input data is by using the R data entry window, which activated using the edit() function. Specifically, the edit() function displays a window similar to spreadsheet (see Figure 2).

R code: 6.1.0.

```
data <- edit (as.data.frame (NULL))
```

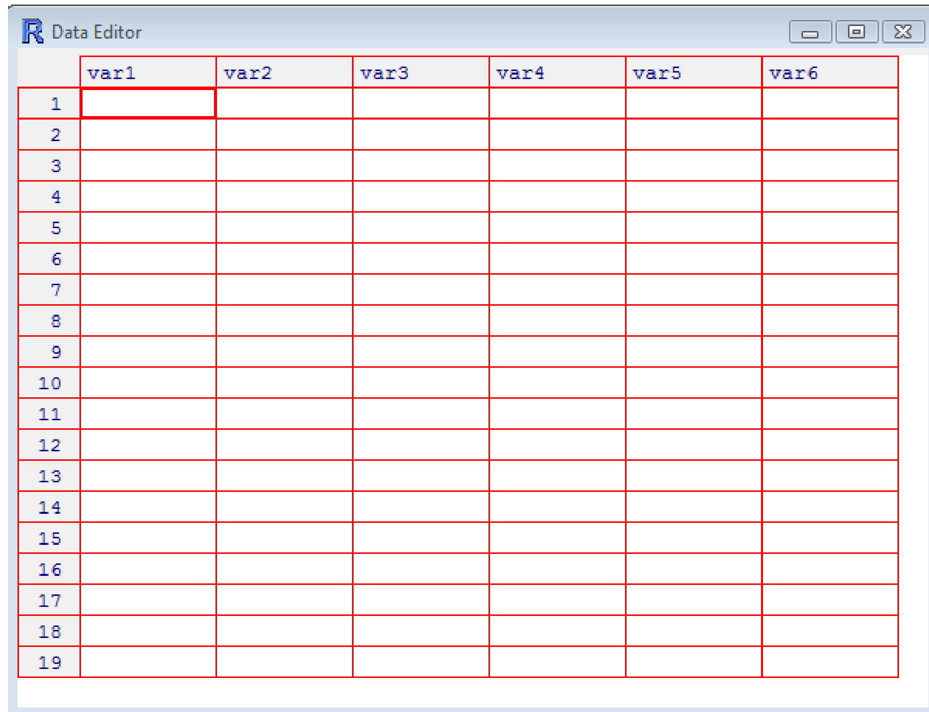


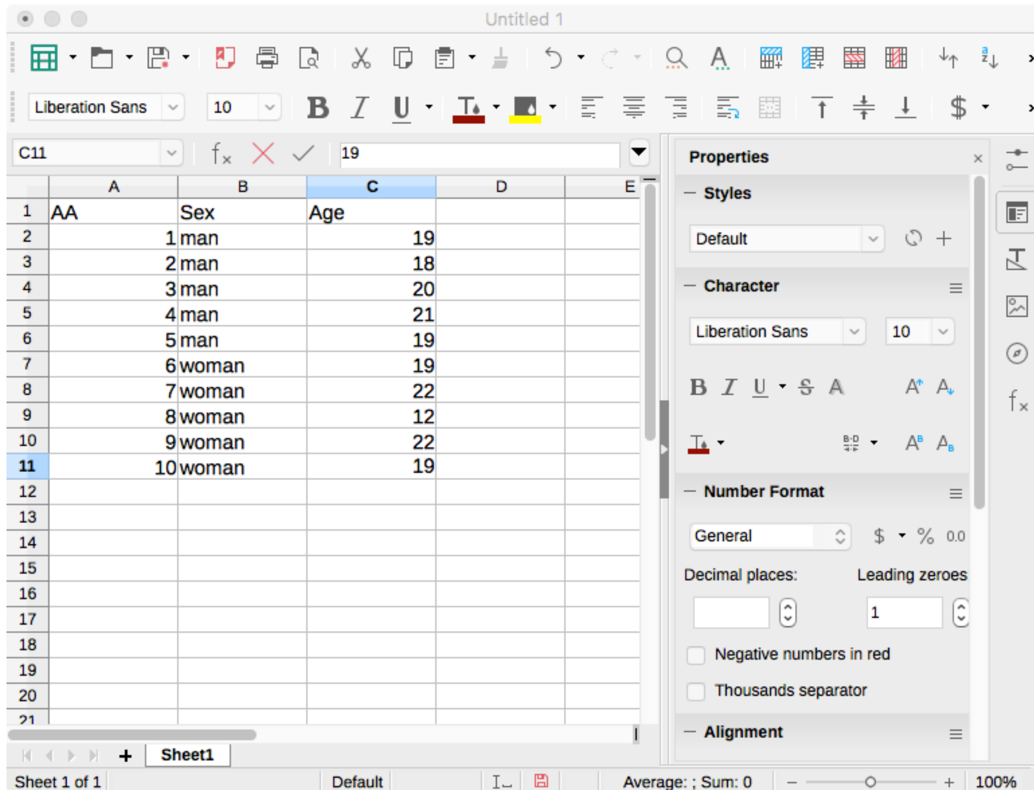
Figure 2: R Data Editor.

However, the data input window does not provide the flexibility offered by applications, such as Calc and Excel.

6.2 Import data from text files

Data are often created in applications (such as Praat, DMDX, and PsychoPy) and electronic questionnaires (such as SurveyMonkey or Google Forms) in different formats such as excel sheets (*.xls), simple text files (*.txt), and comma separated values (*.csv) From these the csv files are the most commonly used when importing data in R because they are simple text files and their format can be accessed

by all applications.



	A	B	C
1	AA	Sex	Age
2	1	man	19
3	2	man	18
4	3	man	20
5	4	man	21
6	5	man	19
7	6	woman	19
8	7	woman	22
9	8	woman	12
10	9	woman	22
11	10	woman	19

Figure 3: Table format LibreOffice Calc.

To open a csv file you can use the function read.table() or read.csv().

R code: 6.2.0.

```
read.table (path,  header = TRUE, sep  = ",")
```

This function accepts three main parameters:

1. The path to the file. It should be entered in quotes. The import method differs depending on the operating system:

- On Windows the path, the path locations are separated by two backward slashes
  e.g., `C:\\Users\\Research\\Desktop\\ mydata.csv`.
- On Mac OS X or Linux, the path locations are separated by forward slashes: `/Users/Research/Documents/RData/mydata.csv`.

2. header = FALSE: Indicates whether the first column of the table contains names of columns. If there are column names in your table then the header parameter should be set to true (header = TRUE); if there are no column names, then the header parameter is set to false (header = FALSE) (if it is false and can be left blank since the base price of the header parameter is FALSE).
3. sep = “” defines if the file has columns separated by commas (sep = “,”), or semicolon (sep = “;”) or tab sep = “\t”.

Sometimes, you open a file but R does not identify the columns and puts everything in one column. This means that the separator your chose in the option sep is not the correct one. You might have selected a comma as a sep but your data are separated by tabs. To see how your data are separated you simply type the name of the variable you used to assign the data, if the data are not in the correct format you will see the separator it is used in the file in the output and all the data in one column.

Optionally, you must determine how to separate decimals. Depending on the locale of your operating system it is possible that the decimal data values are separated by a period (.) or comma (,). So, you might need to define in R how decimal values are separated using the dec operator.

R code: 6.2.1.

```
data <- read.table  (
   "C://Users//Research//Desktop//Duration.csv",
header = TRUE, sep = ";", na.strings = "NA", dec =. "")
```

6.3 Save files

The data stored in a text file separated by commas * .csv or * .txt is the write.table () function:

R code: 6.3.0.

```
write.table (data, "c:\\data.txt", sep = ",")
```

The data can be stored in many formats. To save a dataframe as an Excel Work Sheet (*.xls) we can use the *xlsReadWrite*.

R code: 6.3.1.

```
library ("xlsReadWrite")
write.xls (Data, "c:\\data.xls")
```

To save the dataframe in SPSS format you can use the *foreign* package.

R code: 6.3.2.

```
# Write out text datafile and an SPSS program to read it
library (foreign)
write.foreign (Data, "c:/mydata.txt", "c:/mydata.sps",
   package = "SPSS")
```

7 Workflow in R

Before we close this chapter it is important to note that the R script should have a clear structure to allow the researcher or someone else to understand and reproduce the analysis. An example of a structured script is the following:

R code: 7.0.0.

```
rm (list  = ls  (all =  TRUE)) #clear memory
# Packages -----
<load packages>
# Import data -----
<Data entry code> #
Descriptive  statistics
<Code  descriptive  statistical  analyzes>
```

```
# Plots -----
<Graphs  code>
#  Statistical  analysis -----
<Code of statistical analysis>
```

At the beginning of the file, I add the function

R code: 7.0.1.

this function clears all objects in memory when starting a new analysis.

To conclude, it is always a good practive to provide comments next to the code and clarify what you are doing for future reference or when you share code with other people. In R comments are introduced with the # symbol as in the example above.