BeginR

The Old Testament

David Rengel @IPBS

March 2022

Chapter 1 : Introduction

1.1 R ????

This is an extract from "The Comprehensive R Archive Network" home page:

*R is "GNU S", a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques : linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc. Please consult the R project homepage (www.r-project.org/) for further information.

CRAN is a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R. Please use the CRAN mirror nearest to you to minimize network load.*

On the website mentioned above, R users (useRs!) of any level will find the necessary resources, that is: installation files, updates, libraries (aka packages), FAQs, newsletters, docs etc.

Why use `R`:

- `R` is free.
- `R` is increasingly popular and has become the «standard» for data analysis and visualization in many research areas, both in private and public domains (the figure he below shows the exponential growth of `R` libraries downloads since the early 2OOOs)
- `R` is powerful and versatile.
- `R`provides endles graphical possibilities.
- The `R` community.

The reasons to use `R` are summarised in this phrase by Kan Nishida, taken out of the `blog.exploratory.io` site. Well, sort of:

`R` is like wine, the more you experience it, the more you appreciate what it does, how it does, and why it does. You might hit the initial learning curve, but after you overcome it, then you start feeling how it is beautifully and practically designed to address very common challenges of the everyday data analysis.

1.2 First steps on RStudio

RStudio is an interface for R in order to make it friendlier, easier to use. When you open RStudio, R is being opened in the background as well. Indeed, if you have not installed R, you will not be able to make RStudio work.

When you open RStudio for the first time, you will find three windows. The main one, on the left, is the console. This is where all commands will be executed. in a lab analogy, the console is the bench, that is where you carry out your experiments, where everything works... or does not. The > icon you see in the console is the prompt. When you see it, it means that it is ready for the next command. On the contrary, if you see +instead, it means that it is waiting for something else. If you want to escape from the +and recover the prompt, you should press Esc or Ctrl+C.

If we continue with the lab analogy, you'll need a lab-book. A place where you will write your protocols, the ones that you will be running at the bench. That lab-book is called the source in the programming world. You may open your source window in RStudio by clicking on the double-screen icon on the source tab that you have on the upper side of your screen. Also, you may simply select the File menu, then New file and finally R script. You can have several scripts open at the same time. The source will include your code, that is your commands and your comments to the work you will be carrying out. Your code, also called script, will keep track of your work and, once it has been saved, you, or somebody else, may reopen it and reproduce your work later on or somewhere else. You can save your code at any time, and we suggest you do it regularly. You may do so by clicking on the floppy disk icon or by choosing File then Save. Code names will change colour once they have been saved.

We may all agree that you can improvise at the bench, let that be because you have realized that your lab-book is incorrect, incomplete, or for whatever reason. However, you need to eventually write those modifications into the lab-book; otherwise, you will need to improvise every time you go to the bench. The same is true here, you may well execute everything at the console, but if you leave no trace in the source, you will be compelled to amend or, worse, to make the same mistake, over and over again. This lack of rigor would lead to a waste of time and to people not understanding your lab-book.

The commands on the code or script will be executed in the console using the Run button on the upper menu or, even better, by hitting Ctrl + Enter(Windows) or Cmd + Enter(Mac). You may execute several lines of your code at once by selecting them on the source window and then executing them as shown.

It is important to note that any line starting with # on your code will be considered a comment and will not be executed on the console. We may select several lines in the code and convert them into comment lines using Ctrl+Shift+C (Windows) or Cmd+Shift+C(Mac). Comments are a very important element on any script, as they will allow you or any other useR going through the code understand the logic of the commands used in it.

Please note: RStudio allows the use of loads of shortcuts intended to make your life much easier, especially regarding repetitive actions. I suggest you have a look at the Tools menu. Shortcuts may be modified as well by the user.

1.3 Working Directory (aka wd)

When you open RStudio, there will be a working directory (or wd) defined by default in your computer: if you are using Windows OS the default wd is your *Documents* folder; if you are using a mac OS, your default wd should be *"/Users/yourusername"*. The wdis the folder in which all saved objects will be located.

You may know your wd by running **getwd()**. You are also free to change your it if you wish. There is no obligation as to how useRs should choose their wd, but we strongly suggest you use at least one wd per project. This will help you avoid file overwriting and improve traceability. Under RStudio, you may change your wd from the menu Session -> Set Working Directory. When you do so, you will realize that, in the console, Rruns the **setwd()** function (for Set Working Directory). One of the options you'll get in that menu is setting your wd to the source code location. We strongly suggest using this option.

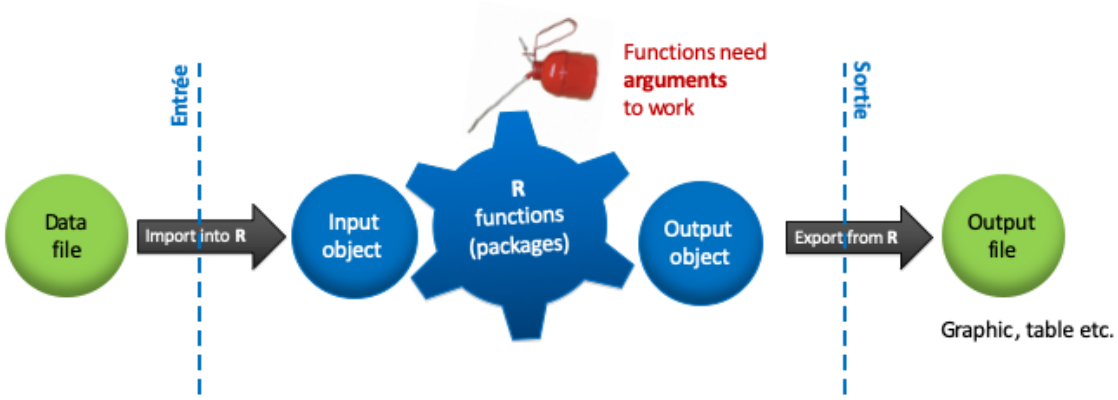
You may access to the list of the physical files on you `wd` by running **dir()** or **list.files()**. This is equivalent to using `Windows Explorer` on Windows or `Finder` on a Mac. It is also good practice to create subfolders as you work. You may create one for your data, another one for your outputs etc. You may create subfolders within your `working directory` with the **dir.create()** function. You may actually use it to create any folder anywhere in your computer, as long as you specify the right relative path to the chosen location. You may navigate using `"./whatever"` to go to or to create the subfolder called `whatever` within your current directory. Likewise, you may use `"../"` to go one step higher in the folder architecture from you current directory.

1.4 `R` Projects

One way of sticking to good practice and having one directory per project is using the `R Projects` proposed by R Studio. This will allow you find yourself at the right working directory every time you open the project without using the **setwd()** function. It will also make easier creating sub directories and so on in the folder. Importantly, if you need to be working on two, or more, projects at the same time, it will make the whole think much easier: every time you click on a `R` project, a new `R Studio` session will be opened where the `working directory` will correspond to the directory where the `Rproj` file is in.

1.5 `R workspace` or working environment

When you work with `R`, you will most certainly find yourselves somewhere along the path shown here below. First, you will have a dataset that you will import into `R`. From this moment on, the dataset is an `R` *object* under the `R` *environment*. You will then use `R` functions and packages to manipulate and/or analyze it so that, in the end, you will have created one or several `R` output objects that, finally, you will export as a `pdf`, an image, a table file or whatever suits your object and your goal best.



All `R` objects in your `R` session (let that be datasets, variables, functions you have created, function outcomes etc.) will conform your `workspace`). Do not mistake it with the aforementioned **working directory** (aka `wd`), which is the physical place on your computer you are located in. In other words: if you shut down your `R` session without saving your `workspace`, all your `R` objects in the closed session will be definitely lost. On the other hand, your working directory will still be there, obviously.

You will be able to list on the console all the objects you have created on your `workspace` by running **ls()**. Your whole workspace, or any object in it individually, may be saved as an `.RData` file in the `wd` (or elsewhere, if you wish, but we suggest you do not so). You may hence save your `workspace` by running the **save.image()** function like this: `save.image("myfile.RData")`. Likewise, you may load a previously saved `R` workspace by running `load("filename.RData")`. This will load all `R` objects in you `.RData` file. If you want to save a single object from your `workspace`, the function to be used is **save()**: `save(myobject, file = "myfile.RData")`.

From the moment you execute the **save()** or **save.image()** functions, the created `.RData` file will physically appear on you computer, in the folder you have chosen to save it, most usually your `wd`.

1.6 The prompt

The use of the *prompt*, i.e. `>`, means that everything that you will run in the `R` console will require that you use *functions*, and those functions will have *arguments*. Therefore, there will not be any pre-established, mouse-clickable menu for your analysis. In other words, you will not find any menu allowing you to perform an ANOVA or a PCA, for instance: you will have to run the suitable functions, with your chosen parameters, by yourself.

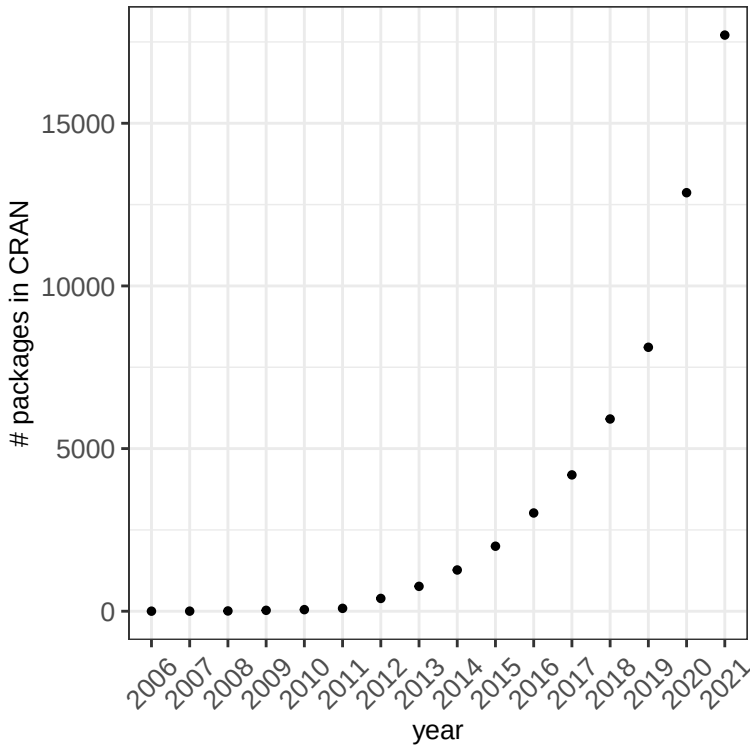
Indeed, `RStudio` interface has made our lives easier in many aspects when using `R`, but it has not changed the core logic under `R`. As a matter of fact, the *prompt* line is the source of the power and versatility of `R`. Pre-established menus restrict choices and hamper versatility. The *prompt* line will require a time investment and energy form your part. In return, it will increase dramatically traceability, provide room for adjustment, and offer nearly unlimited analysis and visualization possibilities.

A function in `R` is called with its name followed by its arguments enclosed by brackets. If no arguments are specified, that is in the case where no argument at all is required, the function will be called by its name followed by empty brackets. If you do not put the brackets and you just execute the name of the function, then the code of the function itself will be rendered, which in most cases is not useful.

When you are writing your code, a very important feature in `R` is the # character, which allows to write comments on your code. Anything written after the # will be interpreted as a comment and will not be executed as code chunk. Indeed, if you are using `RStudio` you will realize that as soon as you write #, the colour of the font changes, thus indicating that what you are about to write will be considered as comment.

1.7 Using libraries (aka packages)

Libraries or *packages* are groups of functions and/or datasets developed within the R community, allowing to accomplish specific tasks. In September 2019, the *Comprehensive R Archive Network* featured 14476 packages. Since the date of the submission of the first package on the 15th March 2006 until the last submission recorded for this document on the 26th October 2020, the number of featured packages has grown exponentially (see the figure below).



Using a package requires two steps:

1 - *Installation*: you may use the **install.packages()** function to retrieve the source codes of the package and install it on your computer. This will be done only once. However, it is advisable to periodically update the packages by running **update.packages()**.

2 - *Loading*: You will use the function **library()** to load the package you have already installed so that you may use it during your session. You will need to do so every time you launch `R` or `RStudio`. The latter allows you to load the package by checking out its name in the list of the installed packages provided with the interface.

You may also get the list of installed packages by running **library()**. The **search()** function allows you to verify the packages that have been loaded. Finally, we can use **ls()**, specifying the package number given by **search()**, in order to know the functions contained in a given package. See below the example with the package *foreign*.

RStudio provides easier, ready-to-click menus to load packages etc. (see the `Packages` tab at the bottom-right window). You will realize that clicking on a package executes the **library()** function straight into the console. However, this will not leave any trace on your code, and this is why we do advise you not to do it, since it is always a good thing to keep a trace on everything you have done to run your code.

```
# Installation of the package :
install.packages("alluvial")
# Verify that the package is among the already installed ones:
library()
# Loading the package
library(alluvial)
# Verify that the package is loaded and, hence, its functions ready to be used
search()
```

When you run **ls()** you are looking at what is in that position 1 under **search()**. That position corresponds to *.GlobalEnv*, that is your `workspace` or working environment. The other positions correspond to the environments of subsequently loaded packages, and they are not usually touched. Inside those environments

you will find all functions and datasets belonging to corresponding `package` and that have been loaded when you execute the **library()** function. In general, the last loaded `package` will be at position 2 under **search()**, immediately after your current working environment.

```
# List available functions in the first loaded package.
ls(pos = 2)
```

It is worth pointing out that the installation of packages featured in the Biology-devoted `bioconductor.org` repository might be carried out by using the installation code provided by Bioconductor, as shown here below for the *limma* (for `Linear Models for Microarray Data`) package. Note that the Bioconductor code proposed here below may not be adapted for R versions below 3.6.

```
if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
biocLite("limma")
```

A third repository for R packages is GitHub. There are several ways to install packages from GitHub. A common way to do so is using the *install_github* function from the remote package. To use it, you will need to install and load the remotes package first, then.

```
install.package("remotes")
library(remotes)
install_github('the_nameof_the_package')
```

1.8 Help

`UseRs` of all levels are encouraged to seek help in any available way instead of wasting their time trying to figure something out all by themselves. There are many ways to ask for help, both on and offline. A thorough view on the ways you may ask for help is given at https://www.r-project.org/help.html and http://search.r-project.org/.

When you are working on your code and you need information about a particular function, its usage or its arguments, the first reflex to get help should be to type **help(yourfunction)** or **?yourfunction**, both rendering identical results, that is the help on the chosen function that the developer provided when he or she submitted the package. Beware that **help()** or **?** work also off line, since the help pages come with the package when you install it on your computer. However, they only work as long as the package has not been only installed on your machine, but also loaded into your workspace.

You may need help on a function whose package you do not remember, or you want to find help of any function related to a given term, let us say *glm*. In this case, you will use **help.search("glm")** or **??glm**. This will result in a list of functions whose name contains the *glm* term, among all packages already installed on your machine, whether the package has been loaded or not.

```
help(plot)
?plot

help(plot)
?plot

help.search("plot")
??plot

help(help.search)
help(help)
```

Overall, help pages comprise the following headings:

- Description: it defines briefly what the function does

- Usage: how the function should be used
- Arguments: The parameters that the function accepts, with some explanations on their use.
- Details: Further explanations on the logic behind the function
- Value: Details on the outcome of the function, that is what you are meant to get after using it
- Note
- Authors
- Reference(s)
- See also: Other functions related to the current search
- Examples: Ready-to-use examples, so that the user may get an idea of how it works.

Those headings are, for the most part, mandatory when the developers submit their package to the CRAN repository. However, some help pages are much more informative than others, depending on developers willingness.

Google itself is very helpful tool as well. If you write any question on the browser followed by the letter *R*, such as "ANOVA R" or "visualization quantitative data R", Google understands you are referring to the `R` language and will duly provide `R`-related hits. The `R` community is so strong that it is very likely that somebody has already asked the same question before you and that somebody has already answered it. If you do not find the answer to your question, you may ask for it in the right forum. There are very conspicuous internet forums devoted to `useRs` seeking for answer for their codes etc., such as `StackOverflow`, `StackExchange` or the `Bioconductor support` webpage for packages accepted in the Biology-devoted `Bioconductor` repository. If the package you are seeking help about does not explicitly mention any forum to address your questions to, you may always contact the developer of the package.

Chapter 2: Data Structures

In this chapter we will see the main data structures we treat under `R` and how we handle them.

`R` language is largely based on the objects that will be created by running functions. There are different data structures depending on their number of dimensions and the data types they are able to receive. Thus, data structures might be mono-, bi- or n-dimensional and they will be able to accept either a single kind of data (homogeneous data) or different kinds of data (heterogeneous data). The table here below summarizes all that for the most currently used objects in `R`:

	Homogeneous data	Heterogeneous data
1-dim	`vector`	`list`
2-dim	`matrix`	`data.frame`
n-dim	`array`	...

It is worth pointing out that there is no zero-dimensional object under `R`, that is scalars. Indeed, numbers or single-word character strings, which might have been considered as scalars, are indeed vectors on length = 1 in `R`.

2.1 Store a value in a variable

`R` can be used as a mere calculator.

```
2+2
2+2*5
(2+2)*5
3*3*3/2
3^3/2
-10/3
pi
```

```r
sin (2*pi/3)
sqrt(4) # Any character after "#"" is considered as a comment
```

However, tha main goal of using R, or any other similar working environment, is to store values and analysis outputs in R objects.

```r
n = 15
n <- 15
5 -> n
N <- 10
(N <- 30) # Creates the object and gives results at the same time
y <- n*6
rm(n)
n
y

a <- log(2)
b <- cos(10)
a
b
a + b
a <- 3
b <- 6
a + b / a + b # ?
(a + b) / (a + b)
```

2.2. Using functions

The objects you will be creating in R will not be, for the most part, the product of simple, straight forward calculations. For the most part, the objects you will produce will be the result of using R functions.

When you use a function in R, you will need round brackets to contain the `arguments` to be used with that function in particular. The arguments in a function might be `mandatory` or `optional`. The former must be made explicit to make the function work. The latter might be omitted and the function will be executed with the value that was fixed by default by the developers. When you use R, it is strongly advised to be aware of what the arguments in a function are meant to be used for, including of course the default values of the optional arguments.

```r
log(2)
?log
exp(1)
log(exp(1))
log(2, base = 2)
log(c(2,4,58), base = 2)
log(50, base = 2)
log2(50)
log(50, base = 10)
log10(50)
?log10
log10(50, base = 2)
log10(2);log2(2)
```

2.3. More than numbers

When using R, you are meant to produce and to use more than numbers. You will certainly create string, character variables as well as boolean, logical variables. Logical variables are produced in response to questions requiring boolean reply, which are the only questions you are supossed to ask. Boolean responses correspond to those questions offering binary and mutually exclusive alternatives, i.e. TRUE or FALSE. Boolean responses are easily converted, or coerced, into numeric values: TRUE equals to 1, and FALSE equals to 0.

There are four ways of asking such questions: == (equal to), != (not equal to, or different from), < (smaller than) or > (bigger than).

```
2 == 3
2 != 3
2 < 3
2 > 3

c <- 2==3
d <- 2<3
c + d
```

Characters go between quotes. Likewise, R is meant to be used with non-numerical variables and objects.

```
a
a <- "toto"
e <- a
e <- "titi"
e
```

Within time you will be creating all sorts of objects. You may need to look for them in your workspace.

```
ls()
ls.str()
b <- 5
b2 <- b*2
ls(pat = "b")
ls(pat = "b|a") # The absences of spaces here surrounding the | is important
ls(pat = "b | a")
acdc <- "back in black"
ls(pat = "b|a")
ls(pat = "c|d")
ls(pat ="^c|^d")
ad <- "toto"
ls(pat="d$")
ls(pat="^c|d$")
ls(pat="^c" & "d$")

rm("a")
ls()
rm(pat="b") #!!
ls()
rm(list = ls(pat="b"))
ls()

rm(list=ls())
ls()
```

2.2 *Vectors*

Vectors are one-dimensional objects and they represent the simplest way to store values in a single object. Indeed, every object we have created so far in this course are vectors because there are not dimension-free objects in R. There are four main vector types: `integer` and `double` for `numeric` variables, `character`, and `logical`.

Vectors are usually created with the **c()** function, i.e. `combine`, though this is not the only way to do so.

```
c(2,3,5,8,4,6)
a <- c(2,3)
c(c(2,3),c(5,8,4),6)
c(a,c(5,8,4),6))
c(1,2,3,4)
```

Using the `column` (`:`) allows creating numeric vectors without using the `comma`, which is very useful when we create vectors with a large number of terms. It also allows us getting rid of the **c()** function.

```
c(1,4)
c(1:4)
c(1:1000)
1:1000
1,4
a <- c(2,3)
b <- c(5,8,4)
c <- c(6)
d <- c(a,b,c)
class(d)

a <- "toto"
d <- c(a,"tata")
d <- c("a","tata")
e <- c("this","is","my","vector")
length(e)
f <- c("this is my vector")
length(f)
```

Mathematical and boolean operators are evaluated term by term within a vector.

```
x <- c(10,20,30)
x
x * x + 2
x + x * 2
(x + x) * 2
x^2
3*x^2

summary(x)
min(x)
max(x)
sum(x)

mean(x)
sd(x)
var(x)
median(x)
```

We can start exploring the use of boolean vectors.

```
sum (x > 10) # Why?
mean (x > 10) # !!

pvalues <- c(0.2, 0.012, 0.03, 0.94)
fcs <- c(2,5,-3,1)
sum(pvalues < 0.05)
(pvalues < 0.05) * fcs
```

What if you have missing values in your variable.

```
x[3] <- NA
x
summary(x)
mean(x) # Why?
is.na(x)
sum(is.na(x))
sum(!is.na(x))
mean(is.na(x))
any(is.na(x))
all(is.na(x))
any(x < 10)
all(x < 10)
mean(x)
mean(x, na.rm = TRUE)
(x < 10, na.rm = TRUE)
sum(x < 10)
```

You may ask at any point what kind of vector you are dealing with. You my even ask whether you are dealing with a vector. The answer you should expect is, as always, boolean: `TRUE` or `FALSE`. And that answer is, of course, a `vector`.

```
is.double(d)
is.integer(d)
is.numeric(d)
is.character(d)
is.logical(d)
is.logical(d)
is.vector(d)
is.numeric(d)
is.null(d)

is.logical(is.character(d))
is.logical(is.numeric(d))
is.logical(d)
is.logical(is.logical(d))
is.vector(is.logical(d))

is.numeric(2)
is.logical(FALSE)
is.logical(T)
is.logical(F)
is.logical("FALSE")
is.logical("TRUE")
```

Here is the reason why you should not create variables that are called `F` or `T`.

```
F <- "whatever"
a <- rnorm(10)
b <- rnorm(10)
t.test(a,b, var.equal = F )
t.test(a,b, var.equal = FALSE )
rm(F)
t.test(a,b, var.equal = F )
```

Numeric vectors can be `integers` or `doubles` though by default they will be the latter. The diffference between both will be, for the most part irrelevant in your daily work. However there are certains aspects that you should bear in mind: `integers` are precise and `doubles` are approximations. Have a look at the following lines:

```
is.double(2)
as.integer(2)
is.double(2.1263589)
as.integer(2.1263589)
as.integer(2.9263589)

is.double(Inf)
as.integer(Inf)

sqrt(2)^2
sqrt(2)^2 == 2
sqrt(2)^2 - 2

20/3
20/3 == 6.666667

round(20/3, 2)
round(20/3, 6) == 6.666667

x <- c(1,2,4)
is.numeric(x)
is.integer(x)
is.double(x)
x
as.integer(x)

x <- c(1L,2L,4L)
is.numeric(x)
is.integer(x)
is.double(x)

x <- c(1,2.7,4.3)
is.numeric(x)
is.integer(x)
is.double(x)
x
as.integer(x)

x <- c(1L,2.7L,4.3L)
x
```

```
is.integer(x)
is.double(x)
as.integer(x)
```

2.2.1 names in a vector as an attibute

Let us have a look at the notion of `attributes` of a vector. `Attributes` are additional information about a variable, that come out as a `list` (more about `lists` later on), and do not modify in any sense the actual information provided by the variable itself. The most common and widely used attribute of a variable is the `names` attribute.

```
d <- c(23,456)
attributes (d)
f <- c("Luke", "Yoda")
names(d) <- f
d
attributes (d)
attributes(d)$who_are_those <- "My vector only accepts Jedis"
attributes(d)$what_is_this <- "those are the ages of: Yoda is 456 and Luke is 23"
attributes(d)
is.numeric(d)
rm(f)
d
names(d) <- c("Yoda", "Luke")
```

2.2.2 Vectorisation

We will mention the notion of *vectorisation*, an important feature in `R` which is meant to make calculations easier and faster. Do not worry about the term itself, just see what is going on in the following lines. `UseRs` should be aware that vectors are considered, in calculation terms, as column vectors. Even if you see the elements in a vector displayed horizontally on your screen, this is only for practical reasons. The fact that vectors are, in fact, column vectors has some implications that we will be looking at further down in the document.

```
1:6 + 5
1:6 * 5
1:6 * c(2,3)
1:6 * c(2,3,4,5) # Why?
```

2.2.3 Subset and extract information from a vector

We use `[]` to subset variables in `R`, that is extracting elements and information from a variable. We can also use `[]` to modify the variable.

We extract the information from a variable by calling between `[]` the index or the name of the element(s) in the variable for which we want to retrieve the information.

```
ages <- c(17,39,35,54,20,47)
names(ages) <- c("Enzo", "Pierre", "Ana", "Celine", "Yvan", "Lola")

ages[]
ages[2]
ages[2:4]
ages[c(2,4)]
ages[2,4]
ages[4:2]
```

```
ages[-2]
ages[c(-2,-3,-4)]
ages[-c(2,3,4)]
ages[-c(2:4)]
ages[-(2:4)]
ages[-2:4] # Why?
```

We can also choose the elements located where the response to a boolean question is `TRUE`.

```
ages < 20
ages[ages < 20]
ages[ages <= 20]
ages[ages == 20]
ages[ages == 25]
ages[ages != 54]

which (ages <= 20)
ages[c(1,5)]

ages[ages > 20 ]
ages[ages > 20 & ages < 50]
ages[ages < 20 & ages > 50]
ages[ages < 20 | ages > 50]
```

We can subset a variable calling the names of the elements.

```
ages["Ana"]
ages[ c("Ana", "Enzo")]
```

We may pick names with a pattern using **grep()** and **grepl()** functions, and I think we should. Both functions pick terms with the requested string of characters, but they differ in the result they produce: **grep()** renders the index of those elements in the variable complying with the requested string whereas **grepl()** produces a logical vector, stating whether the elements in the variable comply with the chosen pattern (`TRUE`) or not (`FALSE`).

```
grep("E", names(ages))
grepl("E", names(ages))

ages[grep("E", names(ages))]
ages[grep("E|e", names(ages))]
ages[grep("n", names(ages))]
ages[grep("a|n", names(ages))]
ages[grep("a.*n", names(ages))]
ages[grep("a.*n", names(ages))]
ages[grep("^E|a$", names(ages))]

ages[grepl("E", names(ages))]
ages[grepl("E|e", names(ages))]
ages[grepl("n", names(ages))]
ages[grepl("a|n", names(ages))]
ages[grepl("a.*n", names(ages))]
ages[grepl("a.*n", names(ages))]
ages[grepl("^E|a$", names(ages))]
```

You can obviously subset a variable according to the values of another variable.

```
heights <- c(178,182,165,172,156,194)
names(height
heights[ages > 20]
```

2.2.4 Sorting information in a vector

We will use two functions to sort the information within a variable: **sort()** and **order()**. They may seem similar and it is almost guaranteed that a new comer to R will mistake them at some point. We will describe those two functions in the following lines. We will mention also **rev()**, a somehow related function to those two. The function **sort()** renders the requested information in increasing, or decreasing, order. On other hand, **rev()** renders the information in a mirror-like reverse manner.

```
x
sort(x)
sort(x, decreasing = TRUE)

rev(x)
rev(sort(x)) == sort(x, decreasing = TRUE)
```

You see in those example above that we did not use `[]`. Indeed, we are not subsetting nor extracting any information from `x`: we are just displaying the information in the variable in a given order. Indeed, using sort between `[]` does not make sense because **sort()** does not produce `index` numbers, as the lines here below will prove.

```
sort(x)
x[c(17,20,35,39,47,54)]
```

What **sort()** does is indeed different from what **order()** does. The latter does two things, actually: first it sorts the variable as **sort()** itself would do, and then it renders the index number at which sorted terms are located in the variable.

```
order(x)
x[order(x)]
x[c(2,5,1,3,6,4)]
x[order(x)] == sort(x)

order(x, decreasing = TRUE)
x[order(x, decreasing = TRUE)]
x[order(x, decreasing = TRUE)] == sort(x, decreasing = TRUE)
```

That is why **sort()** does not work between `[]`: it does not produce `index`. On the contrary, **order()** produces ìndex information, thus being useful to subset sorted information between `[]`. The results of order should be read as, for instance, "the smallest value of x is the second element of the x vector" or "the highest value of x is the fourth element of the x vector". The use of **order()** might not seem clear at this point, but it will be clearer when we deal with two-dimensional datasets.

unique() and **duplicated()** functions are useful to retrieve elements in a variable that might be duplicated (or not). The output of both functions is different: **unique()** produces once all elements in a given variable, beyond any eventual duplication. Hence, the size of the **unique()** outcome will be smaller than that of the original variable if there were any duplications.

On the other hand **duplicated()** will read the variable element by element and will tell us if the current element has already been read or not. Hence, the outcome of **duplicated()** is a logical vector with the same size as the original variable.

```
a <- 3:5
a <- c(a,4:8)
```

```
a
length(a)
unique(a)
length(unique(a))

duplicated(a)
length(duplicated(a))
```

2.2.5 Coercion

All elements in a vector must be of the same kind. That is, you can not store, for instance, characters and numbers in the same vector. Consequently, when we try to combine different kind of data within the same vector, **coercion** will be applied so that the vector will be converted into the most flexible type. Vector types from least to most flexible are: logical, integer, double and character.

In the next code chunk, we will introduce the notion of `coercion`. This is a very important concept in `R` and we are sure it will annoy you at some point when you start coding. That said, it is also a cornerstone of the language, you will learn to cope with it in due time, and you may be able even to take advantage of it.

`Coercion` is implemented when the `useR` tries to stock heterogeneous data in an `R` object whose structure does not accept heterogenous data. In this situation, `R` will accept the input of heterogenous data but the object will be coerced to be converted to the type of data that is the most flexible one given all data inputs.

The following data types are increasingly flexible: `logical`, `integer` (whole numbers), `double` (numbers with decimals, for the most part), and `character`.

```
x <- c(2,5,"toto")
x
summary(x)
is.numeric(x)
is.character(x)

toto <- c(1,FALSE)
c(1,"toto", FALSE)
c("a",1)
c(1,F)
c("a",F)
c("a","F")
F <- pi
c("a",F)
rm (F)
c("a",F)
1 == "1"
2 == "1"
1 == "one"
1 == TRUE
-1 < FALSE
"0one" < 2
"1" < 2
"10000000000000000" < 2
"a" < "b"
```

You may force coercion into a perfectly valid and homogeneus vector by using the corresponding `as.thedatatypeIwant()` function

```r
c(0,2,7)
as.character(c(0,2,7)) # ?
as.factor(c(0,2,7))
as.logical(c(0,2,7))
```

2.2.6 Factors

We should mention now another kind of vector, that is `factors`. At first sight, `factors` may look as `character`, or even as `numeric` if we are dealing with sort of numeric scores (as we do with apoptosis or with the infamous Defcon scale for nuclear menace). However, `factors` are variables comprising a fixed and known set of possible values. Indeed, they are used to work with categorical variables, where the expected and/or the observed values are referred to as `levels`.

`factors` are also useful when you want to display character vectors in a non-alphabetical order.

`factors` can be ordered or unordered and are an important class for statistical analysis and for plotting.

```r
Let us take the following variable
dose_char <- c("low","high","medium","high","low","medium","high")
```

That variable presents, at least, two issues. 1-We cannot easily tell how many times each identical term in the variable appears. 2-Sorting that variable does not provide any valuable information.

```r
summary(dose_char)
sort(dose_char)
```

All those issues may be addressed by simply defining our variable as a `factor`. That, in itself, will allow us to count how many timmes each term appears in the variable.

```r
dose_fct <- factor(dose_char)
dose_fct
summary(dose_fct)
```

The `levels` in that factor are determined, by default, in alphabetical order. By making explicit the expected `levels` in the factor, we can specify the order in which we want them to be consudered. This is important when producing plots as well as in such statistics methods such as, for instance, ANOVA.

```r
dose_fct <- factor(dose_char, levels = c("low","medium","high"))
dose_fct
summary(dose_fct)
```

Sometimes, you may need to specify the order also in quantitative terms because it is meaningful or because it is required by particular type of analysis. Additionally, specifying the order of the levels allows us to compare `levels`.

```r
dose_fct
min(dose_fct) # !! Look at the error message.
dose_fct[2] > dose_fct[1]

dose_fct <- factor(dose_char, levels = c("low","medium","high"),
                   ordered = TRUE)
dose_fct
min(dose_fct)
dose_fct[2] > dose_fct[1]
```

That same logic applies when you are dealing with categorical scores that resemble `numeric` values.

```r
num <- c(4,2,3,1)
num[1] > num[2]
```

```r
sum(num)

fct <- factor(num)
fct
min(fct)
fct[1] > fct[2]

fct <- factor(num, ordered = TRUE)
min(fct)
fct[1] > fct[2]
sum(fct)

fct <- factor(num, levels = c(4,1:3))
min(fct)

fct <- factor(num, levels = c(4,1:3), ordered = TRUE)
min(fct)
fct[1] > fct[2]
```

There is a final issue when dealing with categorical variables. You may have expected values for which there is no actual observations. In our dose_fct example I may have expected, for instance, observations considered as `placebo`. In that case, not having any `placebo` among our observed values may be a valuable information in itself. However, our variable does not tell us anything about it. We can sort that out by including those expected, but not observed, values among our defined `levels`.

```r
dose_fct <- factor(dose_char, levels = c("placebo","low","medium","high"))
dose_fct
summary(dose_fct)
```

Defining all putative expected values allows us also to declare as a missing value any observed value that does not match with any of the expected levels.

```r
bad_fct <- dose_fct
bad_fct[3] <- "not treated"
bad_fct
```

Let us say that the variable in your dataset is already a factor but you are not happy with the reference level, that is the level that will be the reference for further statistics analyisis or plotting. You may customize that with the **relevel()** function.

```r
relevel(dose_fct, ref = "high")
```

2.2.7 Vectors with a pattern

We use **seq()** and **rep()** to create vectors with a "pattern", let that pattern be either `sequential` or `repetitive`.

```r
1:10
seq (from = 1, to = 20, by = 2)
seq (from = 1, to = 20, by = 5)
seq (1, 20, 5)
seq (20, 1, 5)

seq (20, 1, -5)
seq (to = 20, from = 1, 5)
seq (to = 20, 1, 5)
```

```
seq (1, 20, 5)
seq (t = 20, f = 1, 5)

seq (1, 20, by = 5)
seq (1, 20, length.out = 5)

rep (5, times = 10)
rep (5, 10)
rep (10, 5)
rep (c(1,2), 3)
rep (c(1,2), each = 3)
rep (c("wt","mut"), 2)
rep (c("wt","mut"), each = 2)
```

2.2.8 Vectors with randomized elements: sample()

We should mention at this point the **sample()**, which allows creating randomized variables, and might be very useful to simulate data, building learning datasets etc. An important parameter of that function is `replace`, which is set to `FALSE` by default. This means that, in order to make variables with repeated elements, yuo need to specify `replace = TRUE`.

```
sample(10)
sample(3:10)
sample(letters)
sample(c(1,15,-7))
?sample

sample(10, size = 3)
sample(10, 3)
sample(1:5, 10) ## !!
sample(1:5, 10, replace = TRUE)
sample(letters[1:3], 50, replace = TRUE)
sample(c("WT", "KO"), 50, replace = TRUE)
```

Since **sample()** produces random results, you will obtain a different result every time you run it. This is true for every math or statistical procedure requiring randomness, such as `kmeans` or `random forests`.

You may fix an otherwise random result by executing **set.seed()** just before the randomizing command. **set.seed()** requires a value as seed argument. Everytime you use the same seed value, you will obtain the same result. And this should be so on any computer. The result will change if you change the seed value.

```
sample(10)
sample(10)

set.seed(1)
sample(10)

sample(10)

set.seed(1)
sample(10)

set.seed(2)
sample(10)
```

```
set.seed(2)
sample(10)
```

Exercise 1

- Execute the lines here below. They show four vectors with different features of ten different people.

```
heights <- c(182,180,171,178,169,176,165,177,168,184)
weights <- c(91,81,65,80,66,75,55,73,67,94)
ages <- c(24,33,45,18,57,62,39,42,74,27)
gender <- c("M","F","F","M","M","M","F","F","F","M")
```

- What kind of vectors are they?

```
class (heights)
class(weights)
class(ages)
class(gender)
```

- Convert the "gender" vector into a factor

```
gender <- factor(gender)
```

- What is the mean height?

```
mean(heights)
```

- What are the heights of those individuals measuring between 170 and 180 cm?

```
heights[heights > 170 & heights < 180]
```

- What is the proportion of those individuals?

```
mean(heights > 170 & heights < 180)
```

- How many people are taller than 180 cm?

```
sum(heights > 180)
length(heights[heights > 180])
```

- What is the median weight?

```
median(weights)
```

- What is minimum body mass index among those individuals (BMI = masse in kgs divided by the squared height in meters)

```
min(weights/((heights/100)^2))
```

- Can you create a vector that goes from 1 to 20, including only even numbers?

```
seq(2,20,2)
```

- Could you slice the interval between 1 and 20 in nine equally sized slices?

```
seq(1,20,length.out = 10)
```

2.3 *matrix*

`matrix` objects allow storing elements in a two-dimensional table, that is with rows and columns. However, all data in a `matrix` must be of the same kind. If this is not the case, all elements in the `matrix` will be coerced into the most flexible data type. Most often than not, your own two-dimensional dataset will contained in

`data frames` rather than `matrices` (more about `data frames`, later in the document). The main difference between `matrices` and `data frames` is precisely the fact that the latter can store heterogenous data. However, there are occasions where you will need to deal with matrices, either because the math analysis you want to carry out requires so, or because the output of the analysis you have just carried out is `matrix`. The code here below shows how to create and manipulate matrix objects. You will not be creating matrices with the **matrix()** function very often because, as I stated above, it is more likely that you will be dealing with matrices that have already been created for you.

```
set.seed(1)
M1 <- matrix(sample(1:24), ncol = 4)
M1
colnames(M1) <- paste0("var.",1:4)
rownames(M1) <- names(ages)
dimnames(M1)
t(M1) # To trnaspose the matrix

set.seed(2)
M2 <- matrix(sample(1:24), ncol = 4)
M2
colnames(M2) <- paste0("var.",5:8)
rownames(M2) <- sample(names(ages))
```

We will briefly mention that, when using **matrix()**: 1-One may specify the number of columns with `ncol`, or the number of rows with `nrow`. You should not need both. 2-Matrices are filled up columnwise by default. You can fill them up row-wise by changing the argument `byrow` to `TRUE`.

```
matrix(1:15, ncol = 5)
matrix(1:15, ncol = 5, byrow = T)
matrix(1:15, nc = 5)
matrix(1:15, nr = 3, b = T)
matrix(1:15, nc = 5, nr = 5) # !!!
```

2.3.1 Matrix subsetting

The main difference with vector subsetting and manipulation is the fact that, with matrices, two indexes are needed, one for the row and the other one for the columns. Hence, square brackets are used as follows: **`[row, column]`**.This way of subsetting is also valid for `data frames`, as we will see further down

```
M1[1,3]
M1[,2]
M1[2,]
M1[2:3,2:4]
M1[-1,-c(1,3)]
M1[c(1,3),c(2,4)]
M1
M1[c("Celine","Pierre"),c("var.4","var.2")]
M1[c(3,1),c(4,2)]
```

When you choose one single row or one single column in a matrix, `R` will vectorize it by default, meaning that it will lose its two-dimentional structure and become a vector. This migh be inconveninent in some cases, for instance when you want to keep the names of that row or column.

```
M1[,1]
class(M1[,1])
M1[,1, drop = FALSE]
class(M1[,1, drop = FALSE])
```

```r
M1[1,]
class(M1[1,])
M1[1,, drop = FALSE]
class(M1[1,, drop = FALSE])

M1[1,1]
class(M1[1,1])
M1[1,1, drop = FALSE]
class(M1[1,1, drop = FALSE])
```

Subsetting can be produced using boolean questions.

```r
M1[M1[,"var.3"] > 10,"var.2"]
M1[M1[,"var.3"] > 10 & M1[,"var.4"] > 4 ,"var.2"]
M1[M1[,"var.3"] < 10 | M1[,"var.1"] == 14 ,"var.2"]

M1[M1[,"var.3"] > 10,"var.2", drop = FALSE]
M1[M1[,"var.3"] >= 5 & M1[,"var.8"] > 4 ,"var.2", drop = FALSE]
M1[M1[,"var.3"] < 10 | M1[,"var.1"] == 14 ,"var.2", drop = FALSE]
```

Dropping is not necessary when we pick more than one row *and* more than one column because, in such case, the information can not be vectorized.

```r
M1[M1[,"var.3"] > 10, c("var.1","var.2")]
M1[M1[,"var.3"] > 10 & M1[,"var.4"] > 4 , c("var.1","var.2")]
M1[M1[,"var.3"] < 10 | M1[,"var.1"] == 14 , c("var.1","var.2")]
```

You can obviously subset a given matrix according to what happens in another matrix.

```r
M1[M2[,"var.7"] >= 5,]
M1[M2[,"var.7"] >= 5 & M2[,"var.8"] > 4 ,"var.2"]
M1[M2[,"var.6"] < 10 | M2[,"var.7"] == 1 ,c("var.1", "var.3")]
```

And you can subset information using **grep()** and **grepl()**.

```r
M1[grep("n", rownames(M1)), c(1,4)]
M1[grepl("n", rownames(M1)), c(1,4)]

M1[-grep("n", rownames(M1)), c(1,4)]
M1[-grepl("n", rownames(M1)), c(1,4)] # !!
M1[!grepl("n", rownames(M1)), c(1,4)]
```

You can bind `matrices` column-wise and row-wise if they are dimensionally compatible. When you do the former, be sure that the order of the rows in both `matrices`is the same, and when you do the latter, you have to be sure about the variables.

```r
M1
M2
cbind(M1,M2) ## !!!
rbind(M1,M2)
```

2.3.2 Operating with matrices

Standard operations are executed term by term, as it was the case for vectors.

```r
M1*2
M1 + M2
M1 * M2
```

Also, it should be considered the fact that operations are run column-wise. Therefore, we must be aware of what we are doing to avoid surprises.

```
M1*c(1,2)
M1*c(1:4)
t(A)
t(A)*c(1:4)
A*c(1:5)
```

We use the %*%operator in order to produce the matrix product between two compatible matrices: the `ncol` of the first must be the same as the `nrow` of the second, and viceversa.

```
M1 %*% M2
M1 %*% t(M2)
```

The following lines are about the **solve()** function, which enables resolving matrix equations as well as inversing matrices. Indeed, **solve()** provides the `x` matrix so that `a %*% x = b`. This operations are very common in many math and stat procedures, such as PCA, and they will be runnig behind the curtains, many times without us realizing about it.

```
a <- matrix(c(3,5,2,4), nrow = 2, ncol = 2)
a
b <- matrix(c(8,2), nrow = 2, ncol = 1)
b
solve(a,b)
```

In the example above we have that `3x + 2y = 8` and that `5x + 4y = 2`. We use `solve()` to find that `x = 14` and `y = -17`.

The `a` matrix above must be square (i.e. `ncol(a) = nrow(a)`), so that you have as many equations to solve as you have unknown variables. On the other hand, `nrow(b) = nrow(a)`, whereas each column of `b` proposes a different result for the equations proposed by `a`. Hence, the product of **solve()**, `x`, will have the same number of rows as `a` and `b`, and as many columns as `b`, each of the columns being the possible solution for the equations in `a`, that is compatible corresponding column in `b`.

```
set.seed(1)
a <- matrix(sample(1:9), nrow = 3, ncol = 3)
set.seed(1)
b <- matrix(sample(1:6), nrow = 3, ncol = 2)
solve(a,b)
```

The inverse of a square matrix `a`, i.e. `a^-1^`, is a matrix whose matricial product with `a` produces an identity matrix, i.e. a matrix whose diagonal present unit numbers, and zeros elsewhere.

```
?solve()
solve(a)
solve(a, matrix(c(1,0,0,1),ncol=2))
```

You may realize from above, and taking ito account the `a` and the identity matrices that if `3x + 2y = 1` and `5x + 4y = 0`, then `x = 2` and `y = -2.5`.

On the other hand, when `3x + 2y = 0` and `5x + 4y = 1`, then `x = -1` and `y = 1.5`.

apply() and its family: an introduction

We will introduce here a very important and useful function, namely **apply()**. This is the most straight forward function of a whole family of functions, including `lapply()`, `sapply()` and `tapply()` that you will certainly use if you continue programming with `R`, and are introduced later on in this document.

apply() allows producing results dimension-wise, let that be row- or column-wise. Some shortcut functions have been created for the most widely used **apply()** applications.

```
apply(M1,1,mean) # 1 is for rows
rowMeans(M1)
class(rowMeans(M1))

apply(M1,2,mean) # 2 is or columns
colMeans(M1)

apply(M1,1,sum)
rowSums(M1)

apply(M1,1,sd)
apply(M1,2,sd)
```

You may run any funtion you want to each of those dimensions. When you use more complex of customized functions, you need to define the function you want to run with **function()**.

```
apply(M1,1, function(x) mean(x)/sd(x))
apply(M1,2, function(x) mean(x)/sd(x))
```

We use `x` as we could use any other term.

```
apply(M1,1, function(rows) mean(rows)/sd(rows))
apply(M1,2, function(columns) mean(columns)/sd(columns))
```

In the example above, `x` is the most widely used notation in that kind of situation. Using `rows` and `columns` notations are used just to try better explain what is going on. In fact, we could have used whatever notation we wished.

Be careful with this (remember that `vectors` are indeed `column vectors`):

```
var.means <- colMeans(M1)
M1 - var.means # !!!
t(M1) - var.means
t(t(M1) - var.means)
```

2.3.4 Missing value actions

We will finish the matrices chapter with some lines about missing values, i.e. `NAs`. These short lines will not deal with the issue of missing value imputation, which is a vast statistical issue that deserves a training course on its own. We will be just having a quick look at how `R` deals with datasets as soon as it detects missing values.

```
M1.NAs <- M1
M1.NAs[4,1] <- NA
M1.NAs[2,4] <- NA
M1.NAs

apply(M1.NAs,1,mean)
apply(M1,1,mean)

apply(M1.NAs,2,mean)
apply(M1,2,mean)

na.omit(M1.NAs)
na.exclude(M1.NAs)
```

```
na.fail(M1.NAs)

apply(na.omit(M1.NAs),2,mean)
apply(na.omit(M1.NAs),1,mean)
```

Exercise 2

- Build a matrix M with the heights, the weights and the ages from the previous exercise as variables.

```
M <- matrix(c(heights, weights, ages), ncol = 3)
```

- Give names to the three variables in M.

```
colnames(M) <- c("height", "weight", "age")
```

- Build the same matrix, with variable names, with just one command.

```
M <- cbind(heights, weights, ages)
```

- What is the mean of each of those variable in the matrix?

```
colMeans(M)
apply(M, 2, mean)
```

- Can you display now the mean, the median and some other information regarding all variables in M with just one command?

```
summary(M)
```

- Could you calculate the standard deviation for those variables in the matrix?

```
apply(M,2,sd)
```

- Can you add the gender to the matrix? What happens when you do so?

```
toto <- cbind(M, gender.ch)
summary(toto)

tata <- cbind(M, gender)
tata

summary(M)
apply(M,2,mean)
```

- Calculate the mean height for those individuals heavier than 70 Kg

```
M <- cbind(heights, weights, ages)
M[M[,"weights"]>70,"heights"]
mean(M[M[,"weights"]>70,"heights"])
```

- Calculate the mean height and mean age for those individuals heavier than 70 Kg

```
M[M[,"weights"]>70,c("heights","ages")]
colMeans(M[M[,"weights"]>70,c("heights","ages")])
apply(M[M[,"weights"]>70,c("heights","ages")],2,mean)
```

2.4 *array*

`array` type takes matrix structure to n-dimensions, n being higher than 2. The object `H`, created might be represented as a 3-dimentional object (./array_3D.pdf)

```
E <- array(c(1:8, rep(1,8),seq(0,1,len=8)), dim = c(2,4,3))
help("array") # What entry data type requires the array() function?
class(E)
E[, , 1]
class(E[, , 1])
dim(E)
length(E)
nrow(E)
ncol(E)
E+10
H <- array(1:12,c(2,3,2)) # What does this do?
H
apply(H,1,mean) #*
apply(H,2,mean) #*
apply(H,3,mean) #*
```

Exercise 3

- What is going on on these three code lines?

```
apply(H,1,mean)
apply(H,2,mean)
apply(H,3,mean)
```

Answer:

On the first line, we calculate means by row. The elements of the first row are all those on the top horizontal slice of `H` (values: 1,3,5,7,9,11). Likewise, row 2 is the bottom slice of the array (values : 2,4,6,8,10,12).

On the second line means are produced by column, hence the resulting vector is of length 3, corresponding to the following vertical slices: left - [1,2,7,8], centre - [3,4,9,10] and right - [5,6,11,12].

On the third case, a vector of length 2 is obtained by calculating means on *front* (values: 1,2,3,4,5,6) and *back* (values: 7,8,9,10,11,12) vertical slices.

- Create a 4-dimension `array` and calculate the sum of all elements in each dimension.

```
H <- array(1 :24,c(2,3,2,2))
apply(H,1,sum)
apply(H,2,sum)
apply(H2,3,sum)
apply(H2,4,sum)
```

2.5 *list*

Objects of type `list` allow storing heterogenous data (numeric, character etc.) in a single, non-structured object. Being non-structured means, so that data stored in the `list` might be `vectors`, `matrix` etc., you may even a store a list `list` within a `list`, and so in endless recursive manner! You can visualize a `list` as a dresser cupboard where every drawer is different from the next one (see the figure below). What you put in each drawer will be up to you, both in size and in nature.

The `list` is a very important object type because of its flexibility. Indeed, many `R` function outcomes are lists. Also, you will be creating lists yourself very often when you use loops etc.

2.5.1 Subsetting `lists`

We may subset lists with `[]`, `[[]]` or `$`. If you subset with simple square brackets, in the same way as we use them with vectors, the result will be a smaller list. Using single square brackets will allow seeing one or several elements of a list at once, but not manipulating their contents. Using double square brackets you can subset only one of the components of the list but, on the other hand, you will break the `list` level of hierarchy and will gain access to the actual nature and content of the data in the chosen element, and you will be able to manipulate it.

The use of `$` is very handy, even recommended, when the components of the list are named. The output will be identical to the one you obtain by using with `[[]]`.

If we continue with the dresser analogy, `[]` will allow you to identify what is there in the drawer, but they won't let you to put your hand in. That is, you will not be able to subset any information within that drawer, nor make any math with that object in that drawer. However, you will be able to get the regarding more than one drawers within a single `[]`. When you use `[[]]` or `$` you will have access to the object in that drawer and you will be able to subset it, do maths, etc. On the other hand you will be forced to do so one element of the list at a time for every `[[]]` or `$`.

```
mylist <- list(matrix = M1,
          vector = 1:8,
          text = "toto",
          score = 8)

list(M1,1:8,"toto",8)

list(toto =
        list(toto =
               list(toto =
                      list(toto =
```

```
                          list())))))

mylist
names(mylist)
mylist[1]
colMeans(mylist[1])
mylist[[1]]
colMeans(mylist[[1]])
mylist[1:2]
mylist[c(1,4)]
mylist[1] + mylist[4]

mylist[[1]] + mylist[[4]]

mylist["matrix"]
mylist[["matrix"]]
mylist[[1]]
mylist$matrix
colMeans(mylist$matrix)
mylist$vector
mylist[[1]] + mylist[[4]] == mylist$matrix + mylist$score
mylist$v
mylist$sc
mylist[c("text","score")]
mylist[c("text","sc")] # It does not work because looking for, literally, "sc" element
mylist$matrix + mylist$sc
```

Those lines above indicate that the non-ambiguity does not apply under "". In this case, the exact string pattern needs to be matched.

As we said above, using `[[]]` or `$` to subset will give you access to the nature of the data stored in that element. This means also that you may then subset that element according to its nature. This is not the case if you use `[]`.

```
mylist[[1]]
mylist[[1]][1,]
mylist[[1]][1,][2]
mylist[[1]][1,2]



mylist$matrix[1,]
cos(mylist$sc) + mylist[[2]][3]
cos(mylist$sc) + mylist$v[3]
mylist[1]
mylist[1][1,]   # !!
length(mylist)  # This tells the number of elements in mylist, whatever those elements are.
length(mylist$vector)
```

You can easily add a new element to a list. You can do that in two ways, using `[[]]` or `$`. It is straight forward using the `$` if your list elements have names, as it is the case of our `x` list.

```
mylist
length(mylist)
names(mylist)
mylist$ten.to.one <- 10:1
```

```
mylist
names(mylist)
length(mylist)

mylist[["alphabet"]] <- letters
names(mylist)
length(mylist)
mylist
```

These final lines are meant to raise awareness about the use of commas, that is the function they are meant to modulate.

```
list(1,2,3,4,5)
list(c(1,2,3,4,5))
list(c(1,2,3,4,5),6)
list(1:5)
as.list(1:5) # !
list("toto",c(1,2,3,4,5))
```

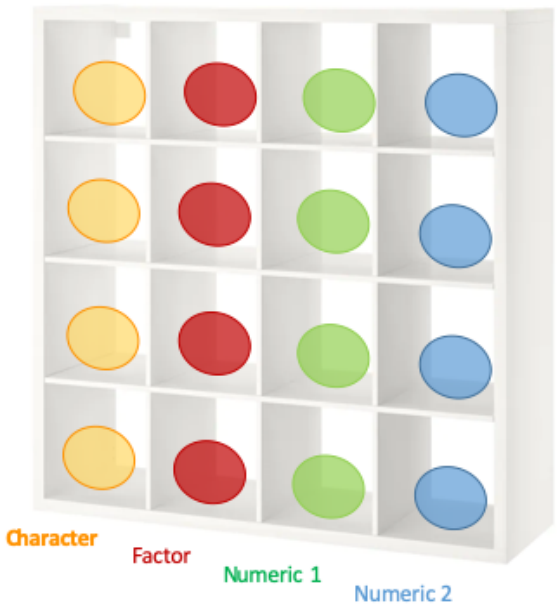
2.6 *data.frame*

`data frame` is a particular data structure, as well as an extremely important one. Indeed, `data frame` is the most widely used data structure when dealing with data analysis.

As it happens with the `matrix` structure, observations appear in rows and variables in columns. The big difference with a `matrix` is that a `data frame` may host, column-wise, heterogeneous data (numeric, character etc.). A helpful way to think about data frames is to visualize them as a group of vectors organized in columns, where every vector is a variable containing a distinct data type, which may differ from the data type in the other columns.

What makes `data frames` particular then, is the fact that they inherit the main advantages of two data structures we have already seen, namely `matrix` and `lists`. Indeed, `data frames` keep the well-structured shape of a `matrix` object, with rows and columns but, at the same time, `data frames` accept heterogeneous data as `lists` do. In other words, `data frame` is a singular `list` where every element is a `vector` of any kind and where all elements are of the same length, making it possible to arrange those `vector` elements in columns.

When we went through lists, we proposed to visualize them as cupboard dressers. We could use a similar analogy here. In this case we can visualize as a very widely known shelves (see figure here below), where you will store one object per pigeon hole. All objects on the same column will be of the same nature, but columns might independent in nature from one another.



In the following lines we will see how to create and deal with `data frames` using the **data.frame()** function. This is not the way you will be using most often to create `data frames`. In fact, most often you will not be creating data frames; instead, you will find `data frames` as a result of importing your datasets into `R`. Indeed, importing datasets into `R` will produce, in most cases, `data frame` objects. This is not the case, for instance, when you try to build a `data frame` out of existing `vectors` using tools such as **cbind()**, which renders a matrix.

```
set.seed(1)
height <- runif(20,150,180) # 20 random values between 150 and 180, taken from the uniform law

set.seed(1)
weight <- runif(20,50,90) # 20 random values between 50 and 90, taken from the uniform law

set.seed(1)
gender <- sample(c("M","F"), 20, replace = T)

set.seed(1)
eyes <- sample(c("blue","black","green","brown"), 20, rep = T)

class(height)
class(weight)
class(gender)
class(eyes)

mydf <- cbind(height,weight,gender,eyes)
class(mydf)
summary(mydf)
summary(mydf)
mydf
```

That is different from what we obtain using the **data.frame()** function, obviously. You may use the `stringsAsFactors` argument to state if you want character variables to appear as such (default) or as factors

(stringsAsFactors = TRUE)

```
mydf <- data.frame(height, weight, gender, eyes)
mydf
summary(mydf)
mydf <- data.frame(height, weight, gender, eyes, stringsAsFactors = TRUE)
mydf
summary(mydf)
```

2.6.1 Subsetting data frames

Because data frames share features with matrices and lists, we can subset them using [,], [], [[]], and $. The ways of implementing them, as well as their advantages and limitations, are (almost) the same that we have already seen. Let us see a few examples.

```
mydf[,1]
mydf[,"height"]
mydf[,grep("h", colnames(mydf))]
mydf[[1]]
mydf$height
mydf$h
```

There is a slight difference using [] with data frames as compared to lists: using [] with data frames enables data manipulation, which is not the case with lists.

```
mydf[1]
class(mydf[1])
mydf[1] + 2 # !!
mydf["height"] + 2
mydf[["height"]] + 2
mydf$height + 2

mylist[1]
mylist[1] + 2
mylist[[1]] + 2
```

You may obviously add variables (i.e. columns) to your data frame as you go

```
set.seed(1)
mydf$score <- factor(sample(1:4, 20, replace =TRUE))
mydf$BMI <- mydf$weight / (mydf$height/100)^2
summary(mydf)
```

The use of rownames is deprecated in data frames, and there is a good reason to be so. That said, you may feel more comfortable with them, especially when carrying out analysis such as trancriptomics etc. As we have seen with the matrix, you may use the function **rownames()** in order to create them. When you import a dataset, oyu will have he choice of doing so with or without rownames and, therefore, you will not need that function.

```
rownames(mydf) <- letters[1:20]
mydf
```

Subsetting rows of a data frame will be done in the same way as we do it with matrices, but the default outcome will be different in nature: subsetting even a single row from a data frame will still render a data frame as long as you pick more than one column, which was not the case with matrices. There is a good reason for that: the data in a row of a data frame is meant to be heterogenous and, therefore, vectorizing it would imply coercion.

```r
mydf[1,]
mydf["a",]
class(mydf["a",])
mydf["a",1:2]
mydf["a",1]
class(mydf["a",1])

M1[1,]
class(M1[1,])
```

We can of course combine rows and columns subsetting as we wish, as long as it complies with what we've just specified.

```r
mydf[1,3]
mydf[2:5,"weight"]
mydf$weight[2:5]
mydf$w[2:5]
mydf[["weight"]][2:5]
mydf["weight"][2:5] # !!

mydf[c(1,4),c(2,3)]
mydf[c("a","d"),c("weight","gender")]
mydf[c("d","a"),c("gender","weight")]
mydf[c("m","d"),c("eyes","height")]

sum(mydf$eyes == "green") # How many people do have green eyes.
mydf[mydf$eyes == "green", "weight"]
mydf[mydf$eyes == "green", "weight", drop = FALSE]
mydf[mydf$eyes == "green", c("weight","height")]
mydf[mydf$eyes == "green" | mydf$eyes == "brown", "weight"]
mydf[mydf$eyes == "green" | mydf$eyes == "brown", mydf$weight] ## !!
mydf[mydf$eyes == "green" | mydf$eyes == "brown",]$weight
mydf[mydf$eyes == "green" | mydf$eyes == "brown", "weight", drop = FALSE]
mydf[mydf$eyes == "green" | mydf$eyes == "brown", c("eyes", "weight")]
mydf[mydf$eyes == "green" & mydf$height > 170, c("gender", "weight")]
```

Some of those lines might be hard to code at once when you start with `R`. Do not hesitate to write commands ste by step should you wish it. Let us see an example. Let us say I want to know the weight and gender of those people taller than 170 cm with eye colour different from green.

```r
aux <- mydf[mydf$eyes != "green",]
aux <- aux[aux$height > 170,]
aux <- aux[, c("gender", "weight")]
# In one step:
mydf[mydf$eyes != "green" & mydf$height > 170, c("gender", "weight")]
```

2.6.2 head() and tail()

head() and **tail()** are two very useful functions that will show you, by default, the first or the last six elements of a data set. It can be used with `vectors`, `matrices`, `lists` or `data fames`. In the case of `matrices` and `data frames`, where those two functions are most useful, they will show the first or the last six rows. The number of shown elements can be modified.

```r
head(mydf)
tail(mydf)
```

```
head(mydf,2)
tail(mydf,1)
```

NOTE: You can also verify that nothing is missing from your dataset by checking its dimensions.

```
nrow(mydf) # The number of rows in your dataset
ncol(mydf) # The number of columns in yor datase
dim(mydf) # The dimensions of your dataset:
          # first the number of rows, then the number of columns
```

2.6.4 Contingency tables with table()

The **table()** function, which is used to create contingency or frequency tables between variables in a data frame is a very useful one.

```
table(mydf$gender)
table(mydf$eyes)
table(mydf$gender, mydf$eyes)
as.data.frame(table(mydf$gender, mydf$eyes))
summary(as.data.frame(table(mydf$gender, mydf$eyes)))
```

2.6.5 Combining two datasets with merge()

merge() combines two dataset, in the way `recherchev` might do it in excel, but reducing to zero the hassle and the risk of errors. We wiil crete/import two `data frames` into `R` that will help us understand how **merge()** works.

```
df1 <- data.frame (name = c("John", "Mary", "Laura", "Robert","Zoe"),
                   status = c ("married", "single", "married", "single", "married"),
                   weight = c(75,68,48,72,65))

df2 <- data.frame (name = c("Mary", "John", "Robert", "Laura", "Pat"),
                   status = c ("single", "married", "single", "married", "single"),
                   height = c(165,182,178,160,173))
```

or (we will be covering this later on)

```
df1 <- read.table("./Files/df1.txt", header = TRUE, sep = "\t")
df2 <- read.table("./Files/df2.txt", header = TRUE, sep = "\t")
```

or going through the *File -> Import Dataset -> From text* menu in `RStudio` (more about this later on).

When using **merge()**, by default: (1) it will look at all columns with the same name in both datasets and will keep observations with perfect matches for those columns in both datasets, and (2) it will duly arrange and display distinct information coming from either dataset and corresponding to those common observations.

```
df1
df2
merge(df1, df2)
merge(df2, df1)
```

Default features can be modified using the `all` and the `by` arguments.

Using `all = TRUE` will render all possible observations. This means that some observations regarding common columns in one dataset maight not be matched in the second one, and vice versa, thus generating missing values in the output

```
merge(df1, df2, all = TRUE)
merge(df2, df1, all = TRUE)
```

You can choose which columns should be taken into account for merging, among those sharing the same name between both datasets, instead of taking all of them by default. This might be useful when you have, for instance, two columns with the same name in either dataset, but displaying different information.

```
df3 <- df2
df3$status = c ("employed", "unemployed", "employed", "unemployed", "employed")
merge(df1,df3)
merge(df1,df3, by = "name")
merge(df1,df3, by = "name",all = TRUE)
```

The `by` argument might also be useful to detect errors in data sets.

```
df4 <- df2
df4$name <- df1$name[1:5] # Here I am inducing a mistake
df4
merge(df1, df4) # Error in the data sets?
merge(df1, df4, by = "name") # Inded, status do not match in df1 and df4
merge(df1, df4, all = TRUE)
```

2.6.6 Performing groupwise functions with aggregate()

The **aggregate()** cuts the `data frame` in distinct groups, allowing then to apply a function on one of the variables to each of those subgroups.

```
aggregate (df1$weight,
           by = list (married_single = df1$status),
           FUN = mean)
```

In the example above, we take `df1` and make subgroups according to levels in the status variable. Then we calculate the mean weight for each of those subgroups. The lines here below are meant to untangle the whole thing.

```
mean(df1$weight[df1$status == "married"])
mean(df1$weight[df1$status == "single"])
```

We can of course run more complex functions. In such cases, we need to go through the `function(x)` operator.

```
aggregate (df1$weight,
           by = list (married_single = df1$status),
           FUN = function(x) sd(x) / mean(x))
```

The `by` argument under **aggregate()** allows creating subgroups according to more than one variable. That is why the argument requires a list as input. We will see that with another example dataset. Please import the `genes.df.txt` dataset using `File -> Import Dataset -> From Text (Base)`, or running the following line:

```
genes.df <- read.table("./Files/genes.df.txt", header = TRUE, sep = "\t")
genes.df
```

We can calculate the mean of each gene grouped by `genotype` and `treatment`. In other words, we will be calculating the mean of all experimental replicates for every gene.

```
gene1.mean <- aggregate (genes.df$gene.1,
                         by = list (gtype = genes.df$genotype,
                                    treat = genes.df$treatment),
```

```
                          FUN = mean)

gene2.mean <- aggregate (genes.df$gene.2,
                          by = list (gtype = genes.df$genotype,
                                     treat = genes.df$treatment),
                          FUN = mean)

gene3.mean <- aggregate (genes.df$gene.3,
                          by = list (gtype = genes.df$genotype,
                                     treat = genes.df$treatment),
                          FUN = mean)

gene1.mean
gene2.mean
gene3.mean
```

Now, you could merge all results into a single `dataframe`. The only problem is that all columns in all three gene mean results display the same number, so that if you run

```
merge(gene1.mean, gene2.mean, gene3.mean)
```

you will get no results.

There are at least two ways to get it sorted. One of them is to merge according to `gtype` and `treat`, but not `x`. The other issue is that **merge()** does not accept more than two `data frames` as an input, so that you will need two consecutive mergers in order to get all three genes in.

```
first.merge <- merge(gene1.mean, gene2.mean, by = c("gtype", "treat"))
first.merge
final.merge <- merge(first.merge, gene3.mean, by = c("gtype", "treat"))
```

In that `final.merge`, `x.x`, `x.y` and `x` are your mean values for `gene.1`, `gene.2` and `gene.3`, respectively. All you have to do now then is give those columns proper names.

```
colnames(final.merge)[3:5] <- c("gene.1","gene.2","gene.3")
```

Of course, things are trickier when you have many more than three variables, but let us stick with that example by now.

Exercise 4

- We have seen how to build a matrix containing the variables heights, weights, ages and gender, and we've seen what happens because of coercion. What can we do so that what is numeric remains numeric?

```
df <- data.frame(height = heights,
                 weight = weights,
                 age = ages,
                 gender = gender)
summary(df)
```

- Add a variable "name" indluding the following information. `c("Pierre","Nathalie","Morgane","Enzo","Antoine", "Jean-Claude","Rachel","Agathe","Marie","Thomas")`

```
df$name <- c("Pierre","Nathalie","Morgane","Enzo","Antoine","Jean-Claude",
             "Rachel","Agathe","Marie","Thomas")
```

- Please arrange the variables so that the names appear first, then the gender, the age and then the rest.

```r
df[,c(5,1:4)]
df <- df[,c("name","gender","age","height","weight")]
# or
df <- df[,c(5:1)]

# df[,c(ncol(df),1:(ncol(df)-1))]
```

- What are the weights of those people taller than 175 cm?

```r
df[df$height > 175,"weight"]
# or
df$weight[df$height > 175]
```

- Can we have the names of those people as well?

```r
df[df$height > 175,c("name","weight")]
```

- Build a contingency table telling how many people taller, or shorter, than 175 cm there are per gender.

```r
table(df$gender, df$height > 175)
as.data.frame(table(df$gender, df$height > 175))
```

- Extract the height of male individuals who are lighter than 75 kg. It is possible to do so with a single code line (see `&` operator).

```r
df[df$weight<75 & df$gender=="M",5]
df[df$gender=="M" & df$weight < 75,"height"]
df$height[df$gender=="M" & df$weight < 75]

toto <- df[df$gender=="M",] # Here I filter male indiv.
toto <- toto[toto$weight < 75,] # Here I filter male indiv lighter than 75Kg
toto <- toto[,"height"] # Here I pick on the heights of those filtered individuals
```

- `iris` is a data frame containing anatomical features of petals and sepals in three different plant species. Calculate the mean surface of petals and sepals by species, where surface is calculated by multiplying width and Length. Then create a data.frame, if possible within a single comand line, with Species names, Petal Surface and Sepal Surface.

```r
iris$Petal.Surface <- iris$Petal.Length * iris$Petal.Width
iris$Sepal.Surface <- iris$Sepal.Length * iris$Sepal.Width

toto <- aggregate (iris$Petal.Surface, by=list (Sp=iris$Species), FUN=mean)
tata <- aggregate (iris$Sepal.Surface, by=list (Sp=iris$Species), FUN=mean)
colnames(toto)[2] <- "petal.m"
colnames(tata)[2] <- "sepal.m"
merge(toto,tata)
# or
iris.surface <- merge(aggregate (iris$Petal.Surface,
                                 by=list (Sp = iris$Species),
                                 FUN = mean),
                      aggregate (iris$Sepal.Surface,
                                 by=list (Sp = iris$Species),
                                 FUN = mean),
                      by = "Sp")
colnames(iris.surface)[-1] <- c("petal.m", "sepal.m")
```

- Explain the names of those variables in the data frame you just obteined. Change “x.x” and “x.y” by

"Petal.Surface" and "Sepal.Surface".

```
colnames(iris.surface)[2:3] <- c("Petal.Surface","Sepal.Surface")
```

Chapter 3: Data Import / Export

3.1 Dataset import

We tend to use always the same code line when we import a dataset into R. We may use **read.table()** function, which is a very old-fashioned function, but still does the job of rendering a `data frame`. Arguments such as `row.names` and `header` allow taking into account (or not) rownames and headers. Arguments such as `sep` and `dec` allow specifying the nature of column separators and decimal notation.

```
import_A <- read.table("./Files/import.training.txt",
                       row.names = 1, header = TRUE, sep = "\t")
import_B <- read.table("./Files/import.training.txt",
                       header = TRUE, sep = "\t")
import_C <- read.table("./Files/import.training.txt",
                       header = TRUE, sep = "\t", dec = ",")

head(import_A)
head(import_B)
head(import_C)

class(import_A)
class(import_B)
class(import_C)

summary(import_A)
summary(import_B)
summary(import_C)

str(import_A)
str(import_B)
str(import_C)
```

Functions such as **read.csv()** and **read.csv2()** are just particular cases of that original **read.table()** function. The function **fread()** from the `data.table` package offers automatic detection of header, separators and decimal notation with a very time-efficient performance as well.

All that said, *R Studio* proposes a very handy and efficient menu to import data in R under '*File > Import Dataset*, which automatically detects decimal notation, separator etc. Please play around with the three *import.training* files and see what happens when you execute the import at the prompt.

3.2 Dataset export

We will cover now data export from R, as far as `data frames` or `matrix` are concerned (we will deal later with figure exports). Remarkably enough, `RStudio` does not offer at the export the same menu it offers at the import. Thus, functions such as **write.table()**, **write.csv()** and **write.csv2** may be used. They are quite similar to the analogous **read...()** functions we have already seen, but a couple of arguments differ a little: `row.names` is a logical argument when exporting and numerical when importing, and `col.names` replaces `header` at export.

When you export the data set there are two things that you should consider, similarly to what you do when importing datasets, namely: what do you want to do with the rows / columns structure and what do you

want to do with the separator and decimal notations. You usually will want to keep the same row / column structure you had under R, and will adapt the sep / dec to your own taste.

Let us take the import_A that you have just created with **read.table()** and put it into **write.table()** in order to export it. Beware that, unless you state it otherwise, every exported file will appear in the current working directory.

```
?write.table
write.table(import_A, "export_A.xls",
            row.names = TRUE, col.names = NA, sep = "\t") # Why col.names = NA?
write.table(import_A, "export_AA.xls",
            row.names = TRUE, col.names = NA, dec = ",", sep = "\t")
write.table(import_B, "export_B.xls",
            row.names = FALSE, col.names = TRUE, sep = "\t")
```

Please, play around with the arguments of that function. Try **write.csv()** and **write.csv2()** as well. Finally, the **sink()** function allows deriving everything from the console to a text file until you close it using dev.off().

```
A <- seq(1,10,l=50)
write.table(A,"A.txt")
sink("Abis.txt")
A
summary(A)
B <- c(1:5)
B
sink()
```

A very useful function to export data is the **fwrite()** from the data.table package. This function works faster and does not need separator parameter when you write a csv file.

```
install.packages("data.table")
data.table::fwrite(import.training, file = "exported_A.txt", sep = "\t")
data.table::fwrite(import_A, file = "exported_Ar.txt", sep = "\t", row.names = TRUE)
data.table::fwrite(import_A, file = "exported_A.csv")
```

Chapter 4: Programming

4.1 Loops

Loops in R are largely implemented with **for()** or **while()** functions (though one must be very careful with the latter cause it may render never ending loops if you do not pay enough attention to your coding). Loops are very often combined with conditional requests, which are implemented with **if()** and **else()** functions or with the condensed function, **ifelse()**.

It should be noted that, if you are new to coding, loops in R seem to be as painstaking to implement as unavoidable. With time, you will realize that none of that is true: you will first understand that loops are actually very easy to grasp and, then, you will start thinking about most efficient ways to code in order to get rid of them altogether because they might be really time consuming.

Loop removal might be in some cases a trivial affaire. In some other cases, however, loop removal will require deeper coding knowledge. Indeed, functions such as **apply()**, **lapply()**, **sapply()**or **tapply()** are meant to do the job in a much more time-efficient manner than loops. You already know **apply()**. You will learn about the others if you continue using R and this is why we introduce them in the 4.2 section.

These examples here below show simple examples of for loops, with or without the conditional postulates if and else. Bear in mind that, if the loop contains more than one command, then you will need {} to open and close the loop. Otherwise, only the first command will be executed. RStudio gives you a good hint

about it with line indentation: if the code line you are writing does not fall at the indentation site you are expecting, there must be a good reason for it.

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

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

for (i in 1:10)
  print(1:i)
print(i/2)

for (i in 1:10){
  print(1:i)
  print(i/2)
}

for (i in 1:10)
  print(1:i); print(i/2)

for (i in seq(10,1,-2))
  print(1:i)

for (i in c(1,25,49,5,8))
  print(i+1)
```

The following lines are meant to tell how to prime an empty data structure, that you be progressively filling up with every loop iteration.

```
set.seed(1)
alphabet <- sample(1:26)
names(alphabet) <- letters

vocals <- c()
for (i in c("a","e","i","o","u"))
  vocals <- c(vocals, alphabet[i])

# As opposed to:
for (i in c("a","e","i","o","u"))
  vocals <- c(alphabet[i])
```

The following line follows the same reasoning, but takes the whole thing a little bit further. We create a meaningless list where the important thing is to have as many elements in it as iterations we will produce with our loop. In the example below, we will run an anova (produced with **aov()**) for every gene in our `genes.df` dataset. We will store the results of each anova in a drawer of our "dresser", i.e. in an element of our list. We will store those results in an iterative manner, so that the *ith* element in our list stores the result of the *ith* iteration of our loop. The tricky thing here is: we have to by storing the results obtained with the 4th column in our dataset, i.e. our first gene, in the 1st element of our list.

```
anova.results <- list(1,2,3)
for(i in 4:6)
  anova.results[[i-3]] <- summary(aov(genes.df[,i] ~ genotype * treatment, data = genes.df))
# Or:
for(i in 1:3)
```

```
  anova.results[[i]] <- summary(aov(genes.df[,i+3] ~ genotype * treatment, data = genes.df))
# Or:
aov.loop <- list(1,2,3)
names(aov.loop) <- colnames(genes.df)[grep("gene",colnames(genes.df))]
for (i in 1:length(aov.loop))
  aov.loop[[i]] <- summary(aov(genes.df[,names(aov.loop)[i]] ~ genotype*treatment,
                               data = genes.df))
```

4.2 Conditional requests

Conditional requests are implemented with **if()** and **else()** functions, or with the condensed function, **ifelse()**. Conditional requests will produce a given outcome when the leading boolean question is `TRUE`, and a different one when the answer is `FALSE`.

We'll have a look at a couple of simple examples here below:

```
x <- rnorm(100) # 100 random numbers from the Normal distribution, where mean = 0.
ifelse(x > 0, "yes", "no")

x <- abs(rnorm(100, mean = 0.05))
ifelse(x < 0.05, "signif", "not signif")
```

Quite often, **if()** and **else()** functions appear embedded in **for()** loops.

```
y <- 0
z <- 0
compil.x <- c()
for (i in 1:50) {
  x <- sample(1:10,1)
  if (x > 5) y <- y+1
  else z <- z+1
  compil.x <- c(compil.x, x)
}

y
z

x <-sample(1:10, replace = TRUE)
v <- 0
for (i in 1:10)
 v <- ifelse (x[i] < 5, v+1, v)
```

4.3 Alternatives to loops

Loops are important when you code, and it is OK if you use them under `R`. As you make progress with your coding, you will learn how to implement alternatives to loops. As we have already mentioned, functions such as **apply()**, **lapply()**, **sapply()**or **tapply()** are meant to provide loop alternatives. We will introduce them in this section. Do not get frustrated if you do not grasp all of it at first. Indeed, they are as powerful as they are subtle, and you may need some time before you fully understand the way they work.

All *whatever***apply()** functions comprise at least two parameters, that is `whateverapply(X, FUN, ...)`. The `X` object will be somehow the input reference and `FUN` will be the function that will be implemented.

4.3.1 apply()

We have already gone through **apply()**. This function is meant to avoid loops. See the following two examples.

```
iris
head(iris)
class(iris)
mydata <- iris[,-5]
mydata
```

Let us use a loop to calculate medians and the square root of the mean (don't ask why) for each species.

```
medians <- c()
sqrt_means <- c()
for (i in 1:ncol(mydata)){
  medians = c(medians, median(mydata[[i]]))
  sqrt_means = c(sqrt_means, sqrt(mean(mydata[[i]])))
}
medians
sqrt_means
```

We can avoid that nonsensical loop using **apply()**.

```
medians <- apply(mydata,2,median)
sqrt_means <- apply(mydata,2, function(x) sqrt(mean(x)))
medians
sqrt_means
```

4.3.2 lapply()

According to `?lapply`, "**lapply()** returns a *list* of the same length as `X`, each element of which being the result of applying `FUN` to the corresponding element of it". Bear in mind that even though the outcome of **lapply()** will always be a list, `X` does not need to be a list.

Thus, look at these three lines, producing the same results.

```
lapply(1:10, function(x) x*2)
lapply(matrix(1:10,ncol = 2), function(x) x*2)
lapply(as.list(1:10), function(x) x*2)
```

Because...

```
length(1:10)
length(matrix(1:10,ncol = 2))
length(as.list(1:10))
```

All that said, `X` *can* be a list

```
mylist <- list(8,
               1:10,
               c(4,7,25),
               matrix(1:30, ncol = 5))
mylist*2
mean(mylist)
lapply(mylist, function(x) x*2)
lapply(mylist, mean)
lapply(1:length(mylist), mean) #!!
lapply(1:length(mylist), function(x) mean(mylist[[x]]))
```

```
mylist2 <- mylist
mylist2[[5]] <- c("Hello, my name is not Donald")
lapply(mylist2, function(x) if (is.numeric(x)) x*2 else x)
lapply(mylist2, function(x) ifelse (is.numeric(x), x*2, x)) ## !!!
```

4.3.3 sapply()

sapply() works in a similar fashion as **lapply()**, but it will simplify the result outcome whenever that is possible, which is not always the case, as we can see in the following example.

```
lapply(mylist, function(x) x*2)
sapply(mylist, function(x) x*2)

lapply(mylist, mean)
sapply(mylist, mean)

sapply(mylist2, function(x) if (is.numeric(x)) mean(x) else x)
sapply(mylist2, function(x) if (is.numeric(x)) mean(x) else x, simplify = FALSE)
lapply(mylist2, function(x) if (is.numeric(x)) mean(x) else x)
```

If the simplify argument is set to `FALSE`, the outcome will present the same structure as a list. If the argument is set to the default `TRUE`, the outcome will be wrapped into a `vector` whenever that is possible.

4.3.4 tapply()

tapply() computes a function on a vector according to the levels of a factor. It is a very powerful and useful function that may let you avoid not only loops, but also more complex functions such as, for instance, **aggregate()**.

```
iris
tapply(iris$Sepal.Length, iris$Species, median)
```

The alternative to that, using a loop would have been as follows:

```
sp.medians <- list(1,2,3)
for (i in levels(iris$Species))
  sp.medians[[i]] <- median(iris[iris$Species == i,"Sepal.Length"])
```

And the alternative using **aggregate()**

```
aggregate(iris$Sepal.Length, by = list(Sp = iris$Species), FUN = median)
```

Please note the the nature of those thre possible outputs is different: `vector`, `list` and `data frame`, respectively.

Here below, another couple examples on the use of **tapply()**.

```
tapply(iris$Sepal.Length, iris$Species,  function (x) x*2)
tapply(iris$Sepal.Length, iris$Species,  function (x) x/mean(iris$Sepal.Length))
```

You should also be aware of the use of the **do.call()** function, which will allow you performs a given "function" *within* the object you propose as the second argument.

```
myresult <- tapply(iris$Sepal.Length, iris$Species,  function (x) x/mean(iris$Sepal.Length))
do.call("cbind",myresult)
```

Exercise 5 * What do you think of these lines here below

```
a <- c()
b <- c(30,45,60,90)
for (i in 1:length(b)) a[i] <- cos(b[i])
```

Answer: That loop is actually pointless

```
  a <- cos(b)
```

- Obtain the equivalent to y and z here below, without any loop.

```
y<-0
z<-0
for (i in 1:10) {
  x<-runif(1)
  if (x>0.5) y=y+1
  else z<-z+1
}
y
z
```

Answer

```
x <- runif(10)
y <- sum(x>0.5)
z  <- 10-y
```

- On the example here below, remove the `for` loops, first on `j`, then on `i`.

```
M <- matrix(1:20, nr = 5, nc = 4)
res = rep(0,5)
for (i in 1:5){
  tmp <- 0
  for (j in 1:4)
    tmp <- tmp + M[i,j]
  res[i] <- tmp
}
```

Answer : `j` loop removal

```
for (i in 1:5) res[i]=sum(M[i,])
```

`i` loop removal

```
res <- apply(M,1,sum)
# ou
res <- rowSums(M)
```

4.4 Functions

We have been using many functions. Some of them, such as **lst()**, **c()**, **exp()** or **log()** are included within the base `R`. Furthermore, you will be loading hundreds of new functions as you install and load new packages. On top of that, you might need to be able to develop and code your own functions, created by yourself, and this is what we will be talking about in this chapter.

Every function in `R`, wherever it comes from, is defined by its *name* and *arguments*. The arguments of a function might be *mandatory* or *optional*. As for the latter, a default value is proposed when the function is coded. It should be noted that, for a function to work, it is not mandatory to have mandatory arguments.

The function called **function()** allows creating a function:

```
MyFunction <- function (x) {x+2}
ls()
MyFunction
MyFunction()
MyFunction(3)
MyFunction(5)
x <- MyFunction(4)
x
MyFunction.bis <- function (x = 3) {x+2}
MyFunction.bis()
MyFunction.bis(12)
```

Regarding optional arguments, please note the difference between `Function2`, `Function3` and `Function4` here below:

```
Function2 <- function (a, b=0) {a+b}
Function2 (2,3)
Function2 (5)
Function2 (565)
Function2 (b = 5)
Function2 (5,5)

Function3 <- function (a, b = a) {a+b}
Function3 (2,3)
Function3 (5)
Function3 (5,0)

Function4 <- function (a,b) {a/b}
Function4 (6,2)
Function4 (2)
Function4 (b = 3)
Function4 (b = 3, a = 27)
Function4 (3, 27)
```

The outcome of **function()** will correspond to the last line in the source code of function. See an example here below, and please note the use of a `list` to render the results of the so called *Calcule()* function.

```
Calcule <- function (r) {
  p <- 2*pi*r
  a <- pi*r*r
  list (radius = r,
        perimeter = p,
        area = a)}
res <- Calcule (3)
res
res$rad
a # !!
p # !!
```

The last line in the function determines the outcome of the function.

```
Calcule.toto <- function (r) {
  p <- 2*pi*r
  a <- pi*r*r
  list (radius = r,
        perimeter = p,
```

```
        area = a)
  print("toto")
  }
Calcule.toto()
Calcule.toto(5)
```

Let us work out another example.

```
multiply_xy <- function(a, b, c) {
  data.frame(mult = a * b,
             cat.size = ifelse(a * b < c, "small", "big"))
}
multiply_xy(iris$Sepal.Length, iris$Sepal.Width, 20)

apply(iris[,c(2:4)], 2, function(x) multiply_xy (iris$Sepal.Length, x, 20))
do.call("cbind",apply(iris[,c(2:4)], 2, function(x) multiply_xy (iris$Sepal.Length, x, 20)))
```

Of course, you may use loops within functions, so that the outcome will depend, for instance, on one or many conditional requirements. Let us see that with an example.

```
covid_guideline <- function (fever, headache, cough, diarrhea) {
  fever_38 = fever >= 38
  if (sum(fever_38, headache, cough, diarrhea) <= 2) {
    return ("Stay home and see how it evolves. Let us know if you notice further symptoms")
  } else {
    return ("You must be tested for the COVID and remain confined")
  }
}

covid_guideline(39, 0, 1, 1)
covid_guideline(37.5, 0, 1, 1)
covid_guideline(37.5, FALSE, TRUE, TRUE)
covid_guideline(37.5, FALSE, TRUE, "un peu")
```

Exercise 6

- Is it mandatory to create a `list` to render the results of `Calcule()` ?

Answer: All resulting 3 elements being numeric, a simple vector could suffice.

```
Calcule <- function (r) {
  p <- 2*pi*r
  a <- pi*r*r
  c (r, p, a)}
resultat <- Calcule (3)
resultat
```

- Write a function to calculate the area of a rectangle from side lengths l1 and l2. The function must equally return the length and the width of the rectangle.

```
rectangle <- function(l1,l2){
 p = (l1+l2)*2
 a = l1*l2
 list(width = min(l1,l2), length = max(l1,l2), perimeter = p, area = a)}
```

Using `rectangle()`:

```
rectangle(4,6)
rectangle(6,4)
res <- rectangle(8,7)
```

- What if we decided to make the l2 argument in the *rectangle()* function optional, with a default value of l2=l1? What are the implications?

```
rectangle.bis <- function(l1,l2=l1){
 p = (l1+l2)*2
 a = l1*l2
 list(width = min(l1,l2), length = max(l1,l2), perimeter = p, area = a)}
rectangle.bis(6)
```

- Write a function to calculate the n first elements of the Fibonacci sequence ($u_1 = 0; u_2 = 1; u_n = u_{n-1} + u_{n-2}, n > 2$)

```
fibo <- function(n){
  res=rep(0,n)
  res[1]=0
  res[2]=1
  for (i in 3:n) res[i]=res[i-1]+res[i-2]
  res}
```

- Use that function to calculate the ratio between two consecutive terms in that sequence. Represent that ratio against the number of elements for n = 20. What do you see?

```
# The ratio between two consecutive terms of the Fibonacci sequence tends towards
# the golden ratio (~1.618034)
res <- fibo(20)
ratio <- res[2:20]/res[1:19]
plot(1:19, ratio, type="b")
```

- Could you please explain the outcome of the code chunk below?

```
Calcule <- function (r) {
   p <- 2*pi*r
   a <- pi*r*r
   m <- list (radius = r, perimeter = p, area = a)
   toto <- letters[r]
   return(list(m,toto))}
Calcule(3)
```

Answer: The function only returns the product of its last line. In this case, the alphabet. And that is regardless the r parameter.

Chapter 5: Graphics

In this chapter we will focus on some basic graphic features in `R`. Graphics are one of the strong points in `R`, allowing endless possibilities. Moreover, in the last few years there have been great efforts from the community to build strong and rich resources that provide reproducible, elegant and meaningful graphical outputs. One of the packages devoted to graphics is `ggplot2`, which has become sort of the golden standard in `R` graphics. This package is part of what its main developer, Hadley Wickham, calls `tidyverse`, a series of packages or libraries dealing with data treatment sharing the same grammar and logic. We will be covering `tidyverse` in another course that you may consider as a second part to this one.

Here we will be covering basic graphic features because, being beginners as you are supposed to be, we encourage you to learn producing richer `R` graphics outputs straight from `ggplot2`. This chapter will allow

you to produce quick, relevant graphics on-the-go. In any case, you will grasp the importance of the `plot` keyword in `R` by running `help.search("plot")`.

5.1 Discrete data

Functions such as **pie()** and **barplot()**, as is happens with most graphic funtions in `R`, have a very large number of argument to modify the appearance of the resulting plot.

```
vec <- c(12,10,7,13,26,16,4,12)
pie(vec)
```

```
pie(vec,clockwise = T)
```

```
names(vec) <- LETTERS[1:8]
pie(vec)
```

```
barplot(vec)
```

```
vec2 <- vec*2
plot(vec,vec2)
```

```
plot(vec,vec2, col = "red")
```

```
plot(vec,vec2, col = 2)
```

```
plot(vec,vec2, col = 3)
```

```
colors()
plot(vec,vec2, col = "wheat3")
```

```
plot(vec,vec2, col = 1:7)
```

```
plot(vec,vec2, pch = 17)
```

```
plot(vec,vec2, pch = 15)
```

```
help(par)
```

The `par(mfrow=c(x,y))` command splits the graphics window into x rows and y columns and includes the upcoming graphics row by row.

```
par(mfrow=c(2,2))
help(par)
pie(vec)
barplot(vec)
plot(vec,vec2)
dev.off()
```

You may came back to a single-graph layout by executing `dev.off()` or `par(mfrow=c(1,1))`

```
par(mfrow=c(1,1))
barplot(vec,col=1:8)
```

```
barplot(vec)
```

```
barplot(vec,col=3)
```

```
barplot(vec,col=rep(c(2,4),4))
```

```r
barplot(vec,col=rep(c(2,4),each = 4))
```

```r
barplot(vec, col = "white", border = rep(c(2,4),4))
```

```r
dotchart(vec)
```

```r
par(bg="lightgrey")
dotchart(vec,pch=16,col=1:8)
```

```r
dotchart(vec,pch=21,col=1:8)
```

```r
dotchart(vec,pch=15,col=1:8)
```

```r
par(bg="white")
```

5.2 Quantitative data

```r
x <- rnorm(50) # Data simulation from a normal law, ?rnorm for more information
```

```r
boxplot(x)
```

```r
hist(x)
```

```r
barplot(x)
```

```r
stripchart(x)
```

```r
barplot(x)
```

```r
iris
class(iris)
summary(iris)
boxplot(iris)
```

```r
boxplot(iris, las = 2)
```

```r
boxplot(iris$Sepal.Length)
```

```r
boxplot(Sepal.Length ~ Species, data = iris)
```

```r
boxplot(Sepal.Length ~ Species, data = iris, las = 2)
```

```r
boxplot(Sepal.Length ~ Species, data = iris)
```

```r
boxplot(Sepal.Length ~ Species, data = iris, col = 2:4)
```

```r
boxplot(Sepal.Length ~ Species, data = iris, border = 2:4, col = "white")
```

5.3 Plotting two variables on x and y

In order two plot two variables on x and y, respectively, **plot(var1, var2)** suffices. We will see here how to do so, and also how to add extra features to your plot, regardless the function you used to produce, let that be **plot()**, **boxplot()**, **hist()** etc.

```r
x<-seq(-10,10,l=50)
plot(x,sin(x))
```

```
plot(x,sin(x),type="l")
abline(v=0,col="blue",lwd=5,lty=3)
abline(h=sin(0.7),col=3)
text(-5,-0.5,"whatever",font=3)
lines(c(-10,10),c(-1,1),col=2)
```

```
help(abline)
help(lines)
```

The graphical options are listed in the *Graphical Parameters* section of `help(par)`. The arguments `main,xlab, ylab, sub...`, are used to place the legends of the axes and the graph.

```
par(mfrow=c(1,2))
plot(x,sin(x),type="l",col=1,main="sinus")
plot(x,cos(x),type="b",col=3,xlab="Abscisses")
```

5.4 Export of graphics

In `RStudio`, you can use the *Export* tab of the graphic window to export your graphic. However, for the shake of traceability we would suggest you use the functions associated with saving graphics files: **bmp()**, **jpeg()**, **png()**, **pdf()**, **postscript()**. The procedure to follow is as follows:

1. Create a graphics file to which the graphics output is redirected
2. Draw the graph: the graph does not appear on the screen.
3. Close the file. Don't forget this step! The graph output will then return to the screen for the next plot.

Those functions will allow you determine figure size and resolution. Also, bear in mind that , whereas `png` or `jpeg` files, for example, will only accept one page per file, `pdf` format will accept as many pages as you feed in before closing the file by executing, as always, `dev.off()`.

```
png("myboxplot.png")
boxplot(Sepal.Length ~ Species, data = iris, col = 2:4)
dev.off()

pdf("mygraphs.pdf")
plot(1:100)
text(20,80,"abcdef")
pie(vec)
dev.off()

pdf("iris-boxes.pdf")
for (i in 1:4)
  boxplot(iris[[i]] ~ iris$Species, col = 2:4)
dev.off()
```

Exercise 7

- Plot the heights vs the weights from the data.frame example. Then make the points be dark blue squares.

```
plot(df$height, df$weight, col = 4, pch = 15)
plot(df$height, df$weight, col = 4, pch = 21)
plot(df$height, df$weight, col = 4, pch = 17)
dev.off()
```

- Make two plots on the same page using the iris data, showing (1) the histogram of petal length of the *setosa* species and (2) the boxplot of sepal length of the virginica species colored in green.

```
png("toto.png")
par(mfrow = c(1,2))
hist(iris[iris$Species == "setosa", "Petal.Length"])
boxplot(iris[iris$Species == "setosa", "Sepal.Length"], col = 3)
dev.off()
```

- Change the label of the x axis on both plots and put the corresponding Species names.

```
par(mfrow = c(1,2))
hist(iris[iris$Species == "setosa", "Petal.Length"], xlab = "Setosa", ylab = "whatever")
boxplot(iris[iris$Species == "setosa", "Sepal.Length"], col = 3, xlab = "Virginica")
dev.off()
```

Chapter 6: (A very little of) Statistics

The prerequisites for this training course stated that no experience on statistics was required. This chapter will give you arguments to judge whether we are breaking our promise or not. In any case, we should be reminded that, according to the `r-project` website : *R is 'GNU S', a freely available language and environment for* ***statistical*** *computing and graphics which provides a wide variety of statistical and graphical techniques : ...*

But there is no need to run. The following lines will provide an extremely light insight into `R` power on statistics. The statistical power of R is awesome and we do not intend to explore even an infinitesimal part of it. We will just try to demystify the prompt and give a few arguments as to why menu-driven softwares might be less convenient than one might think.

Lastly, we will put some emphasis on some aspects regarding inferential statistics, somehow neglecting equally important topics such as descriptive statistics. If we have reached this chapter during the course, it is because you have done fine. However, it is certain that we will not have much time left and, therefore, we have taken the decision here of focusing on what might already be familiar to people not trained in statistics.

6.1 Inferential statistics

We will first create a simulated dataset. Then we will run the most widely known statistical test: the `t` tests or `Student` test.

```
test.data <- data.frame(x = rnorm(100), y = rnorm(100, mean=1))
t.test(test.data$x,test.data$y)
t.test(test.data$x,test.data$y, paired = TRUE, var.equal = TRUE)
```

We can store the result outcome in an object.

```
my.outcome = t.test(test.data$x,test.data$y)
my.outcome
class(my.outcome)
```

Description of *htest* object: This class of objects is returned by functions that perform hypothesis tests (e.g., **t.test()**, **wilcoxon.test()** etc.), that contain information about the null and alternative hypotheses, the estimated distribution parameters, the test statistic, the p-value, and (optionally) confidence intervals for distribution parameters.

```
names(my.outcome)
my.outcome$p.value
my.outcome$parameter
```

Why are the degrees of freedom not a whole number? The **t.test()** function assumes by default that the variances of both samples are different (`var.equal = FALSE`). This makes sense because unequal variances make for a more conservative option that the opposite. Truth is, when variances are different, we cannot

properly talk about `t` or `Student` test, but of `Welch` test, which is the one that is being performed in this case with the **t.test()** function.

Anyhow, the equality of variances can also be tested, for instance with the `Fisher test for the equality of variances`.

```
var.test(test.data$x, test.data$y)
var.test(test.data$x, test.data$y)$p.value
```

The result of the test does not allow us to accept the alternative hypothesis, i.e. that both variances are not equal. Hence, we can modify the variance parameter when running the `t.test`.

```
t.test(test.data$x, test.data$y, var.equal=T)
```

Nullity test of the correlation coefficient: We can also easily test if both variables are correlated.

```
cor.test(test.data$x, test.data$y)
```

!! We can also test the normality of the data, for instance with the `Kolomogorov-Smirnov` test:

```
ks.test(test.data$x, test.data$y)
```

Does it make sense to test the normality of the whole dataset, all variables confounded?

```
?ks.test
ks.test(test.data$x,"pnorm")
ks.test(test.data$y,"pnorm")
mean(test.data$y)
ks.test(test.data$y,"pnorm",1)
```

I agree that the use of this last argument is not very clear for the non-initiated.

Exercise 8

- Test of Normality: Apply the `Shapiro-Wilk` test instead of the `Kolmogorov-Smirnov` test. Look for the right function to do so.

```
help.search("shapiro")
```

The help search tells us that the function that allows us to implement the test is the `shapiro.test()` function. To see how to use the function and to test the normality of the `x` and `y` variables of the `test.data` dataset.

```
help(shapiro.test)
shapiro.test(test.data$x)
shapiro.test(test.data$y)
```

- Test the correlation between `x` and `y` using the Spearman correlation coefficient.

```
?cor.test
cor.test(test.data$x, test.data$y)
cor.test(test.data$x, test.data$y, method="spearman")
```

6.2 Descriptive statistics and some graphics

Here we return in part to the graphs for quantitative data discussed in previous chapters.

```
x <- runif(100)
y <- runif(100)
mean(x);var(x);sd(x)
min(x);max(x)
```

```
?quantile
quantile(x);median(x)
quantile(x,0.5)
quantile(x,0.9)
```

The **boxplot()** and **hist()** functions may not produce a graph (option `plot=FALSE`).

The **stem()** function produces a *stem-and-leaf* diagram (stem and leaf) which gives a more "rustic" view of the distribution of data than a histogram, though the information it gives might be highly explanatory.

The **hist()** function provides options to change the appearance of the histogram.

```
boxplot(x)
```

```
hist(x)
```

```
boxplot(x,plot=FALSE)
cor(x,y)
cor(x,y,method="spearman")
stem(x)
stem(y)
x[25]=2
res=boxplot(x)
```

```
res
hist(x)
```

```
x[25]=runif(1)
hist(x,density=10)
```

```
hist(x,plot=FALSE)
hist(x,nclass=5)
```

Exercise 9

- Calculate some statistical indicators for a sample of length 1000 drawn randomly according to a normal distribution of mean 10 and variance 25. Perform some basic graphs with it.

```
x2 <- rnorm(1000,mean=10,sd=5)
summary(x2);boxplot(x2);hist(x2)
```

- Do the same with the following vector

```
x3 <- c(rnorm(500,5,1),rnorm(500,10,1))
```

- What is the particularity of this new vector, `x3`?

Answer:

```
summary(x3)
boxplot(x3)
```

```
hist(x3)
```

The distribution of the `x3` vector clearly shows a bi-modality. It is visible on the histogram, but not on the boxplot.

6.3 Linear regression and ANOVA

Student test compares the means of two samples, or just of one sample compared to an expected mean. When more samples are involved and/or when different factors must be considered (genotype and treatment, for

instance), then it is likely we will perform a regression and an ANOVA.

```
head(cars)
cars.lm <- lm(dist ~ speed, data=cars)
summary(cars.lm)
```

Let us draw all that.

```
plot(cars)
abline(cars.lm)
```

```
plot(cars, ylim = c(-20, 120), xlim = c(0,25))
abline(cars.lm)
```

That summary gives a lot of information and the following paragraph will tell about it. We will focus on some of the indicators on the Coefficients part.

The first column on the Coefficients matrix tells us about the `Estimates` of the linear regression, that is, the intercept and the slope of the model: `y = -17.58 + 3.93*x`, which is the same as `dist = -17.58 + 3.93*speed`. The units of speed and dist being, respectively, miles per hour and feet, the slope of the line, i.e. 3.93, tells us that for every m/h speed is increased, the distance to stop de the car increases in 3.93 feet. The `Std. Error` columns tells about the robustness of the afore mentioned `Estimates`. The calculation of that Std Error is a rather complex one to be explained here (or anywhere, to tell the truth). The `t value` quantifies how many standard errors our coefficient estimate is far away from 0, and it is calculated by multiplying the `Esimate` by the `Std. Error`. The higher it is, the further the `Estimate` is away from 0. And lastly, `Pr` is the Probability of getting such a (o more extreme) `t value`. Low `Pr` equals to low p-value, equals low probability of your `Estimate` being zero. The last three lines in the Summary are highly explanatory. The `Multiple R-squared` number tells us that 65% percentage of the total variability in the distance might be explained by the speed. That is, speed does not explain everything, but it explains a big deal nonetheless. In fact, that last line, the one reflecting on the `F-statistic` and its p-value tells us how well speed predicts distance. The low p-value indicates that speed explains distance much better that the null model, that is the one where speed des not play any influence on distance, that is the mean value of distance. That `F statistic` and its p-value is what you obtain when you run an ANOVA.

```
anova(lm(dist ~ speed, data=cars))
aov(dist ~ speed, data=cars)
summary(aov(dist ~ speed, data=cars))
```

Exercise 10

- In the light of the following image, could you please interpret what the code below tells.

```
Call:
lm(formula = Petal.Length ~ Species, data = iris)

Residuals:
   Min     1Q Median     3Q    Max
-1.260 -0.258  0.038  0.240  1.348

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)        1.46200    0.06086   24.02   <2e-16 ***
Speciesversicolor  2.79800    0.08607   32.51   <2e-16 ***
Speciesvirginica   4.09000    0.08607   47.52   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4303 on 147 degrees of freedom
Multiple R-squared:  0.9414,    Adjusted R-squared:  0.9406
F-statistic:  1180 on 2 and 147 DF,  p-value: < 2.2e-16
```

```r
summary(lm(Petal.Length~Species, data = iris))
anova(lm(Petal.Length~Species, data = iris))
```

Answer:

R chooses in alphabetical order the Species which will play the role of Intercept, i.e. *setosa*. Choosing in alphabetical order does not change the results and can be circumvented. We see that, according to the model, for every unit the *setosa* petal length increases, the versicolor petal increases 2.798 units and *virginica*'s 4.09. All Estimates associated with every species are highly significant. Besides, Species alone explains 94% of the variability in Petal Length. Finally, as expected by all we have just outlined, the ANOVA tells us that the Species effect is highly significant when talking about petal length.

Chapter 7: Tidy data

In the last few years, a bunch of `R` developers have put a lot of emphasis on data tidiness and graphic syntax and grammar. The main developer in this current is Hadley Wickham, who has created a whole bunch of libraries, now grouped under the `tidyverse` name, that have greatly modified the way we code and make graphics in `R` nowadays. This last mini chapter is not intended to explain in detail what `tidyverse` is about. We will just propose to have a look at a tidy COVID dataset, which is ready for tidy graphics on ggplot2.

```r
library(tidycovid19)
remotes::install_github("joachim-gassen/tidycovid19")
library(tidycovid19)
library(zoo)
library(knitr)

merged <- download_merged_data()
```

```r
df <- tidycovid19_variable_definitions %>%
  select(var_name, var_def)
kable(df) %>% kableExtra::kable_styling()

merged

mydata <- merged %>%
  filter(country == "United States") %>%
  fill(total_tests) %>%
  mutate(
    new_cases = confirmed - lag(confirmed),
    ave_new_cases = rollmean(new_cases, 7, na.pad=TRUE, align="right"),
    new_tests = total_tests - lag(total_tests)
  ) %>%
  filter(!is.na(new_cases), !is.na(ave_new_cases))

ggplot(mydata, aes(x = date)) +
  geom_bar(aes(y = new_cases), stat = "identity", fill = "lightblue") +
  geom_line(aes(y = ave_new_cases), color ="red") +
  geom_line(data = mydata %>%
              filter(!is.na(new_tests),new_tests != 0), aes(y = new_tests), color ="blue") +
  theme_minimal()
```