BeginR

The Old Testament

David Rengel

March 2024

Chapter 1: Introduction

1.1 Why 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.
```

Why use `R`:

- `R` is **free**.
- `R` is very popular and has become the golden **standard** for data analysis and visualization in many research areas, both in private and public domains (the figure here 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**.

1.2 First steps in RStudio

`RStudio` is an interface for `R` in order to make it friendlier. 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 probably find three panels. The main one, on the left, is `the console`. This is where all commands will be executed. In a research analogy, `the console` is the bench or the field, that is where you carry out your experiments or collect your data. 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 research analogy, you need a lab-book. A place where you will write your protocols, the ones that you will have with you at the bench so that you know what you will be doing. When you code, that lab-book is called the `source` or the `code`. You may open your source window in `RStudio` by clicking on the double-screen icon on the right side tof 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. 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`, or with `Ctrl+S`. The colour of the tab containing the code name will change colour once it has 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, your default wd should be *"/Users/yourusername"*. The `wd` is the folder in which all saved objects will be located, unless you tell otherwise.

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, `R` runs 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 can 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 can navigate using `"./whatever"` to go to or to create the sub-folder called `whatever` within your current directory. Likewise, you can use `"../"` to go one step higher in the folder architecture from you current directory.

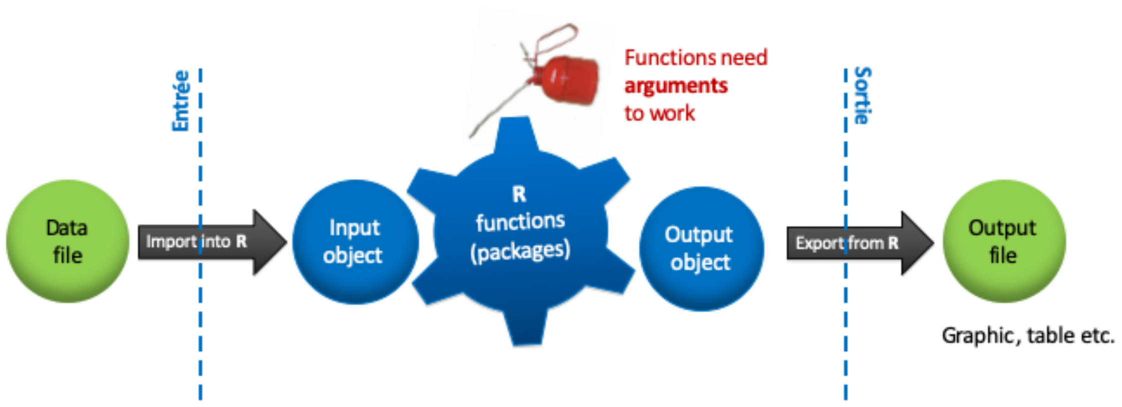
1.4 `R` Projects

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 within the folder. Importantly, if you need to be working on two, or more, projects at the same time, it will make the whole thing 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.

In order to create a new project with RStudio: `File > New Project > New Directory (or Existing Directory) > New Project`. Then you give a name to the project. A new RStudio window will open while, at the same time, the new project has been created in your chosen folder. When you click on that project, an RStudio will open and the working directory will automatically be the one where the project is located in.

1.5 `R workspace` or working environment

When you work with `R`, you will most certainly find yourselves somewhere along the path shown in the image 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 youra = `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")`. You can then **load()** that object as we have seen above.

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 will 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 Help

`UseRs` of all levels should 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 submited 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.

```
help(plot)
?plot
```

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.

While you are working with `R`, you may know that among the packages that you have loaded there is a function whose name contains a certain term you are interested in, but you do not remember the whole name of the function, for instance. Or maybe you want to know about all functions with that term among all loaded packages. The function **apropos("yourterm")** renders all functions from loaded packages *containing* the specified term. It does not provide help about them, it just produces a list with the names of those functions. The results appear on the console, and not in the help tab.

```
apropos("plot")
```

You may remember, or just have a vague memory about a function, whose package of origin you do not remember. Or you want to find help of any function `related` to a given term. In this case, you will use **help.search("yourterm")** or **??yourterm**.

By running these lines here below, we get a list of functions that deal with the term `plot` among all packages already *installed* on your machine, whether the package has been loaded or not. We will see the installation and loading steps further below.

```
help.search("plot")
??plot
```

Internet is an extremely 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", it will most likely it 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 internet *forum*. There are very active 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 (more about this in the next chapter). 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.

A recent addition to this helping resources is ChatGPT and similar AI tools. Tap a specific question and you will very likely get an answer, including code chunks on the matter. It is not always perfect: sometimes you may get outdated information, but it is worth trying. You may also ask how do something in two different ways. For instance: "How do I create a boxplot using either base R or ggplot2?". Paste that question onto ChatGPT and see how it goes.

1.8 Using libraries (aka packages)

Libraries or *packages* are groups of functions and/or datasets developed within the R community, allowing to accomplish specific tasks. There are three main repositories where all those packages are uploaded and curated. It all started with the *Comprehensive R Archive Network*, or *CRAN*, which, to this date, remains the largest `R` repository. The first package was submitted on the 15th March 2006 and, since then, the number of featured packages has grown exponentially, reaching 20317 packages in February 2024 (see the figure below).

And those are "just" the packages under CRAN. `Bioconductor` (bioconductor.org) is a second repository, devoted to biology-related libraries, such as `limma`, `edgeR` and `DESeq2`. `Bioconductor` curates, as of today (February 2024), 2266 packages. The third repository, `GitHub` curates, somehow, though not necessarily so, state of the art, cutting edge libraries. In fact, `GitHUb` is much more than just an `R` repository, but this is beyond the point here.

Using libraries in `R` requires two steps: *installing* and *loading*. The way you install packages differs depending on the repository you are calling upon. You will use **install.packages()** at the prompt to install packages from CRAN. See the example here below. Quotes to name the package being installed are mandatory:

```
install.packages("alluvial")
```

The installation of packages featured in `Bioconductor` repository will be carried out using the installation code provided by Bioconductor itself, which will be visible on their site every time you will try download one of their libraries. See the example below with the *limma* package.

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

There are several ways to install packages from `GitHub`, the third repository we have mentioned. A common one is using the **install_github()** function. To use it, you will need to install and load the *remotes* package first (more about loading just below).

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

Installing libraries means that those packages exist now in your computer. You can use **library()** to get a list of the packages that have been already installed in your computer. This list also exists in the `Packages` tab in the `RStudio` window usually located bottom-right.

```
library()
```

Wherever the package comes from, bear in mind that:

- Installation is done only once, even though it is advisable to periodically update the packages by running **update.packages()**.
- Installing the package does not mean you can use it straight away in your session. You need to load it first. Loading a package, e.g. `mylibrary`, is performed running **library(mylibrary)**. Look at the example here below. Quotes are optional.

```
library(alluvial)
library("alluvial")
```

If you need to use a given library, You will need to load it every time you launch `R` or `RStudio`. If you click on the case besides the name of the library in the `Packages`tab in `RStudio`, you will see in the console that **library(mypackage)** is executed. However, this will not leave any trace on your code, and this is why we do advice you not to do it, since it is always a good thing to keep a trace of everything you have done to run your code.

We can use **search()** to verify that the package is loaded and, hence, its functions ready to be used.

```
search()
```

The last loaded package will usually be at position 2 under **search()**. If you load a new package, you will find the previously loaded one in position 3, and so on. You can use **ls()**, and change its position argument, which by default corresponds to `pos = 1`, to list all the functions available in the package located at that position.

```
# List available functions in the first loaded package.
ls(pos = 2)
# The one loaded before that one
ls(pos = 3)
# And so on.
```

If you use **ls()** with empty brackets, you are looking by default at `position = 1`. That position corresponds to *.GlobalEnv*, that is, your workspace, and it will never change position. Using **ls()** will allow you see the objects you have created in your workspace. Remember: these objects do not exist physically in your computer, and will be lost if you close your session without saving them (more of this later on).

A variant of **ls()** is **ls.str()**, which not only lists the objects within the given position, but tells also about the nature of each of those listed objects.

```
ls()
ls.str()
ls.str(pos = 2)
ls.str(pos = 3)
```

The order in which libraries are loaded is important. Indeed, if two functions with the same name are available in two different packages, only the one coming from the last loaded package will be valid when you call for that function name. It is extremely likely you will find yourselves in this situation if you continue to code with `R`. These conflicts are easy to circumvent by adding the name of the package before the name of the function, like this: *package::function*.

Chapter 2: First notions at the prompt

2.1 Basic notions about operators and functions.

We will start manipulating `R`, first as a calculator.

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

`R` becomes interesting when we start using *functions*. Quite a lot of functions are included in default packages in `R`, being ready to be used as you open your session. Others will belong in specific libraries that you will need install and load, as we have already seen above. Finally, functions might also be created by you; as we will see further down in the course.

Any function, wherever it comes from, will be called by its name, followed by `()` brackets comprising all necessary `arguments` for the suitable execution of that function.

Arguments will allow functions to work, and will allow us tuning those functions to obtain the optimal result. Importantly, arguments of a function can be `mandatory` or `optional`. Mandatory arguments have to be explicitly implemented by the user because there is no default value for them within the function. As for the optional arguments, `R` proposes a default value for them; however, the default value for such an argument might be modified when running the function. In order to know what the arguments of a function are, and whether those are mandatory or optional, you just need to reach for the help of the function.

```
log(2)
?log
exp(1)
log(exp(1))
log(2, base = 2)
log(50, base = 2)
log2(50)
?log2
log2(50, base = 10)
log2(50, base = 2)
```

We store the results in variables. To affect variables, we may use three operators producing identical results, i.e. `->`, `<-` et `=`. Same-name variable erases the previous one.

```
n = 15
n <- 40
60 -> n
n -> 60 !!
```

The reason to use `->` or `<-` rather than `=` are both "philosophical" and "practical". As for the "philosophical" reasons: `2+2 = 4` is an absolute, unequivocal truth; on the contrary, `n <- 40` is a choice since we could have called it whatever and `n` could have taken any value we want. As for the practical reasons: `->` or `<-` will allow you to distinguish what you choose to do from what you are obliged to do. Indeed, when you are using functions, arguments will be asked and you will be using `=` because that is the way arguments are asked. It will be really helpful as you read your code, and therefore a sign of good practice, to use `->` or `<-` to give names to the variables and objects you have created.

Remember that `R` is case sensitive.

```
x <- 1
X <- 10
x
X
```

You may create the object and produce the results at the same time, though in practice you won't be using this that often.

```
(X <- 25) # Creates the object and gives results at the same time
```

You may obviously use the created variables to operate. Be aware that when you use a name for a variable that has already been given, `R` will erase the previous content under that name without asking about it.

```
y <- x*6
y

rm(x)
x
y

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

Can you tell the result of the lines here below?

```
a + b / a + b # ?
(a + b) / (a + b)
```

2.2 Listing and removing

We have already mentioned **ls()**, which allows listing `R` objects. We will have a deeper look into that function. We will also mention here **rm()**, which allows removing `R` objects from your workspace. You will be using these two variables quite often. We will also see some regular expression operators that will be extremely helpful when coding.

We will create a few more variables to fully grasp what those functions do. Some of these new variables will be `character` variables. String characters in `R` are defined by `""`. More about this soon enough on this document, devoted to `vectors`.

```r
b <- 5
b2 <- b*2
acdc <- "back in black"
cure <- "Friday I'm in love"
motorhead <- "overkill"
nirvana <- "nevermind"

ls()
ls(pattern = "a")
ls(pat = "a")
ls(pat = "^a")
ls(pat = "a$")
ls(pat = "^a|a$")
ls(pat = "^a | a$")

ls(pat = "b|a")
ls(pat = "^b|^a")
ls(pat = "^c|d$")
```

We remove objects from the workspace using **rm()**.

```r
rm(a)
ls()
```

Notice that we have removed just the `a` object, not every object containing an `a`. We can tune the function to remove every object in the workspace containing a given pattern.

```r
rm(list = ls(pat = "b"))
ls()
```

We can remove all variables in the workspace.

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

The **ls()** is not to be mistaken with **list.files()**, which lists the files in your working directory or, if the suitable path is stated, in any other folder in your computer. You can use **file.remove()** to remove physical files from the computer. Careful, it will not ask for confirmation.

```r
list.files()
list.files(path = "../")
file.remove("crap.R")
```

Chapter 3: 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`

	Homogeneous data	Heterogeneous data
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 of length one in `R`. Indeed, all variables we have created so far are, in fact, vectors of length one.

3.1 *Vectors*

3.1.1 Using c(), or not, to create `numeric`, `character` and `logical` vectors.

All variables we have defined in the introductory chapter above are, indeed, `vectors`. `Vectors` are one-dimensional objects and they represent the simplest way to store values in a single object. To create vectors of length higher than from scratch, usually we use the **c()** function, where `c` stands for `combine`.

There are four main vector types: `integer` and `double` for numbers (both are `numeric`), `character` and `logical`. 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. 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.

Let us start by seeing how **c()** works. You may use **length()** to know how many elements your vector has.

```r
c(2,3,5,8,4,6)
a <- c(57,3)
a
```

You can of course use **c()** to combine previously existing vectors.

```r
a <- c(2,3,7)
b <- c(5,8,4)
c <- c(6,7)
abc <- c(a,b,c)
abc
```

Use **length()** to know how many elements your vector has.

```r
length(a)
```

There is a shortcut of when you want to create vectors of length > 1. You can use a colon, i.e. `:`, when you want to create a vector of step 1 (or -1) between two numbers.

```r
c(1:5)
1:5
5:1
(-10):5
5:(-5)
```

We have already mentioned string `character` vectors. Remember that string characters in `R` go between quotes.

```r
e <- c("this","vector","has","five","elements")
length(e)
f <- c("this, vector, has, one, element")
length(f)
```

`Factors` are a particular type of `character` vector, used where categorical levels are required. `Factors` are used indeed to define categorical variables in statistical analysis and are, hence, extremely important in `R`.

Let us take the following variable, where L, M and H stand respectively for `Low`, `Medium`, and `High`

```
dose_char <- c("L","H","M","H","L","M","H")
summary(dose_char)
```

The summary of that variable does not tell us how many times each term in the variable appears. That issue, which is not trivial, can be addressed by simply defining our variable as a `factor`, using th **factor()** function.

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

When we define a factor, by default, we define its `levels`. You can access those with **levels()**.

```
levels(dose_fct)
```

Notice that **levels()** produces a character `vector` where the levels of the factor do not appear, by default, in the order they appear in the factor, but in alphanumerical order. However, this can be modified using the `levels` argument within the **factor()** function or, alternatively, using the **relevel()** function if we want to redefine just the reference level.

```
dose_fct <- factor(dose_char, levels = c("L","M","H"))
dose_fct
summary(dose_fct)
relevel(dose_fct, ref = "L")
```

The order in which factor levels are defined is important in many scenarios. For instance, when you run an ANOVA in `R`. Imagine that your explaining variable is the dose. The ANOVA will read the `levels` in the order they have been defined, and will take the first one as the references level. By default, therefore, the reference level will be `H`, and not `L`. Please note that this default behaviour will not affect the significance of your analysis as such. In other words, it won't change the p-value or the amplitude of the effect. However, it will have an influence on the sign of the fold-changes etc., which might be disturbing.

Another property of `factors` is that they might be, quantitatively speaking, ordered, which is important in some cases. The logical parameter `ordered` in the **factor()** function allows you specify whether the factor levels are indeed ordered. Again, the order follows alphanumerical logic unless you modify it with the `levels` argument, in which case the order follows what you state under that argument.

```
dose_fct
min(dose_fct) # !! Look at the error message.

dose_fct <- factor(dose_char,
                   levels = c("L","M","H"),
                   ordered = TRUE)
dose_fct
min(dose_fct)
```

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

```
num <- c(4,2,3,1)
min(num)

fct <- factor(num)
fct
min(fct)
sum(fct)

fct <- factor(num, ordered = TRUE)
min(fct)
```

```
sum(fct)
```

`Logical` vectors are very important in `R`. This `vectors` are boolean-type, comprising only two possible elements: `TRUE` or `FALSE`. `Logical vectors` are, most often, the product of boolean operators such as `==` (equal to), `!=` (different from), `<` (smaller than), `>` (bigger than), `<=` (smaller or equal than), or `>=` (bigger or equal than). See the example below.

```
2 == 3
2 != 3
2 < 3
2 > 3
"a" == "b"
"a" < "b"
```

The result of each of those boolean operations above is a `logical` vector of length one. See here below another example of `logical vector`.

```
pvs <- c(0.7, 0.3, 0.02, 0.2, 0.05)
pvs <= 0.05
```

Another boolean operator that renders `logical vectors`, and that you will be using very often is `%in%`. It allows you to ask whether members of a variable are among members of another variable. You will be using it a lot to show data regarding certain individuals of your choice. The length of the resulting logical vector is the same as the variable you are asking about, the one in front of the operator.

```
c("a","c","t") %in% c("w","a","z","t")
c("w","a","z","t") %in% c("a","c","t")
```

If you add the `!` operator, which means `not`, you would be asking for elements of the first variable that ar `not in` the second variable. Like this:

```
!c("a","c","t") %in% c("w","a","z","t")
!c("w","a","z","t") %in% c("a","c","t")
```

You realize that the `!` operator just reverses the TRUE / FALSE responses: if an element of the first variable `is not` in the second one, then `TRUE` is produced.

One function related to logical operators is **which()**, used in various situations where you need to identify the positions of elements satisfying a specific condition in a variable or a dataset. While the logical vector itself is crucial for evaluating conditions, **which()** complements it by providing the locations where those conditions are met, which can be essential for subsequent analysis or data manipulation.

```
which(c("a","c","t") %in% c("w","a","z","t"))
which(pvs <= 0.05)

v <- c(8, 5, 4, 2, 1, 7, 1)
which(v != 5)
v >= 5 & v <= 7
which(v >= 5 & v <= 7)
v < 2 | v > 4
which(v < 2 | v > 4)
```

Two related functions are **which.min()** and **which.max()**.

```
which.max(v)
which.min(v) ## !!
```

We will clarify the use of **which()** down in the document, because it is true that you will see it very often, even when it is not necessary to use it.

3.1.2 Attributes

Any object in `R` may have attributes. Attributes do not change the information conveyed by the object itself, they just give some additional, non-mandatory information about it. The most widely used attribute are the `names` for the elements in that object. In a two-dimensional object, we may have `rownames` and `colnames`; in the case of a vector, we may have just `names`.

```
v <- 1:4
names(v) <- c("Enzo","Louise","Emma","Layla")
v
names(v)
```

Alternatively, you may create the `vector`, with its names, as you go.

```
v <- c(Enzo = 1, Louise = 2, Emma = 3, Layla = 4)
v
names(v)
```

Obviously enough, if you do any math with a `vector`, or any data structure for that matter, the attributes of that data structure, that is the names in our example, will remain unchanged.

```
v <- v*2
v
```

3.1.3 Time to work

1 - Create two vectors `x` and `y` of length 5, perform `vector` addition, and store the result in `z`

```
x <- c(1, 2, 3, 4, 5)
y <- c(6, 7, 8, 9, 10)
z <- x + y
```

2 - Give names to `z` elements

```
names(z) <- letters[1:5]
```

3 - Create a logical `vector` asking if the numbers in `z` are among those in the vector `2:5`

```
z %in% 2:5
```

3.1.4 How to create random, repetitive, and sequential vectors.

There are a bunch of functions that produce random variables with numbers from a given probability distribution. These functions are very useful when you are simulating data, verifying methods or imagining graphics, for instance. All those functions start with an `r`, for random, and then, somehow, the name of the distribution. For instance, **rnorm()** will produce numbers out of the normal distribution, and then there are **runif()**, **rexp()**, **rbeta()**, **rbinom()**, etc.

As for the Normal distribution, default parameters are `mean = 0` and `sd = 1`, but those default parameters can be modified. By default, whatever the distribution we are targeting, the bigger the sample, the closer you will be to the parameters of the chosen distribution.

```
?rnorm
r <- rnorm(10)
mean(r)
sd(r)

r <- rnorm(1000)
mean(r)
sd(r)
```

```
r <- rnorm(10, mean = 5, sd = 3)
mean(r)
sd(r)

r <- rnorm(1000, mean = 5, sd = 3)
mean(r)
sd(r)
```

The function **sample()** will let you randomize pre-defined vectors. When using this function, it is crucial to understand the role of the `replace` argument. See below:

```
?sample
sample(1:10)
sample(1:10, replace = TRUE)

sample(1:10, 7)
sample(1:10, 7, replace = TRUE)
```

`replace = TRUE` becomes mandatory if the size of the required sample exceeds the size of the population.

```
sample(1:2, 5)
sample(1:2, 5, replace = TRUE)

sample(c("WT","KO"), 20, replace = TRUE)
```

The **unique()** function allows us to see which are the elements that appear at least once in a variable.

```
x <- sample(1:10, replace = TRUE)
x
unique(x)
```

The boolean question here below allows you to see if there are indeed duplicated elements. This can be very useful when you are verifying datasets etc.

```
length(x) == length(unique(x))
```

seq() creates vectors with a sequential pattern.

```
seq (from = 1, to = 20, by = 2)
```

We will use those arguments under **seq()** to learn two things about how arguments are to be used under any given function:

- Arguments must be implemented **in the order** they are mentioned in the function (`?myfunction` to know what the order is), **only** if I do not name the arguments. In other words, if I name the arguments within the function, I can put them down in the order that suits me best.
- Arguments might be partially named, as long as no redundancy is observed between arguments.

Regarding the first point above:

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

seq (from = 1, to = 20, by = 5)
seq (1,20,5)
seq (20,1,5)
seq (by = 5, to = 20, from = 1)
seq (5, to = 20, from = 1)
# But:
seq (to = 20, 1, 5)
seq (to = 20, 5, 1)
```

As for the non-redundancy and partial mention to the arguments, all lines below produce the same result.

```
seq (to = 20, by = 5, from = 1)
seq (t = 20, by = 5, from = 1)
seq (t = 20, b = 5, f = 1)
seq (t = 20, b = 5, 1)
```

There is another way of using **seq()**, and that is using the `length.out` argument instead of `by` (both arguments are mutually exclusive).

As we've seen above, `by` defines the amplitude of every step in the sequence. Thus, the number stated under `to` will not appear in the result if the last step to be taken goes beyond that number. The `length.out` argument works in, somehow, the opposite way: both numbers under `from` and `to` will appear in the result. In this case, **seq()** will accommodate the amplitude of each step to the required final outcome length.

See the example here below:

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

The use of both `by` and `length.out` at the same time are incompatible. To distinguish how `by` and `length.out` work, you may imagine how you are going to slice a cake: are you going to slice the cake, whatever its size, according to a fixed size per portion so that the last slice will be certainly smaller, and therefore discarded? That'd be the `by` way. Or are you going to take into account the size of the cake, how many slices should be taken out of it, and then accommodate the slice size? That'd be the `length.out` way.

Finally, the **rep()** function creates vectors with a repetitive pattern. Here, the key argument are `times` and `each`.

```
?rep

rep(1:5, times = 3)
rep(1:5, each = 2)
rep(1:5, times = 3, each = 2)
rep(1:5, 3)

rep (5,10)
rep (10,5)
rep (c(1,2), 3)
rep (c(1,2),each = 3)

rep (c("wt","ko"), 2)
rep (c("wt","ko"), each = 2)
rep (c("wt","ko"), 2, each = 3)
```

It is advised to always mention explicitly the `each` argument in order to avoid confusion with the `length.out` argument.

```
rep (c("wt","ko"), 2, each = 3)
rep (c("wt","ko"), 2, 3)
```

Why?

```
?rep
rep (c("wt","ko"), length.out = 3)

rep(c("wt","ko"), times = 2)
rep(c("wt","ko"), length.out = 4)
```

3.1.5 Time to work

1 - Create a random vector of 100 elements taken from the uniform distribution, where `min = 20` and `max = 50`. Use `?runif`.

```
runif(100, min = 20, max = 50)
```

2 - Create a `vector` called `cards`, in which you put four 1s, four 2s etc up to four 10s. Then:

- Shuffle `cards`
- Take five random cards: Do you have a poker winning hand?
- Take a card 30 times and see if you get six aces.

```
cards <- rep(1:10, each = 4)
sample(cards)
sample(cards, 5)
set.seed(555)
aces <- sample(cards, 30, replace = TRUE)
sum(aces == 1)
```

3 - Create a `vector` with every multiple of 5 up to 100

```
seq(5, 100, by = 5)
```

3.1.6 The class, type and attributes of your vector.

class() and **typeof()** will give you, respectively, the `class` and the `type` of the object you put between the brackets. You will be using these quite often, especially when you are a begineR and do not feel confident as for what you are dealing with. Those two functions may be used with any data structure, but here we will focus on `vectors`. The class and type of a vector may be redundant, but remember: `numeric` vectors can be `double` or `integer` (more about these two later on).

```
e <- c("this","vector","has","four", "elements")
class(e)
typeof(e)
```

A for `numeric` vectors, things might be a little confusing, since they might be `double` or `integer`. Do not worry about it: most of the times you will only need to know if there are numbers involved.

```
class(1:10)
typeof(1:10)
n <- 25
class(n)
typeof(n)
```

Furthermore, there is a bunch of functions starting with `is.`, that allows you asking whether your vector (or any data structure you ask for) is of a precise nature. The answer to any of those questions will be, in itself, a `logical` vector of length 1 stating `TRUE` or `FALSE`. Let us play with those a little.

```
is.numeric(1:10)
is.double(1:10)
```

```r
is.integer(1:10)
is.character(1:10)
is.logical(1:10)

e
is.character(e)
is.numeric(e)
is.logical(is.numeric(e))

en <- is.numeric(e)
is.logical(en)

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

The lines above tell us that it is not a good idea to create variables that are named `T` or `F`. The lines here below tell you why is that.

```r
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 )
```

There is a similar set of functions starting with `as.`, similar to those starting with `is.`, which will allow you changing the class or type of the vector, depending on what your analysis requires.

```r
n
as.character(n)
n + 50
as.character(n) + 50

e
as.factor(e)
as.numeric(e)
```

As we have already said, `numeric` vectors can be of two types, `doubles` or `integers`, the former being the default type. The difference between both will be, most of the times, irrelevant to your daily work. However, bear in mind that `integers` are precise and `doubles` are approximations, which may lead to confusing issues if you are not aware of it. Have a look at the following lines:

```r
is.double(2)
as.integer(2)
is.double(2.1)
as.integer(2.1)
as.integer(2.9)

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

```r
sqrt(2)^2
sqrt(2)^2 == 2
as.integer(sqrt(2)^2) == 2
sqrt(2)^2 - 2

20/3
20/3 == 6.666667
```

3.1.7 Coercion.

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 heterogeneous data. In this situation, `R` will take in the data but the receiving object will be coerced to be converted to the type of data that is the most flexible one given all data inputs.

The more possible values a data type can host, the more flexible it is. The following data types are increasingly flexible: `logical`, `numeric`, and `character`.

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

c(1,FALSE)
c(1,TRUE)
c(1,"a")
c("a",FALSE)
```

Coercion may be a funny issue when resolving boolean operators.

```r
1 == TRUE
-1 < FALSE

"1" == 1
"1" < 2
```

The coercion of `logical` vectors to `numeric` is something you will be using to your own advantage all the time. It is very handy when you need to count occurrences, for instance.

```r
sum(c("a","c","t") %in% c("a", "t", "w", "z"))

v
sum(v != 6)
sum(v >= 2 & v <= 4)
sum(v < 2 | v > 4)
sum(v < 2 & v > 4)
```

Imagine we have a list of p-values and we want to know if they are smaller than 0.05.

```r
pvalues <- c(0.2, 0.012, 0.03, 0.94)
pvalues < 0.05
sum(pvalues < 0.05)
```

Imagine now that I have some fold changes associated to those p-values. I want to consider just those fold changes only if the associated p-value is smaller than 0.05.

```
fcs <- c(2, 9, -3, 1)
fcs * (pvalues < 0.05)
```

There will be more examples of this further down in the document.

3.1.8 Vector subsetting

In this chapter we will introduce the second kind of brackets that are used in `R`, i.e. square brackets `[]`. These are used to subset or extract information from objects in `R`. We will see here how they work in vectors, that is the simplest objects in `R`. We will also mention here a couple of functions you are not aware of yet, but do not worry, we will cover them later in more detail.

```
x <- sample(1:26) *45
names(x) <- sample(letters)
x
```

We can subset vectors according to elements' index, that is according to their position in the vector.

```
x[2]
x[2:4]
x[c(2,4)]
x[2,4] # !!!
x[4:2]
```

We can use negative selections

```
x[-2]
x[-c(2,3,4)]
x[-c(2:4)]
x[-(2:4)]
x[-2:4] # !!!
x
```

We can subset vectors according to elements' names instead.

```
x["a"]
x[c("f", "a", "c" )]
```

A third way to sub-setting is according to a boolean question. Elements for which the boolean question stands true, will be selected.

```
x > 360
x[x > 360]
x[x <= 360]
x[x == 360]
x[x != 360]
```

We can also use double conditions.

```
x[x > 300 & x < 500]
sum(x > 300 & x < 500)
```

The following line gives an error. Why is that?

```
x[x < 300 & x > 500]
```

In those cases, `or` should be the way to reason, not `and`.

```r
x[x < 300 | x > 500]
```

We use **any()** and **all()** to see, respectively, if any or all the elements in a variable comply with a Boolean condition. The result of those two functions will also be `logical`.

```r
any(x > 2000)
any(x < 300 | x > 500)
all(x < 300 | x > 500)
```

You may extract information about one variable according to what is going on in another variable. This will make much more sense when we talk about `matrices` or `data frames`.

```r
gender_x <- sample(c("M","F"), 26, replace = TRUE)
names(gender_x) <- names(x)

gender_x[x > 300 & x < 500]
x[gender_x == "F"]
```

We can subset to actually modify our variable.

```r
y <- x
y[c(2,4,6)] <- 0
y
y["m"] <- 26894321
y
```

3.1.9 Reversing, sorting, rearranging a vector

rev() reverses head-to-tail a given variable.

```r
x
rev(x)
```

sort() sorts a variable from the smallest to the highest value or, if it's a string character vector, in alphanumerical order. That is its behaviour by default. The sorting might be done in decreasing order, by setting `decreasing = TRUE`.

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

We can display the information in our variable, sorted according to the names of its elements, instead of the values themselves.

```r
x[sort(names(x))]
x[sort(names(x), decreasing = TRUE)]
```

Imagine now I have another variable, `y`, and that I want to have a look at the values in `x` according to what is going on in `y`.

```r
y <- round(rnorm(26),2)
names(y) <- names(x)
y
x
sort(y)
names(sort(y))
x[names(sort(y))]
```

That last line gives us the values of `x` sorted according to the increasing values of `y`. Thus, we know that the first individual in the outcome is the one with the smallest value in `y`. All that said, we have used a trick above: to use **sort()** in that way, you need both variables `x` and `y` to have `names`. However, more often than not, this will not be the case. In a data frame, for instance, where each column is a variable with no names attached. In that scenario, using **sort()** to subset variables would be a mistake, one that you will very likely make several times, like this.

```
x[sort(x)]
# or
x[sort(y)]
```

Indeed, **sort()** does not work for sub-setting cause it does not provide index numbers, but the variable itself, with its actual values.

In cases as such you need to use **order()**, which will render a numeric vector with the indices of the sorted elements in the vector.

```
sort(y)
order(y)
order(y, decreasing = TRUE)
```

The difference between **sort()** and **order()** is that the former sorts the variable out and it produces the values of the variable itself, just arranged in increasing or decreasing fashion, whereas **order()** renders the position that the `sorted` elements of `x` occupy in the original variable. Therefore, because **order()** produces indices, we can subset information using those sorted index numbers.

```
x[order(x)]
```

which gives the same result as:

```
sort(x)
```

However, and most crucially, you will be using **order()** to sort one variable, according to the arranged values in another variable. This will be extremely helpful when you deal with 2d datasets, that is `matrices` and `data frames`.

```
x[sort(y)]

x[names(sort(y))]
x[order(y)]
```

The last line above, and not the first one, gives us the information in `x` according to the increasing values in `y`. Thus, we know for instance that the first individual appearing in the output, whatever its value is in `x`, it presents the smallest value in `y`. This is similar to what you would be doing in `excel`, when you arrange data according to increasing (or decreasing) values in one variable, but then you look at how the other variables behave.

Last couple of examples

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

sort(x)
y[order(x)]
```

3.1.10 Basic maths with vectors and an introduction to missing values in `R`.

It is time we introduce a little bit of maths.

```
min(x)
max(x)
sum(x)
sum (x > 300) # Why?
```

```
mean (x > 300) # what is it?
20/26
sum(x[x > 300])

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

A very common issue regarding any data analysis is the presence of missing values. How does `R` deal with them? Here is a taster.

```
y <- x
y[3] <- NA
y
summary(y)
mean(y) # Why?
is.na(y)
sum(is.na(y))
sum(!is.na(y))
mean(is.na(y))
any(is.na(y))
all(is.na(y))
mean(y)
mean(y, na.rm = TRUE)
sum(y < 200, na.rm = TRUE)
sum(y < 200)
```

3.1.11 Vectorisation

We will finish this (long) chapter devoted to vectors by mentioning `vectorisation`, a very 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.

```
x <- 1:6
x + 5
x + c(2,5)
x + c(2,5,3)
x + c(2,5,3,7)
```

`Vectorisation` is something to take into account. We will have more examples about this when we talk about two-dimensional datasets. Indeed, not only `R` considers variables as vectors, but it considers them as `column` vectors. But to see this more clearly, we'll wait to tw-dimensional datasets.

3.1.12 Time to work

1 - Create a vector `x` containing 20 random numbers taken from the normal distribution.

- Extract the last element of `x` using **length()**
- What positions do occupy the smallest and the highest values in `x`?
- Arrange the elements of `x` in decreasing order

```
x <- rnorm(20)
which.min(x)
which.max(x)
```

```r
x[length(x)]
sort(x,decreasing = T)
```

2 - Create a vector `x` with your name, your birthplace, and the date on the month you were born.

- What type of vector is it?
- Give the following names to the elements in that vector: "Me", "Place", "Date"
- Extract the Place from that `x` vector in two different ways.

```r
x <- c("David", "near Bilbao", 26)
class(x)
names(x) <- c("Me", "Place", "Date")
x
x["Place"]
x[2]
```

3.2 *Matrix*

`Matrix` objects allow storing elements in a two-dimensional table, that is with rows and columns. Remember that matrices, as it happened with vectors, will only host homogeneous data. That is, they can be numeric, character, logical etc. but not mixed.

We use the function **matrix()** to create matrices out of scratch. There are three arguments that are key in that function:

-`nrow` and `ncol`: They are uses to specify, respectively, the number of rows and the umber of columns in the matrix. At least one of them must be implemented.

-`byrow`: Matrices are filled up, by default, column-wise. If you set the argument `byrow = TRUE`, the matrix will be filled up row-wise

```r
matrix(1:15, nrow = 5)
matrix(1:15, nrow = 3)

matrix(1:15, ncol = 3)
matrix(1:15, ncol = 5)

matrix(1:15, ncol = 5, nrow = 10)

matrix(1:15, nrow = 5)
matrix(1:15, nrow = 5, byrow = TRUE)
```

The functions **rownames()** and **colnames()** produce dimension-wise names for the matrix.

```r
set.seed(123) # This is just to be sure we all get the same result
myM <- matrix(sample(1:15), ncol = 3)
colnames(myM) <- c("Var.1", "Var.2", "Var.3")
rownames(myM) <- letters[1:5]
dimnames(myM)
myM
```

3.2.1 Matrix subsetting

Regarding subsetting, two indexes are used, separated by a comma. The first one refers to the rows, the second one to the columns. As follows: `mymatrix[rows, columns]`.

```
myM[4,3]
myM["d","Var.3"]
myM[,2]
myM[,"Var.2"]
myM[2,]
myM["b",]

myM[2:4,2:3]
myM[-c(1,5),-1]
myM[c(2,4),c(2,3)]

myM[c("d","a"),c("Var.3","Var.2")]
myM[c(4,1),c(3,2)]
```

We can subset according to the outcome of a boolean question.

```
myM[,"Var.3"]
myM[,"Var.3"] > 10
myM[myM[,"Var.3"] > 10,]
myM[myM[,"Var.3"] > 10, 1:2]
myM[c(F,T,T,F,F),1:2]
```

As we have seen with `vectors` you can make negative choices to discard rows or columns.

```
myM[-c(1,5),-1]
```

We have mentioned above `vectorisation`, by which `R` tends to convert everything to vectors, if you allow it. This is also the case when you extract one single column or one single row form a matrix. You can avoid this by setting `drop = FALSE` within the `[]`.

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

Note that by doing so, you keep the information given by the rownames, which might be handy.

```
myM[myM[,3] >= 8, 1, drop = FALSE]
```

3.2.2 The apply() function.

We will introduce here a very important function, namely **apply()**. This is the most straight forward function of a whole family of functions, including `lapply()`, `sapply()` and `tapply()`, among others. If you continue to code in `R`, you will most probably end up using those functions.

apply() will allow you operate either on columns or in rows. If you set the dimension argument in **apply()**, i.e. `MARGIN`, to 1, that will be rows; if 2, columns. You will see here a few examples. Some calculations are so commun that shortcut functions have been created to keep things simpler.

```
apply(myM, 1, mean)
rowMeans(myM)
apply(myM, 2, mean)
colMeans(myM)

apply(myM, 1, sum)
```

```r
rowSums(myM)
class(rowSums(myM))

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

The functions applied above are in all cases generic functions, already made-up for you. You can apply more complex functions though, as seen her below. In such cases, you will include something like `function(x)` within **apply()**, where `x` will mean `each row` or `each column` depending on whether you use `1` or `2` as the middle argument in **apply()**.

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

apply(myM, 1, function(x) mean(x))
apply(myM, 1, sd)
apply(myM, 1, sd/mean)
```

The use of `x` is purely conventional. Indeed, you can use whatever you feel like in `function()`, as long as you respect it when you develop what you are trying to do under **apply**. for instance, you might find helpful using the words `row` or `column` when you start using apply, so that everything is more explicit to you.

```r
apply(myM, 1, function(row) sd(row)/mean(row))
apply(myM, 2, function(column) sd(column)/mean(column))
```

In fact, what you put down within `function()` is irrelevant, as long as it is coherent with the term you use when calling the functions you need.

```r
apply(myM, 2, function(hzrthbzrurth) sd(hzrthbzrurth)/mean(hzrthbzrurth))
```

What happens with **apply()** if there are missing values in the dataset? In this case, as we have already seen, `R` does not take for granted the fact that you know there are missing values.

```r
M.na <- myM
M.na[4,1] <- NA
M.na[2,3] <- NA
M.na

rowMeans(M.na)
colMeans(M.na)
```

If we decide to remove missing values, this will imply removing the whole row where the missing value is located. Why the row? Remember: in regular datasets, individuals appear in rows, and variables in columns. Therefore, removing a row is tantamount to removing an individual, and not a variable.

```r
na.omit(M.na)
na.exclude(M.na)

colMeans(na.omit(M.na)) # !!
colMeans(myM)
```

3.2.3 Binding matrices

You may have two datasets that you want to bind together. You can do it using **cbind()** (column-wise binding) or **rbind()** (row-wise binding). The dataset to be bound must have compatible dimensions: **cbind()** requires equal number of rows, and **rbind()** requires equal number of columns.

```r
B <- matrix(sample(1:15), ncol = 3)
rownames(B) <- letters[1:5]
colnames(B) <- c("Var.4", "Var.5", "Var.6")
B
C <- matrix(sample(1:15), ncol = 3)
rownames(C) <- letters[6:10]
colnames(C) <- c("Var.1", "Var.2", "Var.3")
C

cbind(myM, B)
rbind(myM, C)
```

Be careful though: nor **cbind()** nor **rbind()** will check how your columns or rows are arranged. When you use **cbind()**, you need to be sure that rows in both datasets belong to the same individuals and that they are ordered in the same way. The same goes for the columns when you use **rbind()**.

```r
rbind(myM, B)
cbind(myM ,C)
```

Indeed:

```r
Bs <- B[sample(rownames(B)),]
cbind(myM, Bs)
```

3.2.4 Last notions about matrices: transposition, math and usefulness.

We *transpose* a two-dimensional dataset when we convert the rows of the original dataset as columns, and vice versa. This is a very important feature in biology, specially in *omics* experiments. Indeed, in regular statistics the number of individuals is higher than the number of the measured variables (p). However, in *omics* experiments we are confronted to singular datasets where p »> n. This is why, for instance, when we receive a transcriptomics dataset, samples appear as columns and genes (i.e. variables) appear as rows; it is much easier to handle this way. The issue here is that some mathematical procedures require the input dataset to be regular; therefore, we must transpose our omics dataset.

In `R`, we transpose a dataset by simply using **t()**

```r
t(myM) # Well, here we go from regular to singular, but you get it.
```

As for **maths**, usual operators in matrices work term by term. Again, be careful, because vectorisation applies here as well.

```r
myM + 2
myM * c(2,3)
myM * c(2:4)

myM + B
myM + B[-1,]
myM + t(B)
```

As for the multiplication, please make the difference between the term by term multiplication, and the product of matrices.

```r
myM * B
myM %*% B

myM * t(B)
myM %*% t(B)
```

As for **usefullness**, you will not be creating matrices from scratch very often. Indeed, as far as 2D datasets go, most of the times you will be using `data frames`. All that said, most of the things things we have said here about `matrices` are also valid far `data frames`, most particularly the way of sub-setting and extracting information, etc.. More importantly, you will be most likely dealing with `matrices` you have not created, because the outcome of some common function in `R` are `matrices`. Therefore, it is important you are also aware about their homogeneous nature etc.

3.2.5 Time to work

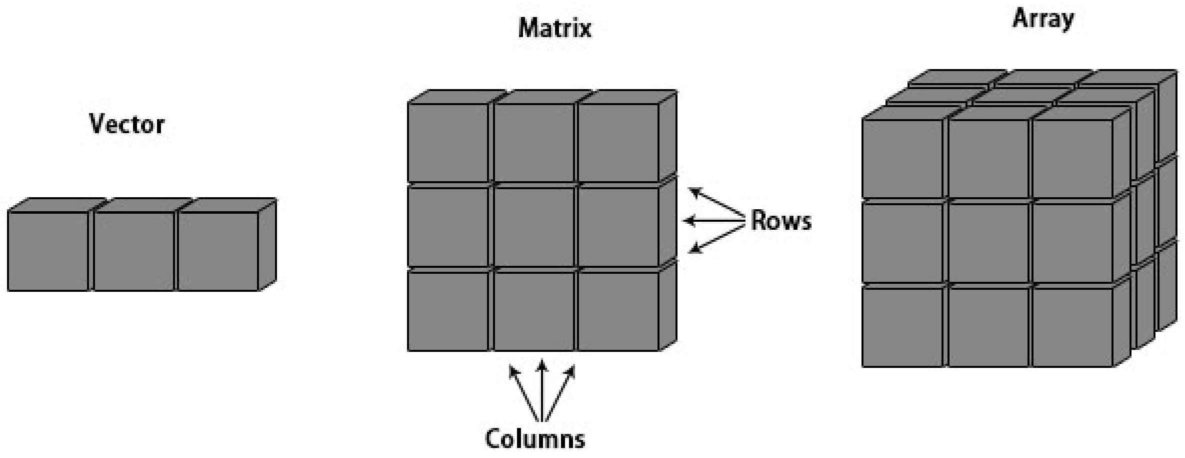
1 - Build a matrix `M1` with five columns randomizing all numbers between 1 and 50

- Calculate the median and the mean of every row in M1
- Calculate the coefficient of variation (sd / mean) of every column in M1
- Multiply the first column in M1 by the second and third columns in M1
- Divide M1 by its seventh value in its fourth column
- Make the sixth value in the third column to be NA.
- Then calculate column wise means of M1 without the incomplete sample

```
M1 <- matrix(sample(1:50), ncol = 5)
apply(M1, 1, median)
apply(M1, 1, mean)
apply(M1, 2, function(x) sd(x)/mean(x))
M1[,1]*M1[,c(2,4)]
M1 / M1[7,4]
M1[6,3] <- NA
colMeans(M1, na.rm = TRUE)
```

3.3 *array*

`array` type takes `matrix` structure to n-dimensions, where n > 2. In the figure here, you will find an example of a simple array, where n = 3.



```
E <- array(c(1:8, rep(1,8),seq(0,1,len = 8)), dim = c(2,4,3))
?("array") # What entry data type requires the array() function?
```

```r
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)
```

3.4 *list*

3.4.1 Creating a list

The `list` is a very important object type because of its flexibility. Indeed, many function outcomes are lists. Also, you will be creating lists yourself very often when you use loops etc. Objects of type `list` allow storing heterogeneous (numeric, character etc.) data in a single object. Besides, lists are non-structured objects, so that each element in a `list` may be `vector`, `matrix` etc.; you may even a store a list `list` within a `list`! You can visualize a `list` as a dresser 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.

list: anything may go in.
Drawer #1: a vector
Drawer #3: a matrix
Drawer #7: a list, why not?
This is why we say list are recursive.

The most straight forward way to create a list in `R` is using **list()**.

```r
mylist <-
  list(
  matrix = matrix(1:15, ncol = 3),
  numeric = rnorm(8),
  text = "I am giving some character to the list",
  score = factor(1:8),
  stupid.list = list(remark = "this is stupid",
                     reply = c("i", "know"),
                     stupidity.score = 94),
  number = 59)

mylist
names(mylist)
```

You can use **length()** to know how many elements a list has, regardless the nature of each of those elements.

```
length(mylist)
```

Can you tell what is going on in the following lines?

```
list(1,2,3,4,5)
list(c(1,2,3,4,5))
list(1:5)
list(c(1,2,3,4,5),6)
```

3.4.2 List subsetting

We can subset lists with `[]`, `[[]]` or `$`. If you subset with simple single square brackets, in the same way as we use them with `vectors`, the result will be a smaller list. Using single square brackets, will allow seeing what is in the drawer(s), selecting or removing some drawers of your choice, but not manipulating the content of those drawers. If you use double square brackets you may subset only one of the components of the list, but you will be able to manipulate it. The use of `$`, which works similarly to `[[]]`, is highly recommended when the elements of the list are named. In summary, if the third element of `mylist` is called `whatever`, you can subset it with `mylist[[3]]`, `mylist[["whatever"]]`, or `mylist$whatever`, but not `mylist$3`.

The following code chunk tries to shed some light on all that.

```
mylist
mylist[1]
colMeans(mylist[1])
mylist[[1]]
colMeans(mylist[[1]])
myshortlist <- mylist[c(1,4)]
mylist[c(1,4)]
mylist[[1:2]] ## !!??
```

Again, when using `[[]]` or `$`, you can access ht edata on that element and manipulate it.

```
mylist[[1]] + mylist[[6]]
mylist[["matrix"]] + mylist[["number"]]
mylist$matrix + mylist$number
mylist[1] + mylist[6]

mylist["matrix"]
mylist[["matrix"]]
mylist[[1]]
mylist$matrix
colMeans(mylist[["matrix"]])
colMeans(mylist$matrix)
colMeans(mylist["matrix"])
```

The non-ambiguity rule applies when calling up an element of a list using `$`: you can use part of the full name of the element, as long as that part is not in common with the name of any other element in the list. However, this does not work if you are sub-setting elements using `[]`. In this case, the exact pattern needs to be matched.

```
mylist$nume
mylist$num
mylist$t
mylist$s
mylist$sc
mylist$st
mylist$m + mylist$sc
```

The non ambiguityrule only applies when using the `$`, not `[[]]`.

```
mylist[["nume"]]
mylist["nume"]
```

Of course, sub-setting may in itself be recursive.

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

mylist[[1]][1,2]

mylist$matrix[1,]
mylist[["matrix"]][1,]

mylist$st[1]
mylist$st$r
mylist$st$rep
mylist[[5]][[2]]
mylist[[5]][[2]][[1]]
length(mylist[[5]][[2]])
length(mylist$st$rep)
```

You can easily add a new element to a list. You can do that in two ways, using `[[]]` or `$`.

```
mylist
length(mylist)
names(mylist)
mylist$ten.to.one <- 10:1
mylist
names(mylist)
length(mylist)
mylist[["alphabet"]] <- letters
mylist
names(mylist)
length(mylist)
```

3.4.3 Time to work

1 - Provide TP_list here below:

```
TP_list <- list(a = 1:10,
                b = "Good morning",
                c = "Hi",
                r = 11:20)
```

- Write an R statement that will add 1 to each element of the first vector in TP_list.
- Write an R expression that will give all elements, except the second, of the first element in TP_list.
- Write an R statement to add a new item `z = "New"` to list_TP.
- Write an R statement that will assign the 11th to 20th letters of the alphabet as names to the elements of `r` in TP_list.
- Write an R statement that will give the length of vector `r` of TP_list.

```
TP_list[[1]] + 1

TP_list[[1]][-2]
TP_list$a[-2]

TP_list$z <- "New"

names(TP_list$r) <- letters[11:20]

length(TP_list$r)
```

3.5 *data.frame*

`Data frames` are the most widely used 2D data structure. The big difference between a `matrix` and `data frames` in R is that the latter may host, column-wise, heterogeneous data (numeric, character etc.). A helpful way to think about data frames may be to visualize them as a group of vectors of equal size organized in columns, where every vector is a variable and can differ in class from the data type in the other columns.

In fact, `data frames` can bee seen as 2D-structured `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 in it is a `vector` of any kind and where all of them have the same length, which makes possible to arrange those elements in columns. This is the most widely used data structure when dealing with datasets.

3.5.1 Creating a data frame

In the following lines we will see how to create `data frames` using the **data.frame()** function. We will also see how not to create what we may wrongly think are `data frames`.

We will start by creating two variables that we are going to assemble (or not) into a `data frame`.

```
# 20 random values between 150 and 180, taken from the uniform law, rounded to zero decimals.
height <- round(runif(20,150,180),0)
# 20 random values between 50 and 90, taken from the uniform law, rounded to zero decimals.
weight <- round(runif(20,50,90),0)
sex <- sample(c("M","F"), 20, replace = T)
eyes <- sample(c("blue","black","green","brown"), 20, rep = T)

class(height)
class(weight)
class(sex)
class(eyes)
```

We might be tempted to use **cbind()** to create a `data frame` out of those variables, but you would be creating a `matrix` instead.

```
not.df <- cbind(height,weight,sex,eyes)
not.df
class(not.df)
summary(not.df)
```

Instead, you should be using **data.frame()** to build a `data frame`, either from existing variables, as here:

```
mydf <- data.frame(height, weight, sex, eyes)
summary(mydf)
```

Or from scratch, as here:

```
mydf <- data.frame(height = round(runif(20,150,180), 0),
                   weight = round(runif(20,50,90), 0),
                   sex = sample(c("M","F"), 20, replace = T),
                   eyes = sample(c("blue","black","green","brown"), 20, rep = T))

mydf
summary(mydf)
```

Bear in mind, however, that even though you will be creating `data frames` on your daily work, most of your datasets of interest will be imported into `R` already as `data frames`.

The use of `rownames` is deprecated in `data frames`, and there is a good reason to be so. That said, you can feel more comfortable with `rownames`, especially when carrying out analysis such as transcriptomics etc. As we have seen with the `matrix`, you can use the function **rownames()** to create them.

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

3.5.2 Data frame sub-setting

Because they present the advantages of both `matrices` and `lists`, we can subset subset `data frames` using `[,]`, `[]`, `[[]]`, and `$`, and you can even combine them. This might seem messy, but you will very soon see the logic of it and will be able to implement and customize those sub-setting tools in your daily work.

You can use `[,]` with `data frames` in the same way you use it with `matrices`. The class of outcome will be in agreement with what you would get out of `matrices`, that is if you subset columns, along with or without rows.

```
myM[1:3,1]
mydf[1:3,1]

myM[1:3,2:3]
mydf[1:3,2:3]

myM[,1]
mydf[,1]

class(myM[1:3,1])
class(mydf[1:3,1])

class(myM[1:3,2:3])
class(mydf[1:3,2:3])

class(myM[,1])
class(mydf[,1])
```

However, the outcome with `matrices` and `data frames` will be inevitably different when you use `[,]` to subset a single row, regardless whether you subset columns or not.. And that is for a good reason: `data frames` are meant to contain heterogeneous data

```
myM[1,]
mydf[1,]

class(myM[1,])
class(mydf[1,])

myM[1,1:2]
mydf[1,1:2]

class(myM[1,1:2])
class(mydf[1,1:2])
```

Also:

```
mydf[,"sex"]
class(mydf[,"sex"])

mydf[,c(1,3)]
mydf[2:5,c("sex","weight")]

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

As we have said, because `data.frames` are also a particular type of `list`, we can subset them also using `[]`, `[[]]` and `$`. In all those cases, we must keep in mind that we will be sub-setting columns (i.e variables), since `data frames` are a special type of `list` where columns are the elements of that `list`.

When you subset with `[]`, without any mention to rows, the nature of the date is kept as it was: a `data.frame`. This is not the case when you use `[[]]` or `$`

```
mydf[1]
class(mydf[1])
mydf["height"]

mydf[,1]
class(mydf[,1])
mydf[,"height"]
# Bear in mind:
myM[1] #!!
```

As opposed to these two here below. Indeed, information is vectorised when `[[]]` or `$` are used.

```
mydf[[1]]
mydf$height
```

When we use `$` to sub set a `data frame`, the non-ambiguity rule applies

```
mydf$h
```

It is with `data frames`, that sub-setting on boolean conditions makes whole sense. That is something you will be using all the time. Look at the examples here below.

```
sum(mydf$eyes == "green") # How many people do have green eyes.
mydf[mydf$eyes == "green", "weight"] # Weights of people with 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", c("eyes","weight")]
mydf[mydf$eyes == "green" & mydf$sex == "M", "weight"]
mydf[mydf$eyes != "green" & mydf$height > 170, c("sex", "weight")]
```

When you start coding, it is not always easy to grasp some of those lines above in one go. Let us do the last one, for instance, step by step.

```
mydf[mydf$eyes != "green" & mydf$height > 170, c("sex", "weight")]

res <- mydf[mydf$eyes != "green",]
res
res <- res[res$height > 170,]
res
res <- res[, c("sex", "weight")]
res
```

Three last points about `data frames`.

First: As opposed to what happens with lists, you can manipulate variables when you use `[]` to subset `data frames`, we do not need to use `[[]]`.

```
mydf[1] + 50
mydf[1:2] + 50

mydf["height"] + 50
mydf[c("height", "weight")] + 50
# As opposed to:
mylist["numeric"] + 50
```

Second: We can add new elements to a `data frame` as we go, as we do with `lists`.

```
mydf$score <- factor(sample(1:4, 20, replace = TRUE))
mydf["treatment"] <- rep(c("Ctr","Tr"), each = 10)
mydf$BMI <- mydf$weight / (mydf$height/100)^2
```

Last: When using the **data.frame()** function to create a `data frame`, we can tune the `stringsAsFactors` argument so that all string character variables in the `data frame` will be factors.

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

3.5.3 Time to work

1 - Load the `./Files/ATH.RData` object in your workspace.

- What class of object is it?
- What class of variables are in that object?
- Do you think any of those variables should be of a different nature?

2 - How many genes in chromosome 5 are coded in the each of the strands?

4 Some useful functions

We have already used a bunch of R functions. There are thousands upon thousands of R functions, most of which you will never know about. Even covering what are to be considered as *basic* functions might be endless work. In this chapter we will mention just a few of them, without any pretension of being thorough or exhaustive. These functions, however, we'll probably help you in your daily work and you will be using some of them very often.

4.1 head(), tail(), ncol(), nrow(), dim()

The **head()** and **tail()** functions will print, respectively, the first or the last six elements of a `vector` or a `list`. In the case of a `matrix` or a `data frame`, they will show you the first or the last six rows. The number of elements/rows to be displayed can be changed from six to whatever you wish, using the `n` argument.

```r
head(1:20)
tail(1:20)

head(mylist)
head(mylist, 2)

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

On the other hand, **nrow()**, **ncol()** and **dim()** are to be used only with 2D datasets and will tell you, respectively, how many rows, columns, or both there are in a `matrix` or a `data frame`. These functions will allow you, for instance, verify that everything is in order or to use use those functions to sub set datasets without necessarily knowing how many rows or columns there are in the dataset.

```r
ncol(mydf)
nrow(mydf)
dim(mydf)

mydf[nrow(mydf),]
mydf[,ncol(mydf)]
```

4.2 table()

You will be using **table()** to create contingency or frequency tables between variables in a data frame.

```r
table(mydf$eyes)
table(mydf$height)
```

Things get more interesting when you cross two variables.

```r
table(mydf$eyes, mydf$sex)
```

It might be more legible if you print the outcome as a `data frame`

```r
data.frame(table(mydf$eyes, mydf$sex))
```

The logic behind **table()** is that string vectors have been converted into factors, thus allowing counting.

```r
summary(data.frame(table(mydf$eyes, mydf$sex)))
```

You can use boolean operators to build contingency tables

```r
mydf$height > 160
mydf$T160 <- mydf$height > 160
```

```r
data.frame(table(mydf$eyes, mydf$height > 160))
data.frame(table(mydf$eyes, mydf$T160))
data.frame(table(mydf$weight < 75, mydf$height > 160))
```

4.3 merge()

merge() is a function that you will be using very often. It combines two datasets into a single one, in a similar way as **cbind()**. However, and this is crucial, as opposed to what happens with **cbind()**, **merge()** deals with differences in rows, and needs at least one common variable between both datasets. This is similar to what `recherchev` does in excel, for instance, but without the hassle nor the risk of error.

We are free to change this default behaviour with the `by` argument and thus be able to manually specify which columns we want to be considered as *common* when merging two data frames. This might be useful when you have two columns with different content but showing the same name in both data frames, for instance. It can also be useful when you suspect there is an error in one of the `data frames`.

Let us play with some small datasets.

```r
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"),
                   status = c ("single", "married", "single", "married"),
                   height = c(165,182,178,160))
```

or

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

Do not worry about the **read.table()** command at this point, we will be dealing with dataset import later on.

```r
df1
```

The default **merge()** behaviour:

- It considers, by default, all columns with identical names in both datasets and renders each of those variables once.
- Regarding common variables, only common observations with perfect matches in each of those common variables will be rendered . Incoherent observations will be dismissed by default
- Rows will be alphabetically arranged.

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

Those default features can be modified using `by` and `all` arguments, respectively. The former will allow picking the common variable(s) you want to use to do the merging. The latter will allow producing all existing observations in shared variables, thus producing missing values in exclusive variables coming from the dataset where a given observation does not exist.

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

merge(df1, df2, by = "name")
merge(df2, df1, by = "name")
```

Both arguments are useful to solve possible discrepancies between the datasets.

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

Those arguments might also be useful to detect errors.

```
df4 <- df2
df4$name <- df1$name[1:4]
df4
merge(df1, df4) # Error in the datasets?
merge(df1, df4, by = "name") # Indeed, status do not match in df1 and df4
merge(df1, df4, all = TRUE)
```

4.4 grep() and grepl()

grep() and **grepl()** select elements in a vector containing a given pattern. This will allow, for instance, picking the `rownames` or the `colnames` in a dataset containing a given chain of characters. The difference between **grep()** and **grepl()** is that the former produces the index of the elements in the vector containing the chain of characters, and the latter returns a logical vector stating whether the elements contain (`TRUE`) or not (`FALSE`) that chain of characters.

```
mydf
grep("ght", colnames(mydf))
grepl("ght", colnames(mydf))

grep("ght$", colnames(mydf))
grepl("ght$", colnames(mydf))

grep(".e", colnames(mydf))
grepl(".e", colnames(mydf))

grep("^.e", colnames(mydf))
grepl("^.e", colnames(mydf))

grep("^.*e", colnames(mydf))
grepl("^.*e", colnames(mydf))
```

In most base-`R` situations, **grep()** or **grepl()** will produce the same outcome.

```
mydf[, grep("ght", colnames(mydf))]
mydf[, grepl("ght", colnames(mydf))]
```

In the examples given here below, we will stick to **grep()**, knowing that the results with **grepl()** would be identical.

```
mydf[, grep("^.e", colnames(mydf))]
mydf[grep("M", mydf$sex), grep("^.e", colnames(mydf))]
mydf[,grep("x|y",colnames(mydf))]
mydf[,-grep("x|y",colnames(mydf))]
```

However, there are exceptions where both functions do not produce identical results. Because the outcome of **grep()** is numeric, for instance, we can combine its result with any index from any other column we want to consider. You should not be doing so with **grepl()**

```
mydf[,c(grep("^w", colnames(mydf)),4)]
mydf[,c(grep("^w", colnames(mydf)),ncol(mydf))]
mydf[,c(grepl("^w", colnames(mydf)),4)]
mydf[,c(grepl("^w", colnames(mydf)),ncol(mydf))]

mydf[,c(grepl("ght", colnames(mydf)),4)]

mydf[,c(grep("h", colnames(mydf)),ncol(mydf))]
mydf[,c(grepl("h", colnames(mydf)),ncol(mydf))]
```

4.5 sub() and gsub()

We use those two functions to substitute string patterns by something else. The **sub()** function substitutes the first match, whereas **gsub()** substitutes all matches.

```
c <- c("p", "pp", "ppp")
c
sub("p", "t", c)
gsub("p", "t", c)
```

They work with numbers to, but the outcome will not be numeric anymore.

```
n <- c(2, 22, 222)
n
sub(2, 4, n)
gsub(2, 4, n)
as.numeric(sub(2, 4, n))
```

As always, you can use those in a 2D dataset. And you can embed substitutions.

```
mydf2 <- mydf
mydf2
mydf2$sex <- sub("M", "male", sub("F", "female", mydf2$sex))
mydf2
```

And you can leave empty quotes if you just one to remove patterns

```
c <- c("ps", "pps", "pppss")
c
sub("s", "", c)
gsub("s", "", c)
```

4.6 aggregate()

The **aggregate()** function executes conditioned calculations by slicing the dataset in distinct groups, and then applying a function to each of those groups. Please note that this approach has been greatly improved and simplified in the dplyr package from tidyverse.

```
aggregate (mydf$weight,
           by = list (gender = mydf$sex),
           FUN = mean)
```

The `by` argument requires a list because it allows multiple partitioning variables.

```
aggregate (mydf$weight,
           by = list (gender = mydf$sex, eye_colour = mydf$eyes),
           FUN = mean)

aggregate (mydf$weight,
           by = list (gender = mydf$sex, eye_colour = mydf$eyes),
           FUN = function(x) sd(x)/mean(x))
```

4.7 Time to work

We will be using the `ATH` object here.

```
load("./Files/ATH.RData")
```

1 - Create a new variable `gene_length` out of existing `Lower.end` and `Upper.end` variables.

2 - Make a contingency table with the `Strand` and the `Chr` variables.

3 - Using **aggregate()**, calculate the average gene length of those groups in the contingency table.

4 - Load `./Files/ATH_transc.RData` and get the genomic information for those genes. Could you explain the results?

Chapter 5: Data Import / Export

5.1 Dataset import

dataset import into R has been greatly eased and improved in RStudio thanks to dataset import-devoted tabs and menus. Nonetheless, many long-term `useRs` tend to use always their favorite function to import their datasets, eventually tuning its parameters to adapt the import to a given dataset. There are loads of functions and packages to import data into `R`, and it would be sort of a waste of time here to go through all of them. We will try to keep things simple.

A classic function to import datasets is **read.table()**, which will render a `data frame` in `R`. It should be noted that this function does allow importing `.txt` or `.csv` files, but not `.xls` or `.xlsx` files. Arguments of **read.table()** 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.

We invite you to pen the `mport.training.txt` file with excel of any similar tabular software and see how it looks. Then use the following lines to import that dataset, and see what the different arguments produce.

```
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_bis.txt",
                       header = TRUE, sep = "\t", dec = ",")
import_C <- read.table("./Files/import.training_bis.txt",
                       header = TRUE, sep = "\t")

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()**, which are widely used, are just particular cases of that original **read.table()** function, that you are free to try. 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. Time-efficiency might be an issue indeed when you are dealing with very large datasets

As we have already 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 *import.training* file and see what happens when you execute the import at the prompt.

Whatever the method you choose to import your dataset, using any of the functions above or the built-in `RStudio` menu, the corresponding line will be executed at the console. It is good practice to copy that line in your code, for the sake of traceability.

5.2 Dataset export

We will cover now `data frame` or `matrix` export from R (we will deal later with figure exports). Remarkably enough, `RStudio` does not offer to export `data frames` or `matrices` the same menu it offers to import tabular files. Thus, functions such as **write.table()**, **write.csv()** and **write.csv2** will be used, thought there are, again, many others. They are quite similar to the analogous **read...()** functions we have already seen, but a couple of arguments differ:

- `row.names` is a logical argument when exporting and numeric when importing
- `col.names` replaces `header` at export.

As you do when importing, there are a fez things that you should consider when you export your dataset, 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. And have a look at the **sink()** function.

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

```
B
sink()
```

A commonly 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_A, file = "exp.fwrite.txt", sep = "\t")
data.table::fwrite(import_A, file = "exp.fwrite.r.txt", sep = "\t", row.names = TRUE)
data.table::fwrite(import_A, file = "exp.fwrite.csv")
```

Chapter 6: Programming

6.1 Loops

Loops in `R` seem unavoidable, especially when you start coding. Therefore, it is crucial to understand how they work. Loops in `R` are largely implemented with **for()** or **while()** functions, though I personally try to avoid the latter cause it may render never-ending loops if you do not pay enough attention to your coding.

Find here below an extremely simple example of a `for` loop.

```
for (i in 1:10)
  print(paste("This is the result of the iteration number", i))
```

If you are using **while()**, you need to first initialize the variable outside the loop, and after every iteration.

```
i <- 1
while(i <= 10){
  print(paste("This is the result of the iteration number", i))
  i <- i + 1
}
```

Using loops will allow us introducing curly brackets, i.e. `{}`, in `R`. Curly brackets have two main roles in `R`, one of them being setting limits to loop environments. Those brackets are not necessary when the loop contains only one command.

Indeed, the following code produces the same result as the one above.

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

However, look at the different outcomes of the following loops. Notice the default indentation proposed by `RStudio` in both cases.

```
for (i in 1:10){
  x <- 1:i
  print(x/2)
}

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

When you produce a loop but do not store the results, nothing seems to happen because nothing is stored.

```
for (i in 1:10)
 i*2
```

Usually, when you run a loop, you want to store the result of each iteration of the loop. Depending on the context, you can want to create a `vector`, a `data frame`, or a `list` to store those results. To do so, uou can create an empty object outside the loop that you will subsequently fill up as the loop unfolds.

In the chunk here below, we store the outcome of the loop in the *res* variable.

```r
res <- c()
for (i in 1:10)
  res[i] <- i*2
# or
res <- c()
for (i in 1:10)
  res <- c(res, i*2)
```

The use of `i` in the loop is a mere convention. You can use anything to call for iterations, obtaining the same results.

```r
res <- c()
for (whatever in 1:10)
  res[whatever] <- whatever*2
res
```

Please import the `genes.df.txt` dataset using `File -> Import Dataset -> From Text (Base)`, or running the following line:

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

Imagine now that we want to do a two-way ANOVA for each gene, putting the genotype and the treatment in the model. We want then to store the results in an list, where each element in the list correspond to the ANOVA for one gene. There is another issue here, though; one that you will encounter quite often: the first n variables of the dataset (three, in our case) are not genes and will not be computed. How can you adapt your code to take that issue into account?

The easy way. You start your loop where the first gene is located, then you remove the empty elements from the resulting list.

```r
res <- list()
for(i in 4:ncol(genes.df))
  res[[i]] <- summary(aov(genes.df[,i] ~ genotype * treatment, data = genes.df))
res <- res[-c(1:3)]
```

More elegantly, perhaps, you can adapt the iteration operator, like this:

```r
res <- list()
for(i in 4:ncol(genes.df))
  res[[i-3]] <- summary(aov(genes.df[,i] ~ genotype * treatment, data = genes.df))
# Or, alternatively:
res <- list()
for(i in 1:ncol(genes.df))
  res[[i]] <- summary(aov(genes.df[,i+3] ~ genotype * treatment, data = genes.df))
```

You can name the elements of the list in agreement with the gene names.

```r
names(res) <- colnames(genes.df[4:ncol(genes.df)])
```

A final, and important, point to this short introduction to **for()** loops: the iteration values taken by `i`, or whatever you call it, are not necessarily meant to be consecutive numbers.

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

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

Actually, `i` does not necessarily have to be a number

```
for (i in c("A", "Y", "B", "C", "D", "E")) {
  print(paste("Alphabet position", which(LETTERS == i)))
}
```

6.2 Conditional statements `if-else`

You will be using conditional statements quite often in `R`. Those are statements by which, depending on the values of a given variable, you will take one action or another. This is done with **if()** and **else()** functions, or with the **ifelse()** wrapper function. The way conditional statements and actions should be interpreted is as follows: "if the condition is `TRUE`, do action 1; if not (i.e. the condition is `FALSE`), then do action 2".

Two very simple examples:

```
x <- rnorm(100) # 100 random numbers from the Normal distribution, where mean = 0.
ifelse(x > 0, "positive", "negative")
```

Let us build a fake p-value list.

```
pvs <- abs(rnorm(1000, mean = 0.05))
pvs <- pvs[pvs < 1]
pvs
```

I want to label those as significant, or not, depending on whether they are smaller than 0.05.

```
sum(pvs < 0.05)
ifelse(pvs < 0.05, "signif", "not signif")

pvs.df <- data.frame(pvs)
pvs.df$signif <- ifelse(pvs.df$pvs < 0.05, "signif", "not signif")
```

Conditional statements are quite often used within loops. Here below, a few examples:

```
y <- 0
z <- 0
w <- 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[i] <- x
}
```

This one here below implements **ifelse()**

```
x <-sample(1:10, replace = TRUE)
v <- 0
```

```
for (i in 1:10)
 v <- ifelse (x[i] < 5, v + 1, v)
```

In the next example we use an `R` built-in dataset, comprising sepal an petal lengths and widths in three different iris species. We want to know if those measures are, arbitrarily so, large or small.

```
iris.score <- matrix(ncol = 4, nrow = 150)
colnames(iris.score) <- colnames(iris[-5])
for (i in 1:4)
  iris.score[,i] <- ifelse(iris[,i] < mean(iris[,i]), "small", "large")

for (i in colnames(iris.score))
  iris.score[,i] <- ifelse(iris[,i] < mean(iris[,i]), "small", "large")
```

6.3 Alternative to loops: the `apply` family

Long-iterations loops are time consuming and not very elegant in terms of coding. As you gain experience with `R`, you will be learning how to get rid of them, probably by using a function from the `apply` family. Indeed, functions such as **apply()**, **lapply()**, **sapply()** or **tapply()** are meant to do the job in a much more time-efficient manner that loops. You already know **apply()**. You will learn about the others if you continue using `R` and this is why we introduce them here.

6.3.1 apply() family

lapply() and **sapply()** work in a very similar way, the difference being that the former will produce a list, and the second will simplify the result if possible.

Here is how they work, as compared to loops.

```
res <- c()
for (i in 1:10)
  res[i] <- i*2
res

lapply(X = 1:10, FUN = function(x) x*2)
lapply(1:10,function(x) x*2)

sapply(1:10,function(x) x*2)
sapply(1:10,function(x) x*2, simplify = FALSE)
```

Mind that the use ox `x`, and not `i`, again, is purely consensual. In the example above, you see that by running `simplify  = FALSE`, **sapply()** produces identical results as **lapply()**.

We finished the loops section above by combining **for()** and **ifelse()** to create a scoring dataset. In order to do so, we created an empty `matrix` that we filled up with every iteration of the loop.

We can alternatively use **sapply()**, which will know how to simplify the outcome into a `matrix` what would have been a `list` with **lapply()**. Besides, using the names of the variables in `iris` with **sapply()** will allow the resulting `matrix` to have the same variable names.

```
iris.score2 <-
  sapply(colnames(iris)[-5],
         function(x)
           ifelse(iris[, x] < mean(iris[, x]), "small", "large"))
class(iris.score2)
```

If we had used **lapply()** instead, we would have obtained a `list`

```
lapply(colnames(iris)[-5],
       function(x)
         ifelse(iris[, x] < mean(iris[,x]),
                "small", "large"))
```

sapply() presumes by default that it is ok to simplify the result into a `matrix` because it reads an equal amount of elements in each of the `x`. Then, it understands you may want those in a 2D dataset cause, otherwise, you would have used **lapply()**.

The chunk here below tries to explain that

```
iris.list <- as.list(iris[1:4])

sapply(names(iris.list),
       function(x)
         ifelse(iris.list[[x]] < mean(iris.list[[x]]), "small", "large"))

lapply(names(iris.list),
       function(x)
         ifelse(iris.list[[x]] < mean(iris.list[[x]]), "small", "large"))

# We remove the last element from the first element of the list
iris.list[[1]] <- iris.list[[1]][-150]

sapply(1:4,
       function(x)
         ifelse(iris.list[[x]] < mean(iris.list[[x]]), "small", "large"))
```

This is another example. Please notice that both **lapply()** and **sapply()** can be implemented in at least to different ways, the difference being quite subtle. We can sub set and call upon `x` using indices or names. The outcome in either case will be identical with **lapply()**, but not with **sapply()**. Indeed there is a subtle difference with the latter, but it is noteworthy.

```
sample.list <- list(sample1 = sample(1:20,10),
                    sample2 = sample(1:20,10),
                    sample3 = sample(1:20,10))

lapply(1:length(sample.list), function(x) c(mean = mean(sample.list[[x]]),
                                            sd = sd(sample.list[[x]])))
sapply(1:length(sample.list), function(x) c(mean = mean(sample.list[[x]]),
                                            sd = sd(sample.list[[x]])))

lapply(names(sample.list), function(x) c(mean = mean(sample.list[[x]]),
                                            sd = sd(sample.list[[x]])))
sapply(names(sample.list), function(x) c(mean = mean(sample.list[[x]]),
                                            sd = sd(sample.list[[x]])))
```

We come back now to our ANOVA example from above and see what the alternative to the **for()** loop could be.

This is the way we run the loop above.

```
res <- list()
for(i in 4:6)
  res[[i-3]] <- summary(aov(genes.df[,i] ~ genotype * treatment, data = genes.df))
names(res) <- colnames(genes.df[4:ncol(genes.df)])
```

And this could be the **lapply()** alternative

```
lapply(colnames(genes.df)[-c(1:3)],
       function(x) summary(aov(genes.df[,x] ~ genotype * treatment,
                               data = genes.df)))
```

In this particular case, **sapply()** will produce also a list cause it can not figure out how to simplify the result. However, there is a slight improvement when you use **sapply()** with the `colnames` of the dataset instead of the indices: again, names of the variables are shown on the outcome, which is not the case with **lapply()**.

```
sapply(colnames(genes.df)[-c(1:3)],
       function(x) summary(aov(genes.df[,x] ~ genotype * treatment,
                               data = genes.df)))
```

6.3.2 tapply()

tapply() is used to summarise quantitative data according to the levels of qualitative variables, i.e. `factors`. This is similar to what **aggregate()** does. It is most useful with `data frames`, most particularly when those include both categorical data (i.e. `factors`) and quantitative data which, in any case, can not be the case with `matrices`.

For instance:

```
iris
tapply(iris$Petal.Length, iris$Species, mean)
tapply(iris$Petal.Length, iris$Species, function(x) sd(x)/mean(x))
```

tapply() tends to simplification as well because `simplify = TRUE` by default. It can be tuned to `FALSE`, so that the outcome will be a list.

```
tapply(iris$Petal.Length, iris$Species, function(x) sd(x)/mean(x), simplify = FALSE)
```

What if I want to run the same math function on all quantitative variables in your dataset? You can of course run a loop:

```
m <- matrix(nrow = 3, ncol = 4)
for (i in 1:4)
  m[,i] <- tapply(iris[[i]], iris$Species,
                  function(i) mean(i)/sd(i))
rownames(m) <- levels(iris$Species)
colnames(m) <- colnames(iris)[-5]
```

But you can also combine functions from the `apply` family. The code using **sapply()** and **tapply()** instead of the loop is clearer. See here below:

```
sapply(colnames(iris)[-5],
       function(x)
         tapply(iris[[x]], iris$Species,
                function(y)
                  mean(y) / sd(y)))
```

Notice the use of distinct notations for **sapply()** and **tapply()** iterations, namely `x` and `y`, because we are not calculating upon the same elements.

If there is more than one factor in the dataset, you can use both with **tapply()**, so the `function(x)` you choose to execute will be run for the combination of both factors.

To create a combination between variables, we can use **interaction()**, which renders all possible combinations between the variables' observations. The original variables might be of any nature, but the outcome of **interaction()** will always be a `factor`:

```
aux <- data.frame(cond = rep(c("Ctrl","Tr"), each = 4),
                  gtype = rep(c("KO","WT"), 4))
interaction(aux$cond, aux$gtype)

aux <- data.frame(n1 = rep(1:2, each = 4),
                  n2 = rep(3:4, 4))
interaction(aux$n1, aux$n2)
```

We will. use our `genes.df` dataset to see how that works with **tapply()**.

```
sapply(colnames(genes.df)[-c(1:3)],
       function(x)
         tapply(genes.df[[x]],
                interaction(genes.df$genotype, genes.df$treatment),
                function(y)
                  mean(y)))
```

6.4 Functions

6.4.1 Coding a basic function

All the functions we have used so far come from built-ib or downloaded packages. However, you will also be able to build functions created by yourself, and this is what we will be talking about in this chapter. You may wonder why should you be writing your own functions. The main reason is a simple one: you won't have to re-code things you will be doing on a regular basis. That will imply:

1 - Fewer errors 2 - Better traceability 3 - Easier sharing of analysis pipelines as the The functions we will be working here are quite simple and the advantages of building them might not seem obvious. Type the following to see the source code of the **aov()** function to have a clearer idea of the advantages of wrapping your analysis within a function.

```
aov # Without the brackets to see the source code
```

Every function in `R`, whether it comes from any library or your own coding, is executed calling its *name*, followed by its *arguments* located between brackets. We have already seen that the arguments of a function can be either, *mandatory* or *optional*, the latter presenting a default value in the coded function. Functions are built or coded using the function called **function()**. When coding a function, we use `()` to define the *arguments* of the function, and `{}` to define the actions to be taken with those *arguments*. Those curly brackets ere not mandatory if the actions are coded in a single line.

Functions are `R` objects. Those coming from loaded packages will be visible in the environment of the library they come from. Those created by you will be visible in your session environment, that is your `workspace`. Therefore, they will be visible when you execute **ls()**. Likewise, you will be able to **save()** those functions as a physical `.RData` file in your computer, and they will be also saved in the `RData` file when you run **save.image()** to save your whole `workspace`.

Here below, a first example of a function coded with a single mandatory argument.

```
MyFunc <- function (x) x + 2

MyFunc <- function (x)
  x + 2
```

How do you use your new function:

```
MyFunc
MyFunc()
MyFunc(3)
MyFunc(5)
x <- MyFunc(4)
x
```

What if we code `x` as an optional argument?

```
MyFunc <- function (x = 3) x + 2
MyFunc()
MyFunc(12)
```

More about arguments in a function, including the order in which they are coded.

```
MyFunc <- function (a, b = 0) a + b
MyFunc (2,3)
MyFunc (5)
MyFunc (b = 5)
MyFunc (5,5)

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

Functions, of course, are not limited to numeric inputs and outputs.

```
greet_person <-
  function(name, greeting = "Hello, ") {
    paste0(greeting, name, "!")
  }

greet_person("Alice")
```

Please notice that the `greeting` argument is optional

```
greet_person("Donald", g = "F*** off,")
```

6.4.2 The outcome of a function

When you execute a function, it produces a result according to what is stated on the last line, and only the last line, in the function code. Look at this simple examples:

```
MyFunc <-
  function (x) {
    x + 2
  }

MyFunc(3)

MyFunc <-
  function (x) {
    x + 2
    x^2
  }

MyFunc(3)
```

```r
MyFunc <-
  function (x) {
    x + 2
    x^2
    "this is pointless"
  }

MyFunc(3)
MyFunc(1984654654678987464654)
```

Many functions, though, produce multiple results. The way to do so is: first you create within the function and object containing those results and then you code the function to render that object. Therefore, it is quite common to use a `list` to produce the results of a function. Here below we create a function, where the output is a list.

```r
MyFunc <-
  function (x) {
    list(
      add.2 = x + 2,
      square = x^2
    )
    "this is pointless"
  }
```

Please note in the example above that the last line in the function code is the `list`. It is quite common to to produce intermediate objects within the function that will be for further calculations and/or used in the final outcome.

```r
circle <-
  function (r) {
    p <- 2 * pi * r
    a <- pi * r * r
    list (radius = r,
          perimeter = p,
          area = a)
  }

res <- circle (3)

res
```

`a` and `p` variables above belong to the `circle()` function environment, and they are not available in our workspace.

```r
a
p
ls()
```

Again, bear in mind that the last line in the function, and only the last line, defines the outcome of the function. See these examples here, where I change the last line of the `circle()` function.

```r
circle.bad <-
  function (r) {
    p <- 2 * pi * r
    a <- pi * r * r
    list (radius = r,
          perimeter = p,
          area = a)
    "whatever"
  }
circle.bad(5)
circle.bad(20)

circle.bad <- function (r) {
  p <- 2 * pi * r
  a <- pi * r * r
  "whatever"
  list (radius = r,
        perimeter = p,
        area = a)
}
circle.bad(5)
circle.bad(20)
```

6.4.3 Using return(), stop(), warning(), and print().

return() is an old R function originally designed to return the outcome of a function. This means that if you use it only in the last line, it will be redundant. For instance, the two functions here below produce exactly the same results

```
MyFunc <- function (x) {
  return(x + 2)
}

MyFunc(2)

MyFunc <- function (x) {
  x + 2
}

MyFunc(2)
```

You can find in the right forums some rather heated-up geek discussions trying to settle whether **return()** should be used in such cases. We will spare you the details, but we will make a few points about it.

return(), in fact, *exits* a function and returns a value. It is therefore used to specify the value that should be returned when the function is called. Indeed, when **return()** is encountered in a function, it immediately exits the function and returns the specified value. That is, an outcome is produced before reaching the last line of the function code if **return()** is called before that last line is reached.

```
MyFunc <-
  function (x) {
    return("it stops here")
    x + 2
  }

MyFunc(2)
# which produces the same result as:
MyFunc <-
  function (x) {
    return("it stops here")
    return(x + 2)
```

```r
  }

MyFunc(2)
```

Now, imagine I want to develop a function to detect whether a certain value is among the elements in my vector, that I will call a

```r
a <- sample(1:10, 6)
```

We can now build the function in at least two ways, one of them using **return()** and another one not using it.

```r
give_me_five <-
  function(x) {
    if(5 %in% x)
      return("I give you five!")
    else (return("I don't give you five"))
  }

give_me_five <-
  function(x) {
    if(5 %in% x)
      res <- "I give you five!"
    else(res <- "I don't give you five")
  res
  }
```

You pick the one that pleases you most. However, there is one thing that should be noted: **return()** exists the function whenever it is called, before reaching the end of the function. We think this might be time saving when an output can be produced early in a time-consuming function, thus sparing some time. Also, using **return()** can make coding easier in complex functions.

Let us see another example, in which we code a function to calculate the surface of a rectangle/square or a triangle. We let you decide which one reads better.

```r
area_tri_tetra <-
  function(base_or_length,
           height_or_width,
           nb_angles) {
    if (nb_angles %in% c(3, 4) == FALSE)
      return("ERROR: this function only works for surfaces with three or four angles")
    if (nb_angles == 4)
      return(base_or_length * height_or_width)
    if (nb_angles == 3)
      return(base_or_length * height_or_width / 2)
  }


area_tri_tetra(2, 5, 6)
area_tri_tetra(2, 5, 4)
area_tri_tetra(2, 5, 3)

area_tri_tetra <-
  function(base_or_length,
           height_or_width,
           nb_angles) {
    if (nb_angles %in% c(3, 4) == FALSE)
```

```r
      res <- "ERROR: this function only works for surfaces with three or four angles"
    if (nb_angles == 4)
      res <- base_or_length * height_or_width
    if (nb_angles == 3)
      res <- base_or_length * height_or_width / 2
  res
  }

area_tri_tetra(2, 5, 6)
area_tri_tetra(2, 5, 4)
area_tri_tetra(2, 5, 3)
```

You see in the example above that we used **return()** (or not, but let's keep it simple here) to produce an outcome that was, in fact, the product of a wrong input. That was not very elegant. instead, we should have used **stop()** which also exits the function, but automaticcally produces the error message in the console.

```r
area_tri_tetra <-
  function(base_or_length,
           height_or_width,
           nb_angles) {
    if (nb_angles %in% c(3, 4) == FALSE)
      stop("This function only works for surfaces with three or four angles")
    if (nb_angles == 4)
      return(base_or_length * height_or_width)
    if (nb_angles == 3)
      return(base_or_length * height_or_width / 2)
  }

area_tri_tetra(2, 5, 6)
```

Thus, **stop()** triggers errors. We can use **warning()** to produce warnings.

```r
my_warning_function <-
  function(x) {
  # if (x < 0)
  #   warning("Input is negative. Results may be unexpected.")
  sqrt(x)
}

my_warning_function(4)
my_warning_function(-4)
```

Beware of the difference: **stop()** exits the function, whereas **warning()** produces a warning, allowing the function to pursue.

```r
my_stop_function <-
  function(x) {
  if (x < 0)
    stop("Input is negative. Are you dumb?")
  sqrt(x)
}

my_stop_function(4)
my_stop_function(-4)
```

Some people mistake **return()** and **print()**. The latter will print at the console whatever it is asked at any given point in the code, but this will affect the rest of the function code.

```r
MyFunc <-
  function (x) {
    print("whatever")
    x + 2
  }

MyFunc(15)

MyFunc <-
  function (x) {
    return("whatever")
    x + 2
  }

MyFunc(15)
```

Thus, **print()** is very useful to provide annotations as the function runs.

```r
MyFunc <-
  function (x) {
    print(paste("I am adding 2 to", x))
    x + 2
  }
```

Let us see one last example that will allow us combine logical statements and raise awareness about the importance of how the order of the commands might be important. Imagine we coded a function to tell whether you should stay at home during the covid pandemic, depending on your fever and the pressence, or not, of other symptoms.

```r
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")
    return("You must be tested for the COVID and remain confined")
  }

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

Combine logical arguments

```r
covid_guideline <-
  function (fever,
            headache,
            cough,
            diarrhea) {
    fever_38 <- fever >= 38
    if (sum(fever_38, headache, cough, diarrhea) <= 2 &
        sum(fever_38, cough) == 2)
```

```
      return ("You should be tested for the COVID. Remain confined")
    if (sum(fever_38, headache, cough, diarrhea) <= 2)
      return ("See how it evolves. Let us know if you notice further symptoms")
    return("You must be tested for the COVID Remain confined")
  }

covid_guideline(39, 0, 1, 0)
covid_guideline(39, 1, 0, 0)
```

But bear in mind ht order in which you code the staments

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

covid_guideline(39, 0, 1, 0)
covid_guideline(39, 1, 0, 0)
```

6.4.4 Using a `for` loop within a function

We will finish this chapter seeing how to use a `for` loop within the function. First, let us introduce here the `%%` operator: it returns the remainder of the division between the number preceding the operator and the one after the operator.

```
27 %% 3
27 %% 2

27 %% 3 == 0
27 %% 2 == 0
```

Let us now code a function to tell if a given number is a prime number. Reminder: A prime number is a natural number greater than 1 that is not a product of two smaller natural numbers.

```
is_prime <- function(n) {
  if (n != round(n) | n <= 1){
    stop ("Prime number definition applies to natural numbers higher than 1")
  }
  for (i in 2:sqrt(n)) {
    if (n %% i == 0) {
      print(paste(i,"is the smallest natural number by which",n,"is divisible"))
      return(paste(n,"is not a prime number"))
    }
  }
  return(paste(n,"is a prime number"))
}
```

```r
is_prime(97)

# Here is a small explanation on the definition of prime numbers
primes_100 <- c(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
                43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97)
lapply(primes_100, function(x) 2:sqrt(x))
```

6.5 Time to work

1 - Create a function named `power` that takes two arguments, a base, and an exponent. Set the default value of the exponent to 2. The function should return the result of raising the base to the given exponent.

2 - Use the `power` function you have just created to cube the vectors in the following list. Do so in three ways: with a `for` loop, with **lapply()**, and with **sapply()**

```r
numeric_list <- list(c(1, 2, 3),
                     c(4, 5, 6),
                     c(7, 8, 9))
```

3 - Use a `for` loop to create a function named `sum_natural_numbers` that takes a positive integer n as an argument and returns the sum of the first n natural numbers. Provide an error message for cases in which the input is not a positive integer.

4 - Create a function named `filter_data` that takes as arguments a `data frame`, a column name, and a value. The function should return a new data frame containing only the rows where the specified column is equal to the given value. Use that function with the `genes.df` dataset.

5 - Using the **ifelse()** function and the `ATH` dataset, create a variable saying whether each gene in that dataset is long (longer than 2Kb), or short. How many `short`and `long` genes are there?

Chapter 7: (A very little of) Statistics

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 : ....* Stats and graphics, this is most of what `R` is about, though we should not forget pure dta wranggling. We will have in this chapter an extremely light glimpse into `R` power on statistics. It has to be light, given the numberless possibilities `R` offers in the field. Furthermore, new tools are implemented everyday into `R`, and even the most experienced statisticians learn about many of them as they go.

7.1 Inferential statistics

If an statistical test is not proposed in `R`, I'd dare saying the text does not exist.

```r
apropos("test")
```

And those are just default tests, without loading any specific library.

We will just focus for the time being on the function we use to perform a `Student test`, or `t-test`. We will first create a simulated dataset, and will run the **t.test()** function.

```r
test.data <- data.frame(x = rnorm(100), y = rnorm(100, mean=1))
my.tt <- t.test(test.data$x,test.data$y)
my.tt
class(my.tt)
```

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. In fact, it is sort of formatting what, in essence, is a `list`.

```r
length(my.tt)
names(my.tt)
my.tt$p.value
```

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.

And how do we know if the variances are equal? Well, with a test. For instance, with the Fisher test for the equality of variances. A small reminder: our null hypothesis is that variances are equal. A low p-value should allow us to reject our null hypothesis and therefore tell that the variances are indeed different.

```r
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 reject the null hypothesis and we may conclude that the variances are indeed equal. Hence, we can modify the variance parameter when running the **t.test()**.

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

You see that this time, it is not a `Welch`, but a `t-test` that has been performed. Notice that the degrees of freedom are indeed `198` , i.e. `(100 - 1) + (100-1)`.

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

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

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

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

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

```r
?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.

7.2 Descriptive statistics and some graphics

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

```r
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 can 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)
```

7.3 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.

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

Chapter 8: 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")`.

8.1 Discrete data

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

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

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

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

```r
barplot(vec)
```

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

```r
plot(iris$Sepal.Length,iris$Petal.Length)
```

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

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

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

```r
colors()
plot(vec,vec2, col = "wheat3")
plot(vec,vec2, col = 1:7)
```

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

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

```r
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.

```r
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))`

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

```r
barplot(vec)
```

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

```r
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")
```

8.2 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")
```

8.3 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 fonction you used to produce, let that be **plot()**, **boxplot()**, **hist()** etc.

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

```r
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)
```

```r
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.

```r
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")
```

8.4 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()`.

```r
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()
```