

---

## First Steps in R for Phylogeneticists

It is clear that some experience with R greatly helps in handling the materials presented in this book. In the last few years, the practice and teaching of R has increased dramatically. So it is likely that most readers will already be familiar with R. The goal of this chapter is more to remind the most fundamental notions of R, rather than to give a formal introduction for new users. It is focused on the topics required for the present book, and does not cover all introductory concepts and notions about R.

A generally deterring fact for new users is that it is almost impossible to figure out what to do if the user has no notion of languages, commands, or R itself. A learning step must be taken and this obviously has a cost. Progressing in the use of R involves successive learning steps. Of course, there are benefits to taking these steps.

R has spread through the field of computational statistics, and there is now a wide range of packages for many numerical, analytical, and graphical methods. The fields of application of R include analysis of DNA microarray data, genetics (quantitative trait loci, population analyses, etc.), morphometrics, ecological analyses, drawing maps including the use of geographic information systems (GIS) data or GoogleMaps, and interacting with a variety of other programs such as SQL and other forms of databases. Thus learning R for a specific task is very likely to be rewarding very rapidly.

If you do not know R, do not have knowledge of computer languages, and do not want to read introductory documents on R<sup>1</sup> (or cannot), then you should read, certainly carefully, this chapter. If you already have an idea of computer programming but not R, reading this chapter should be easy and will point to the particularities of R.

---

<sup>1</sup> See <http://cran.r-project.org/manuals.html> and <http://cran.r-project.org/other-docs.html>.

## 2.1 The Command Line Interface

The user can interact with R in several ways. The most interactive way is to use the command line interface (CLI). R can also be run in batch mode (i.e., noninteractive) from a system shell. There are several graphical user interfaces (GUIs), but they are restricted to traditional statistical methods (see the *Rcmdr* package), and so do not cover the wide range of methods available in R. Finally, there exist several Web servers to run R through the Internet. In this book, we concentrate on the CLI because it is interactive, versatile, and portable (i.e., the commands will run on all operating systems).

All actions are done on data stored in the active memory of the computer. These data are stored as *objects*. To characterize some data, and thus analyze them relevantly, it is often necessary to have additional information. For instance, consider a numeric variable taking the values 0 or 1: is it a count (i.e., a quantitative variable) or a code for a qualitative variable? In R the complementary information is provided by the *attributes* of the objects. We show some examples in the next section.

Commands in R are made of *functions* and / or *operators* (+, -, \*, etc). A command returns an object that is either displayed on the screen (and not stored in memory), or stored in memory using the assign operator <-. The latter requires giving a name to the object. An object may be displayed by typing its name as a command:

```
> 2 + 7
[1] 9
> x <- 2 + 7
> x
[1] 9
```

R has a wide range of functions and operators to create regular and random sequences. There are also several functions to read data from files on the disk: the most useful for us are illustrated in Section 3.8.

The user does not see her data as in a spreadsheet editor (though this is possible) because many objects with different structure can be stored and manipulated at the same time, and this cannot be represented as a spreadsheet. There are, of course, several functions to manage the objects in memory. `ls` displays a simple list of the objects currently in memory.

```
> ls()
character(0)
> n <- 5
> ls()
[1] "n"
> x <- "acgt"
> ls()
[1] "n" "x"
```

As we have seen above, typing the name of an object as a command displays its content. In order to display some attributes of the object, one can use the function `str` (*structure*):

```
> str(n)
  num 5
> str(x)
  chr "acgt"
```

This shows that `n` is a numeric object, and `x` is a character one. Both `ls` and `str` can be combined by using the function `ls.str`:

```
> ls.str()
n :  num 5
x :  chr "acgt"
```

To delete an object in memory, the function `rm` must be used:

```
> ls()
[1] "n" "x"
> rm(n)
> ls()
[1] "x"
```

There are one function and one operator that are good to learn very early because they are used very often in R: `c` concatenates several elements to produce a single one, and `:` returns a regular series where two successive elements differ by one. Here are some examples:

```
> x <- c(2, 6.6, 9.6)
> x
[1] 2.0 6.6 9.6
> y <- 2.2:6
> y
[1] 2.2 3.2 4.2 5.2
> c(x, y)
[1] 2.0 6.6 9.6 2.2 3.2 4.2 5.2
> 1:10
[1] 1 2 3 4 5 6 7 8 9 10
> 5:-5
[1] 5 4 3 2 1 0 -1 -2 -3 -4 -5
```

It happens from time to time that one forgets to assign the result of a command to an object. This may be worrying if the command took a long time to run (e.g., downloading a large number of molecular sequences through the Internet) and its results are only printed on the console. R actually stores the last evaluation in a hidden object called `.Last.value` and this may be copied into another object, for instance after typing the last command above:

```
> x <- .Last.value
> x
[1] 5 4 3 2 1 0 -1 -2 -3 -4 -5
```

## 2.2 The Help System

Every function in R is documented through a system of help pages available in different formats:

- Simple text that can be displayed from the CLI;
- HTML that can be browsed with a Web browser (with hyperlinks between pages where available);
- PDF that constitutes the manual of the package.

The contents of these different documents are the same.

Through the CLI a help page may be displayed with the function `help` or the operator `?` (the latter does not work with special characters such as the operators unless they are quoted):

```
help("ls")
?ls
?"+"
```

By default, `help` only searches in the packages already loaded in memory. The option `try.all.packages = TRUE` allows us to search in all packages installed on the computer.

If one does not know the name of the function that is needed, a search with keywords is possible with the function `help.search`. This looks for a specified topic, given as a character string, in the help pages of all installed packages. For instance:

```
help.search("tree")
```

will display a list of the functions where help pages mention “tree”. If some packages have been recently installed, it may be necessary to refresh the database used by `help.search` using the option `rebuild = TRUE`.

Another way to look for a function is to browse the help pages in HTML format. This can be launched from R with the command:

```
help.start()
```

This loads in the local Web browser a page with links to all the documentation installed on the computer, including general documents on R, an FAQ, links to Internet resources, and the list of the installed packages. This list eventually leads to the help page of each function.

The help page of a function or operator is formatted in a standard way with sections, among others, ‘Usage’ giving all possible arguments and their

eventual default values, ‘See also’ that lists related help pages (as hyperlinks in HTML and PDF formats), and ‘Examples’ that illustrates some uses of the documented item(s). Finally, data sets delivered with a package are also documented: for instance, see `?sunspots` for a standard data set in R.

A lot of packages have now *vignettes*, PDF documents with additional information and written in a very free by package authors. The list of vignettes installed on a computer is printed with `vignette()`, and a specific one is viewed by giving its name within double quotes (e.g., `vignette("adephylo")`).

Electronic mailing lists have always been critical in all aspects of the development of R [88]. This phenomenon has amplified in recent years with the emergence of special interest groups hosted on the CRAN, including one in genetics (r-sig-genetics) and one in phylogenetics (r-sig-phylo). These lists play an important role in structuring the community of users in relation to many aspects of data analysis, including theoretical ones.

## 2.3 The Data Structures

We show here how data are stored in R, and how to manipulate them. Any serious R user must know the contents of this section by heart.

### 2.3.1 Vector

Vectors are the basic data structures in R. A vector is a series of elements that are all of the same type. A vector has two attributes: the *mode*, which characterizes the type of data, and the *length*, which is the number of elements. Vectors can be of five modes: numeric, logical (TRUE or FALSE), character, raw, and complex. Usually, only the modes numeric and character are used to store data. Logical vectors are useful to manipulate data. The modes raw and complex are seldom used and are not discussed here.

When a vector is created or modified, there is no need to specify its mode and length: this is dealt with by R. It is possible to check these attributes with the functions of the same names:

```
> x <- 1:5
> mode(x)
[1] "numeric"
> length(x)
[1] 5
```

Logical vectors are created by typing “FALSE” or “TRUE”:

```
> y <- c(FALSE, TRUE)
> y
[1] FALSE TRUE
> mode(y)
```

```
[1] "logical"
> length(y)
[1] 2
```

In most cases, a logical vector results from a logical operation, such as the comparison of two values or two objects:

```
> 1 > 0
[1] TRUE
> x >= 3
[1] FALSE FALSE TRUE TRUE TRUE
```

A vector of mode character is a series of character strings (and not of single characters):

```
> z <- c("order", "family", "genus", "species")
> mode(z)
[1] "character"
> length(z)
[1] 4
> z
[1] "order" "family" "genus" "species"
```

We have just seen how to create vectors by typing them on the CLI, but it is clear that in the vast majority of cases they will be created by reading data from files (e.g., with `read.table` or `read.csv`).

R has a powerful and flexible mechanism to manipulate vectors (and other objects as well as we will see below): the indexing system. There are three kinds of indexing: numeric, logical, and with names.

The *numeric indexing* works by giving the indices of the elements that must be selected. Of course, this can be given as a numeric vector:

```
> z[1:2]
[1] "order" "family"
> i <- c(1, 3)
> z[i]
[1] "order" "genus"
```

This can be used to repeat a given element:

```
> z[c(1, 1, 1)]
[1] "order" "order" "order"
> z[c(1, 1, 1, 4)]
[1] "order" "order" "order" "species"
```

If the indices are negative, then the corresponding values are removed:

```
> z[-1]
[1] "family" "genus" "species"
> j <- -c(1, 4)
> z[j]
[1] "family" "genus"
```

Positive and negative indices cannot be mixed. If a positive index is out of range, then a missing value (NA, for *not available*) is returned, but if the index is negative, an error occurs:

```
> z[5]
[1] NA
> z[-5]
Error: subscript out of bounds
```

The indices may be used to extract some data, but also to change them:

```
> x[c(1, 4)] <- 10
> x
[1] 10 2 3 10 5
```

The *logical indexing* works differently than the numeric one. Logical values are given as indices: the elements with an index TRUE are selected, and those with FALSE are removed. If the number of logical indices is shorter than the vector, then the indices are repeated as many times as necessary (this is a major difference with numeric indexing); for instance, the two commands below are strictly equivalent:

```
> z[c(TRUE, FALSE)]
[1] "order" "genus"
> z[c(TRUE, FALSE, TRUE, FALSE)]
[1] "order" "genus"
```

As with numeric indexing, the logical indices can be given as a logical vector. The logical indexing is a powerful and simple way to select some data from a vector: for instance, if we want to select the values greater than or equal to five in `x`:

```
> x >= 5
[1] TRUE FALSE FALSE TRUE TRUE
> x[x >= 5]
[1] 10 10 5
```

A useful function in this context is `which` that takes a logical vector as argument and returns the numeric indices of the values that are TRUE:

```
> which(x >= 5)
[1] 1 4 5
```

The *indexing system with names* brings us to introduce a new concept: a vector may have an attribute called *names* that is a vector of mode character of the same length, and serves as labels. It is created or extracted with the function `names`. An example could be:

```
> x <- 4:1
> names(x) <- z
> x
      order  family  genus species
         4        3      2      1
> names(x)
[1] "order"  "family"  "genus"  "species"
```

These names can then be used to select some elements of a vector:

```
> x[c("order", "genus")]
order genus
     4     2
```

In some situations it is useful to delete the names of a vector; this is done by giving them the value `NULL`:

```
> names(x) <- NULL
> x
[1] 4 3 2 1
```

### 2.3.2 Factor

A factor is a data structure derived from a vector, and is the basic way to store qualitative variables in R. It is of mode `numeric` and has an attribute `"levels"` which is a vector of mode character and specifies the possible values the factor can take. If a factor is created with the function `factor`, then the levels are defined with all values present in the data:

```
> f <- c("Male", "Male", "Male")
> f
[1] "Male" "Male" "Male"
> f <- factor(f)
> f
[1] Male Male Male
Levels: Male
```

To specify that other levels exist although they have not been observed in the present data, the option `levels` can be used:

```
> ff <- factor(f, levels = c("Male", "Female"))
> ff
[1] Male Male Male
Levels: Male Female
```



This is a crucial point when analyzing this kind of data, for instance, if we compute the frequencies in each category with the function `table`:

```
> table(f)
f
Male
  3
> table(ff)
ff
Male Female
  3      0
```

Factors can be indexed and have names exactly in the same way as vectors. When data are read from a file on the disk with the function `read.table`, the default is to treat all character strings as factors (see Chapter 3.8 for examples). This can be avoided by using the option `as.is = TRUE`.

### 2.3.3 Matrix

A matrix can be seen as a vector arranged in a tabular way. It is actually a vector with an additional attribute called *dim* (dimensions) which is itself a numeric vector with length 2, and defines the numbers of rows and columns of the matrix.

There are two basic ways to create a matrix: either by using the function `matrix` with the appropriate options `nrow` and `ncol`, or by setting the attribute `dim` of a vector:

```
> matrix(1:9, 3, 3)
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> x <- 1:9
> dim(x) <- c(3, 3)
> x
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

The numeric and logical indexing systems work in exactly the same way as for vectors. Because a matrix has two dimensions, it can be indexed with two integers separated by a comma:

```
> x[3, 2]
[1] 6
```

If one wants to extract only a row or a column, then the appropriate index must be omitted (without forgetting the comma):

```
> x[3, ] # extract the 3rd row
[1] 3 6 9
> x[, 2] # extract the 2nd column
[1] 4 5 6
```

In contrast to vectors, a subscript out of range results in an error.

R users often hit the problem that extracting a row or a column from a matrix returns a vector and not a single row, or single column, matrix. This is because the default behavior of the operator `[` is to return an object of the lowest possible dimension. This can be controlled with the option `drop` which is `TRUE` by default:

```
> x[3, 2, drop = FALSE]
      [,1]
[1,]      6
> x[3, , drop = FALSE]
      [,1] [,2] [,3]
[1,]      3      6      9
> x[, 2, drop = FALSE]
      [,1]
[1,]      4
[2,]      5
[3,]      6
```

Matrices do not have names in the same way as vectors, but have row-names, colnames, or both:

```
> rownames(x) <- c("A", "B", "C")
> colnames(x) <- c("v1", "v2", "v3")
> x
      v1 v2 v3
A      1  4  7
B      2  5  8
C      3  6  9
```

Selection of rows and / or columns follows in nearly the same ways as seen before:

```
> x[, "v1"]
A B C
1 2 3
> x["A", ]
v1 v2 v3
1  4  7
```

```
> x[c("A", "C"), ]
      v1 v2 v3
A     1  4  7
C     3  6  9
```

R has also a data structure called array which generalizes matrices to any number of dimensions.

### 2.3.4 Data Frame

A data frame is superficially similar to a matrix in the sense that it is a tabular representation of data. The distinction is that a data frame is a set of distinct vectors and / or factors all of the same length, but possibly of different modes.

Data frames are the main way to represent data sets in R because this corresponds roughly to a spreadsheet data structure. This is the type of objects returned by the function `read.table` (see Section 3.8 for examples). The other way to create data frames is with the function `data.frame`:

```
> DF <- data.frame(z, y = 0:3, 4:1)
> DF
      z y X4.1
1 order 0     4
2 family 1     3
3 genus 2     2
4 species 3     1
> rownames(DF)
[1] "1" "2" "3" "4"
> colnames(DF)
[1] "z"  "y"  "X4.1"
```

This example shows how colnames are created in different cases. By default, the rownames "1", "2", ... are given, but this can be changed with the option `row.names`, or modified subsequently as seen above for matrices.

If one of the vectors is shorter, then it is recycled along the data frame but this must be an integer number of times:

```
> data.frame(1:4, 9:10)
      X1.4 X9.10
1       1     9
2       2    10
3       3     9
4       4    10
> data.frame(1:4, 9:11)
Error in data.frame(1:4, 9:11) :
  arguments imply differing number of rows: 4, 3
```

All we have seen about indexing, `colnames`, and `rownames` for matrices apply in exactly the same way to data frames with the difference that `colnames` and `rownames` are mandatory for data frames. An additional feature of data frames is the possibility of extracting, modifying, or deleting, a column selectively with the operator `$`:

```
> DF$y
[1] 0 1 2 3
> DF$y <- NULL
> colnames(DF)
[1] "z"      "X4.1"
```

### 2.3.5 List

Lists are the most general data structure in R: they can contain any kind of objects, even lists. They can be seen as vectors where the elements can be any kind of object. They are built with the function `list`:

```
> L <- list(z = z, 1:2, DF)
> L
$z
[1] "order"  "family"  "genus"   "species"

[[2]]
[1] 1 2

[[3]]
      z y X4.1
1  order 0   4
2  family 1   3
3   genus 2   2
4 species 3   1

> length(L)
[1] 3
> names(L)
[1] "z" "" ""
```

The concepts we have seen on indexing vectors apply also to lists. Additionally, an element of a list may be extracted, or deleted, either with its index within double square brackets, or with the operator `$`:

```
> L[[1]]
[1] "order"  "family"  "genus"   "species"
> L$z
[1] "order"  "family"  "genus"   "species"
```

Note the subtle, but crucial, difference between `[[` and `[`:

```
> str(L[[1]])
chr [1:4] "order" "family" "genus" "species"
> str(L[1])
List of 1
 $ z: chr [1:4] "order" "family" "genus" "species"
```

## 2.4 Creating Graphics

The graphical functions in R need a special mention because they work somewhat differently from the others. A graphical function does not return an object (though there are a few exceptions), but sends its results to a *graphical device* which is either a graphical window (by default) or a graphical file. The graphical formats depend on the operating systems (see `?device` to see those available on your machine), but mostly the following are available: encapsulated PostScript (EPS), PDF, JPEG, and PNG. These are the most useful formats for scientific publication.

EPS and PDF are the publishers' preferred formats because they are line-based (or vectorial) so they can be scaled without affecting the resolution of the graphics. Choosing between these two formats is not always trivial because they have mostly the same capacities. The main weakness of EPS is that it cannot handle color transparency (see the option `alpha` in the function `rgb`) while PDF does. PDF files are usually more compact than the equivalent EPS files, but the latter are more easily modified (even by hand-editing since they are stored in simple text files). EPS files are easily converted into PDF with the 'epstoedit' utility provided with GhostScript.

JPEG and PNG are pixel-based (or raster) formats. They should be avoided for scientific publication because scaling-up graphics stored in such a format is likely to show the limit of resolution (an exception may be when a graphic contains a lot of elements so the EPS or PDF file may be too large). On the other hand, these two formats are well-suited for Web pages (PNG is usually better for line-graphics, whereas JPEG works better with pictures).

There are two ways to write graphics into a file. The most general and flexible way is to open the appropriate device explicitly, for instance, if we write into an EPS file:

```
postscript("plot.eps")
```

then all subsequent graphical commands will be written in the file 'plot.eps'. The operation is terminated (i.e., the file is closed and written on the disk) with the command:

```
dev.off()
```

The function `postscript` has many options to set the EPS files. All the figures of this book have been produced with this function. Similarly, for the other formats mentioned above, the function would be `pdf`, `jpeg`, or `png`.

The second way is to copy the content of the window device into a file using the function `dev.copy` where the user must specify the target device. Two variants of this function are `dev.copy2eps` and `dev.copy2pdf` which use EPS or PDF device (the file name must be specified by the user). Finally, `dev.print()` sends the current graphics to the printer. Under some operating systems, these commands can be called from the menus.

## 2.5 Saving and Restoring R Data

R uses two basic formats to save data: ASCII (simple text) and XDR (external data representation<sup>2</sup>). They are both cross-platform. The ASCII format is appropriate to save a single object (vector, matrix, or data frame) into a file. Two functions can be used: `write` (for vectors and matrices) and `write.table` (for data frames). The latter is very flexible and has many options. The XDR format can store any kind and any number of objects. It is used with the function `save`, for instance, to save three objects:

```
save(x, y, z, file = "xyz.RData")
```

These data can then be restored with:

```
load("xyz.RData")
```

Some care must be taken before calling `load` because if some objects named `x`, `y`, or `z` are present, the above command will erase them without any warning.

In practice, `.RData` files are useful when you must interrupt your session and you want to continue your work at a later time. In that case, it is better to use `save.image()` or quit R and choose to save the workspace image:

```
> q()
Save workspace image? [y/n/c]: y
```

In both cases a file named `'RData'` will be written in the current working directory will all objects in memory. Because these files may not be readable by many other programs, it is safe to keep your original data files (trees, molecular sequences, ...) in their own formats. However, an XDR file may be useful if you want to send a heterogeneous set of data to a colleague: in that case it will be easily loaded in memory with exactly the same attributes.

Apart from these two standard formats, the package `foreign`, installed by default with R, provides functions for reading and writing data files from a few common statistical computer programs. The CRAN has also some packages for reading a range of more or less specialized data formats (`netCDF`, `HDF5`,

---

<sup>2</sup> <http://www.faqs.org/rfcs/rfc1832.html>.

various GIS and map data formats, medical image formats, PDB for 3-D molecular structures, ...)

All commands typed in R's CLI are stored into memory and can be displayed with the command `history()`, saved into a file (named `‘.Rhistory’` by default) with `savehistory()`, or loaded into memory with `loadhistory()`.

## 2.6 Using R Functions

Now that we have seen a few instances of R function uses, we can draw some general conclusions on this point.

To execute a function, the parentheses are always needed, even if there is no argument inside (typing the name of a function without parentheses prints its contents). The arguments are separated with commas. There are two ways to specify arguments to a function: by their positions or by their names (also called *tagged arguments*). For example, let us consider a hypothetical function with three arguments:

```
fcn(arg1, arg2, arg3)
```

`fcn` can be executed without using the names `arg1`, ..., if the corresponding objects are placed in the correct position, for instance, `fcn(x, y, z)`. However, the position has no importance if the names of the arguments are used, for example, `fcn(arg3 = z, arg2 = y, arg1 = x)`. Another feature of R's functions is the possibility of using default values (also called *options*), for instance, a function defined as:

```
fcn(arg1, arg2 = 5, arg3 = FALSE)
```

Both commands `fcn(x)` and `fcn(x, 5, FALSE)` will have exactly the same result. Of course, tagged arguments can be used to change only some options, for instance, `fcn(x, arg3 = TRUE)`.

Many functions in R act differently with respect to the type of object given as arguments: these are called *generic* functions. They act with respect to an optional object attribute: the *class*. The main generic functions in R are `print`, `summary`, and `plot`. In R's terminology, `summary` is a generic function, whereas the functions that are effectively used (e.g., `summary.phylo`, `summary.default`, `summary.lm`, etc.) are called *methods*.

In practice, the use of classes and generics is implicit, but we show in the next chapter that different ways to code a tree in R correspond to different classes. The advantage of the generic functions here is that the same command is used for the different classes.

## 2.7 Repeating Commands

When it comes to repeating some analyses, several strategies can be used. The simplest way is to write the required commands in a file, and read them in

R with the function `source`. It is usual to name such files with the extension ‘.R’. For instance, the file ‘mytreeplot.R’ could be:

```
tree1 <- read.tree("tree1.tre")
postscript("tree1.eps")
plot(tree1)
dev.off()
```

These commands will be executed by typing `source("mytreeplot.R")` in R.

### 2.7.1 Loops

As with any language, R has control and programming structures to execute a series of commands. The most often-used one is the `for`<sup>3</sup> statement, whose general syntax is:

```
for (x in y) <command>
```

where `y` is an object, and `x` successively takes the different values of `y`. It is not required to use these values in `<command>` (e.g., `for (i in 1:5) print("done")`). A `for` loop may encompass more than one command in which case it is necessary to group them within braces:

```
for (x in y) {
  .....
  .....
}
```

`y` may be a vector of any mode, a factor (in which case the numerical coding will be used), a matrix (treated as a vector), a data frame (`x` will be substituted by the different columns of `y`), or a list (`x` will be substituted by the different elements of `y`).

Two commands may be useful here: `next` stops the current iteration and moves to the next value of `x`, and `break` aborts the loop. They are usually combined with an `if` statement which takes a single logical value as argument, for example:

```
for (i in 1:10) {
  if (x[i] < 0) break
  .....
}
```

---

<sup>3</sup> The following words are reserved to the R language and cannot be used to name objects: `for`, `in`, `if`, `else`, `while`, `next`, `break`, `repeat`, `function`, `NULL`, `NA`, `NaN`, `Inf`, `TRUE`, and `FALSE`.



### 2.7.2 *Apply*-Like Functions

In many situations, there is an easier and more efficient alternative to the use of loops and control statements: the *apply*-like functions. `apply` applies a function to all columns and / or rows of a matrix or a data frame. Its syntax is:

```
apply(X, MARGIN, FUN, ...)
```

where `X` is a matrix or a data frame; the second argument indicates whether to apply the function on the rows (1), the columns (2), or both (`c(1, 2)`); `FUN` is the function to be used; and `'...'` any argument that may be needed for `FUN`.

`lapply` does the same as `apply` but on different elements of a list. Its syntax is:

```
lapply(x, FUN, ...)
```

This function returns a list. `sapply` has nearly the same action as `lapply` but it returns its results as a more friendly way as a vector or a matrix with rownames and colnames. If a list is structured (i.e., made of lists of objects), `rapply` is a recursive version of `lapply`, applying `FUN` to the non-list elements. By default, the results are returned as a vector, unless the option `how = "replace"` is used.

`tapply` acts on a vector and applies a function on subsets defined by an additional argument `INDEX`:

```
tapply(X, INDEX, FUN = NULL, ...)
```

Typically, `INDEX` defines groups, and the function `FUN` is applied to each group. By default, the indices of the groups defined by `INDEX` are returned. `by` is a more elaborate function to apply a function `FUN` to the subsets of a vector or a data frame with respect to one or several factors given as a list; its syntax is:

```
by(data, INDICES, FUN, ..., simplify = TRUE)
```

`aggregate` does the same operation than `by` but returns the results in a table rather in a list.

Finally, `replicate` replicates a command a given number of times, returning the results as a vector, a matrix, or a list; for example,

```
> replicate(4, rnorm(1))
[1] -1.424699824  0.695066367  0.958153028  0.002594864
> replicate(5, rpois(3, 10))
      [,1] [,2] [,3] [,4] [,5]
[1,]    8   15   13   10   10
[2,]    5    8   14    3    6
[3,]   10   11    7    8    6
```

## 2.8 Exercises

1. Start R and print the current working directory. Suppose you want to read some data in three different files located in three different directories on your computer: describe two ways to do this.
2. Create a matrix with three columns and 1000 rows where each column contains a random variable that follows a Poisson distribution with rates 1, 5, and 10, respectively (see `?Poisson` for how to generate random Poisson values). Find two ways to compute the means of each column of this matrix.
3. Create a vector of 10 random normal values using the three following methods.
  - (a) Create and concatenate successively the 10 random values with `c`.
  - (b) Create a numeric vector of length 10 and change its values successively.
  - (c) Use the most direct method.

Compare the timings of these three methods (see `?system.time`) and explain the differences.

Repeat this exercise with 10,000 values.

4. Create the following text file:

Mus_musculus	10
Homo_sapiens	70000
Balaenoptera_musculus	120000000

- (a) Read this file with `read.table` using the default options. Look at the structure of the data frame and explain what happened. What option should have been used?
- (b) From this file, create a data structure with the numeric values that you could then index with the species names, for example,

```
> x["Mus_musculus"]
[1] 10
```

Find two ways to do this, and explain the differences in the final result.

5. Create these two vectors (source: [13]):

```
Archaea <- c("Crenarchaea", "Euryarchaea")
Bacteria <- c("Cyanobacteria", "Spirochaetes",
             "Acidobacteria")
```

- (a) Create a list named `TreeOfLife` so that we can do `TreeOfLife$Archaea` to print the corresponding group.
- (b) Update `TreeOfLife` by adding the following vector:

```
Eukaryotes <- c("Alveolates", "Cercozoa", "Plants",  
               "Opisthokonts")
```

It should appear at the same level as **Archaea** and **Bacteria**.

- (c) Update **Archaea** by adding "Actinobacteria".
- (d) Print all the lowest-level taxa.

<http://www.springer.com/978-1-4614-1742-2>

Analysis of Phylogenetics and Evolution with R  
Paradis, E.

2012, XIV, 386 p. 89 illus., Softcover

ISBN: 978-1-4614-1742-2