The digit 1 within brackets indicates that the display starts at the first element of `n`. This command is an implicit use of the function `print` and the above example is similar to `print(n)` (in some situations, the function `print` must be used explicitly, such as within a function or a loop).

The name of an object must start with a letter (A–Z and a–z) and can include letters, digits (0–9), dots (.), and underscores (_). R discriminates between uppercase letters and lowercase ones in the names of the objects, so that `x` and `X` can name two distinct objects (even under Windows).

2.2 Creating, listing and deleting the objects in memory

An object can be created with the "assign" operator which is written as an arrow with a minus sign and a bracket; this symbol can be oriented left-to-right or the reverse:

```
> n <- 15
> n
[1] 15
> 5 -> n
> n
[1] 5
> x <- 1
> X <- 10
> x
[1] 1
> X
[1] 10
```


If the object already exists, its previous value is erased (the modification affects only the objects in the active memory, not the data on the disk). The value assigned this way may be the result of an operation and/or a function:

```
> n <- 10 + 2
> n
[1] 12
> n <- 3 + rnorm(1)
> n
[1] 2.208807
```

The function `rnorm(1)` generates a normal random variate with mean zero and variance unity (p. 17). Note that you can simply type an expression without assigning its value to an object, the result is thus displayed on the screen but is not stored in memory:

```
> (10 + 2) * 5
[1] 60
```

The assignment will be omitted in the examples if not necessary for understanding.

The function `ls` lists simply the objects in memory: only the names of the objects are displayed.

```
> name <- "Carmen"; n1 <- 10; n2 <- 100; m <- 0.5
> ls()
[1] "m"    "n1"   "n2"   "name"
```

Note the use of the semi-colon to separate distinct commands on the same line. If we want to list only the objects which contain a given character in their name, the option `pattern` (which can be abbreviated with `pat`) can be used:

```
> ls(pat = "m")
[1] "m"    "name"
```

To restrict the list of objects whose names start with this character:

```
> ls(pat = "^m")
[1] "m"
```

The function `ls.str` displays some details on the objects in memory:

```
> ls.str()
m :  num 0.5
n1 :  num 10
n2 :  num 100
name :  chr "Carmen"
```


The option `pattern` can be used in the same way as with `ls`. Another useful option of `ls.str` is `max.level` which specifies the level of detail for the display of composite objects. By default, `ls.str` displays the details of all objects in memory, included the columns of data frames, matrices and lists, which can result in a very long display. We can avoid to display all these details with the option `max.level = -1`:

```
> M <- data.frame(n1, n2, m)
> ls.str(pat = "M")
M : 'data.frame':       1 obs. of  3 variables:
 $ n1: num 10
 $ n2: num 100
 $ m : num 0.5
> ls.str(pat="M", max.level=-1)
M : 'data.frame':       1 obs. of  3 variables:
```

To delete objects in memory, we use the function `rm`: `rm(x)` deletes the object `x`, `rm(x,y)` deletes both the objects `x` et `y`, `rm(list=ls())` deletes all the objects in memory; the same options mentioned for the function `ls()` can then be used to delete selectively some objects: `rm(list=ls(pat="^m"))`.

```
row.names, col.names, as.is = FALSE, na.strings = "NA",
colClasses = NA, nrows = -1,
skip = 0, check.names = TRUE, fill = !blank.lines.skip,
strip.white = FALSE, blank.lines.skip = TRUE,
comment.char = "#")
```

`file`	the name of the file (within `""` or a variable of mode character), possibly with its path (the symbol \ is not allowed and must be replaced by /, even under Windows), or a remote access to a file of type URL (http://...)
`header`	a logical (`FALSE` or `TRUE`) indicating if the file contains the names of the variables on its first line
`sep`	the field separator used in the file, for instance `sep="\t"` if it is a tabulation
`quote`	the characters used to cite the variables of mode character
`dec`	the character used for the decimal point
`row.names`	a vector with the names of the lines which can be either a vector of mode character, or the number (or the name) of a variable of the file (by default: `1`, `2`, `3`, ...)
`col.names`	a vector with the names of the variables (by default: `V1`, `V2`, `V3`, ...)
`as.is`	controls the conversion of character variables as factors (if `FALSE`) or keeps them as characters (`TRUE`); `as.is` can be a logical, numeric or character vector specifying the variables to be kept as character
`na.strings`	the value given to missing data (converted as `NA`)
`colClasses`	a vector of mode character giving the classes to attribute to the columns
`nrows`	the maximum number of lines to read (negative values are ignored)
`skip`	the number of lines to be skipped before reading the data
`check.names`	if `TRUE`, checks that the variable names are valid for R
`fill`	if `TRUE` and all lines do not have the same number of variables, "blanks" are added
`strip.white`	(conditional to `sep`) if `TRUE`, deletes extra spaces before and after the character variables
`blank.lines.skip`	if `TRUE`, ignores "blank" lines
`comment.char`	a character defining comments in the data file, the rest of the line after this character is ignored (to disable this argument, use `comment.char = ""`)

The variants of `read.table` are useful since they have different default values:

```
read.csv(file, header = TRUE, sep = ",", quote="\"", dec=".",
         fill = TRUE, ...)
read.csv2(file, header = TRUE, sep = ";", quote="\"", dec=",",
          fill = TRUE, ...)
read.delim(file, header = TRUE, sep = "\t", quote="\"", dec=".",
           fill = TRUE, ...)
read.delim2(file, header = TRUE, sep = "\t", quote="\"", dec=",",
            fill = TRUE, ...)
```

The function `read.fwf` can be used to read in a file some data in *fixed width format*:

```
read.fwf(file, widths, header = FALSE, sep = "\t",
         as.is = FALSE, skip = 0, row.names, col.names,
         n = -1, buffersize = 2000, ...)
```

The options are the same than for `read.table()` except `widths` which specifies the width of the fields (`buffersize` is the maximum number of lines read simultaneously). For example, if a file named data.txt has the data indicated on the right, one can read the data with the following command:

```
A1.501.2
A1.551.3
B1.601.4
B1.651.5
C1.701.6
C1.751.7
```

```
> mydata <- read.fwf("data.txt", widths=c(1, 4, 3))
> mydata
  V1   V2  V3
1  A 1.50 1.2
2  A 1.55 1.3
3  B 1.60 1.4
4  B 1.65 1.5
5  C 1.70 1.6
6  C 1.75 1.7
```


3.3 Saving data

The function `write.table` writes in a file an object, typically a data frame but this could well be another kind of object (vector, matrix, ...). The arguments and options are:

```
write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"))
```

`x`	the name of the object to be written
`file`	the name of the file (by default the object is displayed on the screen)
`append`	if `TRUE` adds the data without erasing those possibly existing in the file
`quote`	a logical or a numeric vector: if `TRUE` the variables of mode character and the factors are written within `""`, otherwise the numeric vector indicates the numbers of the variables to write within `""` (in both cases the names of the variables are written within `""` but not if `quote = FALSE`)
`sep`	the field separator used in the file
`eol`	the character to be used at the end of each line (`"\n"` is a carriage-return)
`na`	the character to be used for missing data
`dec`	the character used for the decimal point
`row.names`	a logical indicating whether the names of the lines are written in the file
`col.names`	id. for the names of the columns
`qmethod`	specifies, if `quote=TRUE`, how double quotes " included in variables of mode character are treated: if `"escape"` (or `"e"`, the default) each " is replaced by \", if `"d"` each " is replaced by ""

The option `byrow` indicates whether the values given by `data` must fill successively the columns (the default) or the rows (if `TRUE`). The option `dimnames` allows to give names to the rows and columns.

```
> matrix(data=5, nr=2, nc=2)
     [,1] [,2]
[1,]    5    5
[2,]    5    5
> matrix(1:6, 2, 3)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> matrix(1:6, 2, 3, byrow=TRUE)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Another way to create a matrix is to give the appropriate values to the dim attribute (which is initially `NULL`):

```
> x <- 1:15
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
> dim(x)
NULL
> dim(x) <- c(5, 3)
> x
     [,1] [,2] [,3]
[1,]    1    6   11
[2,]    2    7   12
[3,]    3    8   13
[4,]    4    9   14
[5,]    5   10   15
```


Data frame. We have seen that a data frame is created implicitly by the function `read.table`; it is also possible to create a data frame with the function `data.frame`. The vectors so included in the data frame must be of the same length, or if one of the them is shorter, it is "recycled" a whole number of times:

```
> x <- 1:4; n <- 10; M <- c(10, 35); y <- 2:4
> data.frame(x, n)
  x  n
1 1 10
2 2 10
```

tions which characterize the series. The options, with the default values, are:

```
ts(data = NA, start = 1, end = numeric(0), frequency = 1,
   deltat = 1, ts.eps = getOption("ts.eps"), class, names)
```

`data`	a vector or a matrix
`start`	the time of the first observation, either a number, or a vector of two integers (see the examples below)
`end`	the time of the last observation specified in the same way than `start`
`frequency`	the number of observations per time unit
`deltat`	the fraction of the sampling period between successive observations (ex. `1/12` for monthly data); only one of `frequency` or `deltat` must be given
`ts.eps`	tolerance for the comparison of series. The frequencies are considered equal if their difference is less than `ts.eps`
`class`	class to give to the object; the default is `"ts"` for a single series, and `c("mts", "ts")` for a multivariate series
`names`	a vector of mode character with the names of the individual series in the case of a multivariate series; by default the names of the columns of `data` or `Series 1`, `Series 2`, ...

A few examples of time series created with `ts`:

```
> ts(1:10, start = 1959)
Time Series:
Start = 1959
End = 1968
Frequency = 1
 [1]  1  2  3  4  5  6  7  8  9 10
> ts(1:47, frequency = 12, start = c(1959, 2))
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
1959       1   2   3   4   5   6   7   8   9  10  11
1960  12  13  14  15  16  17  18  19  20  21  22  23
1961  24  25  26  27  28  29  30  31  32  33  34  35
1962  36  37  38  39  40  41  42  43  44  45  46  47
> ts(1:10, frequency = 4, start = c(1959, 2))
     Qtr1 Qtr2 Qtr3 Qtr4
1959         1    2    3
1960    4    5    6    7
1961    8    9   10
> ts(matrix(rpois(36, 5), 12, 3), start=c(1961, 1), frequency=12)
         Series 1 Series 2 Series 3
```

4.4 Graphical parameters

In addition to low-level plotting commands, the presentation of graphics can be improved with graphical parameters. They can be used either as options of graphic functions (but it does not work for all), or with the function `par` to change permanently the graphical parameters, i.e. the subsequent plots will be drawn with respect to the parameters specified by the user. For instance, the following command:

```
> par(bg="yellow")
```

will result in all subsequent plots drawn with a yellow background. There are 73 graphical parameters, some of them have very similar functions. The exhaustive list of these parameters can be read with `?par`; I will limit the following table to the most usual ones.

`adj`	controls text justification with respect to the left border of the text so that `0` is left-justified, `0.5` is centred, `1` is right-justified, values > 1 move the text further to the left, and negative values further to the right; if two values are given (e.g., `c(0, 0)`) the second one controls vertical justification with respect to the text baseline
`bg`	specifies the colour of the background (e.g., `bg="red"`, `bg="blue"`; the list of the 657 available colours is displayed with `colors()`)
`bty`	controls the type of box drawn around the plot, allowed values are: `"o"`, `"l"`, `"7"`, `"c"`, `"u"` ou `"]"` (the box looks like the corresponding character); if `bty="n"` the box is not drawn
`cex`	a value controlling the size of texts and symbols with respect to the default; the following parameters have the same control for numbers on the axes, `cex.axis`, the axis labels, `cex.lab`, the title, `cex.main`, and the sub-title, `cex.sub`
`col`	controls the colour of symbols; as for `cex` there are: `col.axis`, `col.lab`, `col.main`, `col.sub`
`font`	an integer which controls the style of text (`1`: normal, `2`: italics, `3`: bold, `4`: bold italics); as for `cex` there are: `font.axis`, `font.lab`, `font.main`, `font.sub`
`las`	an integer which controls the orientation of the axis labels (`0`: parallel to the axes, `1`: horizontal, `2`: perpendicular to the axes, `3`: vertical)
`lty`	controls the type of lines, can be an integer (`1`: solid, `2`: dashed, `3`: dotted, `4`: dotdash, `5`: longdash, `6`: twodash), or a string of up to eight characters (between `"0"` and `"9"`) which specifies alternatively the length, in points or pixels, of the drawn elements and the blanks, for example `lty="44"` will have the same effet than `lty=2`
`lwd`	a numeric which controls the width of lines
`mar`	a vector of 4 numeric values which control the space between the axes and the border of the graph of the form `c(bottom, left, top, right)`, the default values are `c(5.1, 4.1, 4.1, 2.1)`
`mfcol`	a vector of the form `c(nr,nc)` which partitions the graphic window as a matrix of `nr` lines and `nc` columns, the plots are then drawn in columns (see section 4.1.2)
`mfrow`	id. but the plots are then drawn in line (see section 4.1.2)
`pch`	controls the type of symbol, either an integer between 1 and 25, or any single character within `""` (Fig. 2)
`ps`	an integer which controls the size in points of texts and symbols

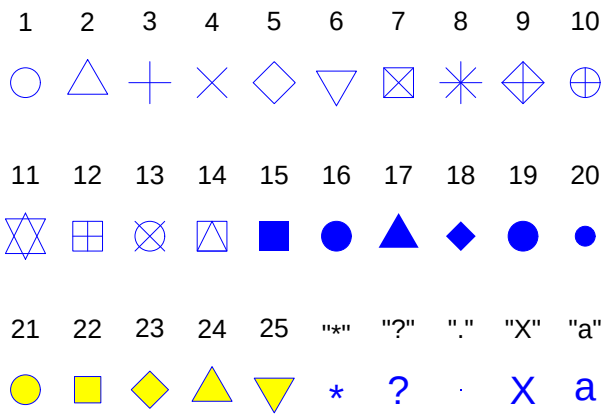

Figure 2: The plotting symbols in R (`pch=1:25`). The colours were obtained with the options `col="blue", bg="yellow"`, the second option has an effect only for the symbols 21–25. Any character can be used (`pch="*"`, `"?"`, `"."`, ...).

`pty`	a character which specifies the type of the plotting region, `"s"`: square, `"m"`: maximal
`tck`	a value which specifies the length of tick-marks on the axes as a fraction of the smallest of the width or height of the plot; if `tck=1` a grid is drawn
`tcl`	id. but as a fraction of the height of a line of text (by default `tcl=-0.5`)
`xaxt`	if `xaxt="n"` the x-axis is set but not drawn (useful in conjunction with `axis(side=1, ...)`)
`yaxt`	if `yaxt="n"` the y-axis is set but not drawn (useful in conjunction with `axis(side=2, ...)`)

4.5 A practical example

In order to illustrate R's graphical functionalities, let us consider a simple example of a bivariate graph of 10 pairs of random variates. These values were generated with:

```
> x <- rnorm(10)
> y <- rnorm(10)
```

The wanted graph will be obtained with `plot()`; one will type the command:

```
> plot(x, y)
```

and the graph will be plotted on the active graphical device. The result is shown on Fig. 3. By default, R makes graphs in an "intelligent" way:

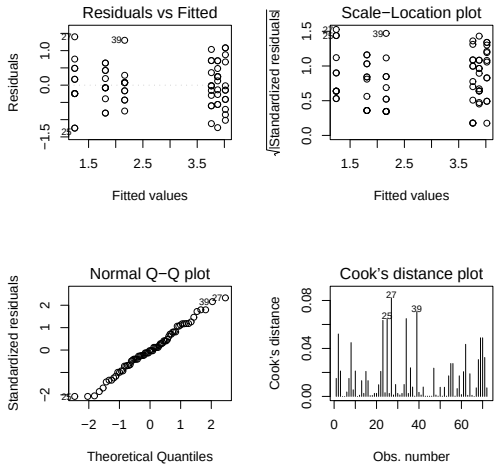

Figure 13: Graphical representation of the results from the function `aov` with `plot()`.

`a+b`	additive effects of `a` and of `b`
`X`	if `X` is a matrix, this specifies an additive effect of each of its columns, i.e. `X[,1]+X[,2]+...+X[,ncol(X)]`; some of the columns may be selected with numeric indices (e.g., `X[,2:4]`)
`a:b`	interactive effect between `a` and `b`
`a*b`	additive and interactive effects (identical to `a+b+a:b`)
`poly(a, n)`	polynomials of `a` up to degree n
`^n`	includes all interactions up to level n, i.e. `(a+b+c)^2` is identical to `a+b+c+a:b+a:c+b:c`
`b %in% a`	the effects of `b` are nested in `a` (identical to `a+a:b`, or `a/b`)
`-b`	removes the effect of `b`, for example: `(a+b+c)^2-a:b` is identical to `a+b+c+a:c+b:c`
`-1`	`y~x-1` is a regression through the origin (id. for `y~x+0` or `0+y~x`)
`1`	`y~1` fits a model with no effects (only the intercept)
`offset(...)`	adds an effect to the model without estimating any parameter (e.g., `offset(3*x)`)

We see that the arithmetic operators of R have in a formula a different meaning than they have in expressions. For example, the formula `y~x1+x2` defines the model $y = \beta_1 x_1 + \beta_2 x_2 + \alpha$, and not (if the operator `+` would have its usual meaning) $y = \beta(x_1 + x_2) + \alpha$. To include arithmetic operations in a formula, we can use the function `I`: the formula `y~I(x1+x2)` defines the model $y = \beta(x_1 + x_2) + \alpha$. Similarly, to define the model $y = \beta_1 x + \beta_2 x^2 + \alpha$, we will use the formula `y ~ poly(x, 2)` (and not `y ~ x + x^2`). However, it is

Package	Description
base	base R functions
datasets	base R datasets
grDevices	graphics devices for base and grid graphics
graphics	base graphics
grid	grid graphics
methods	definition of methods and classes for R objects and programming tools
splines	regression spline functions and classes
stats	statistical functions
stats4	statistical functions using S4 classes
tcltk	functions to interface R with Tcl/Tk graphical user interface elements
tools	tools for package development and administration
utils	R utility functions

Many *contributed* packages add to the list of statistical methods available in R. They are distributed separately, and must be installed and loaded in R. A complete list of the contributed packages, with descriptions, is on the CRAN Web site[18]. Several of these packages are *recommanded* since they cover statistical methods often used in data analysis. The recommended packages are often distributed with a base installation of R. They are briefly described in the following table.

Package	Description
boot	resampling and bootstrapping methods
class	classification methods
cluster	clustering methods
foreign	functions for reading data stored in various formats (S3, Stata, SAS, Minitab, SPSS, Epi Info)
KernSmooth	methods for kernel smoothing and density estimation (including bivariate kernels)
lattice	Lattice (Trellis) graphics
MASS	contains many functions, tools and data sets from the libraries of "Modern Applied Statistics with S" by Venables & Ripley
mgcv	generalized additive models
nlme	linear and non-linear mixed-effects models
nnet	neural networks and multinomial log-linear models
rpart	recursive partitioning
spatial	spatial analyses ("kriging", spatial covariance, ...)
survival	survival analyses

[18] http://cran.r-project.org/src/contrib/PACKAGES.html

```
> z <- x + y
```

This addition could be written with a loop, as this is done in most languages:

```
> z <- numeric(length(x))
> for (i in 1:length(z)) z[i] <- x[i] + y[i]
```

In this case, it is necessary to create the vector `z` beforehand because of the use of the indexing system. We realize that this explicit loop will work only if `x` and `y` are of the same length: it must be changed if this is not true, whereas the first expression will work in all situations.

The conditional executions (`if ... else`) can be avoided with the use of the logical indexing; coming back to the above example:

```
> y[x == b] <- 0
> y[x != b] <- 1
```

In addition to being simpler, vectorized expressions are computationally more efficient, particularly with large quantities of data.

There are also several functions of the type 'apply' which avoids writing loops. `apply` acts on the rows and/or columns of a matrix, its syntax is `apply(X, MARGIN, FUN, ...)`, where `X` is a matrix, `MARGIN` indicates whether to consider the rows (`1`), the columns (`2`), or both (`c(1, 2)`), `FUN` is a function (or an operator, but in this case it must be specified within brackets) to apply, and `...` are possible optional arguments for `FUN`. A simple example follows.


```
> x <- rnorm(10, -5, 0.1)
> y <- rnorm(10, 5, 2)
> X <- cbind(x, y) # the columns of X keep the names "x" and "y"
> apply(X, 2, mean)
        x         y
-4.975132  4.932979
> apply(X, 2, sd)
        x         y
0.0755153 2.1388071
```

`lapply()` acts on a list: its syntax is similar to `apply` and it returns a list.

```
> forms <- list(y ~ x, y ~ poly(x, 2))
> lapply(forms, lm)
[[1]]

Call:
FUN(formula = X[[1]])

Coefficients:
```

The word "*enclosing*" above is important. In our two example functions, there are *two* environments: the global one, and the one of the function `foo` or `foo2`. If there are three or more nested environments, the search for the objects is made progressively from a given environment to the enclosing one, and so on, up to the global one.

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 function with three arguments:

```
foo <- function(arg1, arg2, arg3) {...}
```

`foo()` can be executed without using the names `arg1`, ..., if the corresponding objects are placed in the correct position, for instance: `foo(x, y, z)`. However, the position has no importance if the names of the arguments are used, e.g. `foo(arg3 = z, arg2 = y, arg1 = x)`. Another feature of R's functions is the possibility to use default values in their definition. For instance:

```
foo <- function(arg1, arg2 = 5, arg3 = FALSE) {...}
```

The commands `foo(x)`, `foo(x, 5, FALSE)`, and `foo(x, arg3 = FALSE)` will have exactly the same result. The use of default values in a function definition is very useful, particularly when used with tagged arguments (i.e. to change only one default value such as `foo(x, arg3 = TRUE)`).

To conclude this section, let us see another example which is not purely statistical, but it illustrates the flexibility of R. Consider we wish to study the behaviour of a non-linear model. Ricker's model defined by:

$$N_{t+1} = N_t \exp\left[r\left(1 - \frac{N_t}{K}\right)\right]$$

This model is widely used in population dynamics, particularly of fish. We want, using a function, to simulate this model with respect to the growth rate r and the initial number in the population N_0 (the carrying capacity K is often taken equal to 1 and this value will be taken as default); the results will be displayed as a plot of numbers with respect to time. We will add an option to allow the user to display only the numbers in the last few time steps (by default all results will be plotted). The function below can do this numerical analysis of Ricker's model.

```
ricker <- function(nzero, r, K=1, time=100, from=0, to=time)
{
    N <- numeric(time+1)
    N[1] <- nzero
    for (i in 1:time) N[i+1] <- N[i]*exp(r*(1 - N[i]/K))
    Time <- 0:time
    plot(Time, N, type="l", xlim=c(from, to))
}
```