General Instructions in **R**

Rita Gaio, DM/FCUP

The R language

R is an open-source programming language that brings together features and tools to support graphs and statistical analyses.

- created in 1995 by **R**oss Ihaka and **R**obert Gentleman, Professors at the University of Auckland, New Zealand.
- based on two pre-existing statistical analysis computer languages, **S** and **Scheme**, but, contrary to those, **R** is of public domain
- The name "R" results from the similarity with **S**, and from the initials of its creators
- object-oriented language (there are objects in the program memory):

  'Everything in **R** is an object'.

- functional language (we ask functions to act on objects)
- formal computational language (allows great flexibility in building new *ad hoc* statistical models by comparison with other systems that have simpler interfaces, in terms of menus and formats)
- allows for the performance of statistical analyses and graphics building
- open-source software (anyone can contribute) with a list of available, add-on packages that provide additional functionalities
- freely available at the **R CRAN** site

  http://cran.r-project.org

  for Microsoft Windows XP and Vista, Unix e Linux platforms, and Apple Macintosh.

R Download

Download should be done directly from the oficial site http://cran.r-project.org/ following these steps:

1. Click on *Download R for Windows* or *Download R for (Mac) OS X*, according to pc/laptop.
2. Click on *base*.
3. Click on *Download R x.xx.x for Windows*. The series of xx's identifies the **R** version. (eg 3.6.3).
4. After clicking on the download link, choose a site on the computer to save the file. In the Windows system, it is usual to choose C:\Program Files.

R Installation

1. Double click on the **R** execution file
2. Select the language you want to use and click OK
3. Click on the "Next" > button
4. Check the provided information and click on the "Next" > button
5. Select the folder where you wish to install **R** and click on the "Next" > button
6. Click on the "Next" > button
7. Select No (accept defaults)
8. Choose where to place a program shortcut (by default, the shortcut is created in Menu → Start) and click on the "Next" > button
9. In the "Select additional tasks" window, click on the "Next" button >
10. Click on the "Conclude" button

R Updating

The best way to upgrade **R** is to uninstall the previous version of **R**, then install the new version and copy the old installed packages to the library folder of the new installation.

Command update.packages(checkBuilt = TRUE, ask = FALSE) will update the packages for the new installation. Afterwards, any remaining data from the old installation can be deleted.

Old versions of the software may be kept due to the parallel structure of the folders of the different installations. In cases where the user has a personal library, the contents must be copied into an update folder before running the update of the packages.

R packages

The **R**-language contains different features and functions organized in **packages**, or **libraries**. To activate each of these features we have to load the corresponding package:

```
library(.....)
```

If the package is not installed yet, you will get an error message:

```
Error in library(.....) :
  there is no package called '.....'
```

The package installation can be done in the following way:

- in **RStudio**:

  Tools → Install Packages ...

  Write down the name of the package and click "Install".

- in the **R**-console:

  Packages → Install Package(s) → choose a site (eg, the portuguese) → choose the package

- from the site <http://cran.r-project.org/>:

  choose "R binaries", in the column at the left → windows → contrib → choose your installed **R** version → find the required package → save destination as... (in the **R** *library* folder).

Usually, each package has a pdf tutorial, named *vignette*, created by the package authors. You have access to this pdf file from

<http://cran.r-project.org/>

or directly from the command line

```
browseVignettes("package_name")
```

RStudio

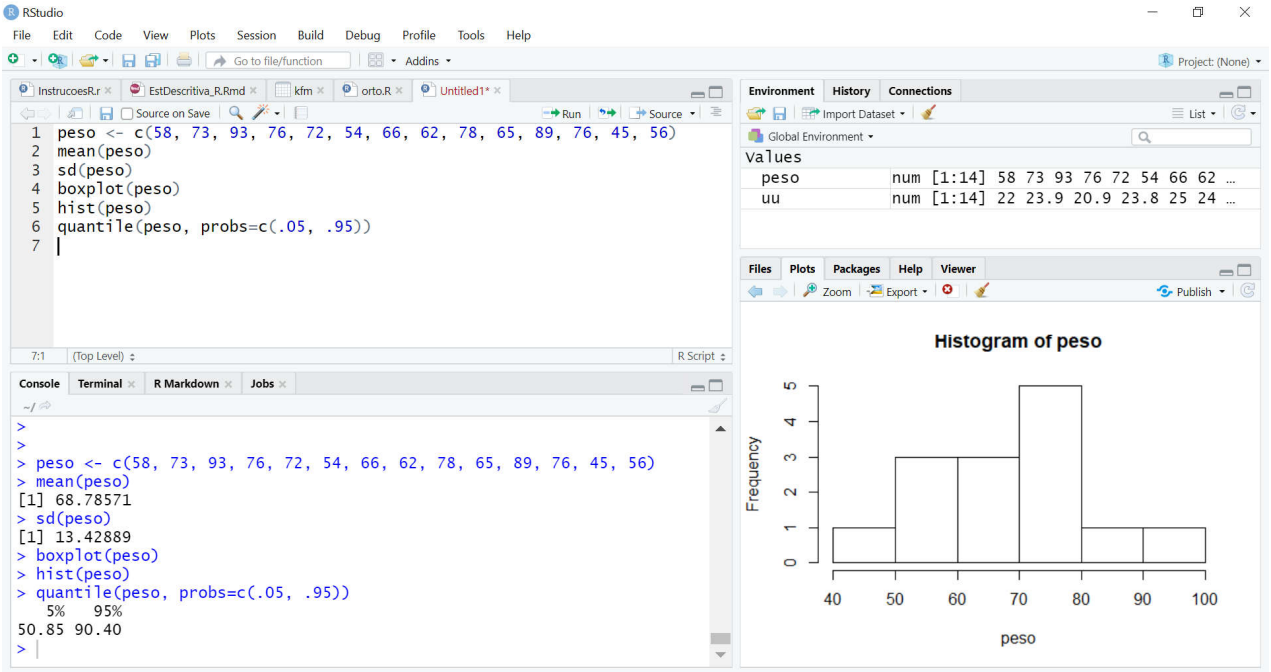
There are text editors that allow you to send selected instructions from a previously constructed file to the **R** console. One of those text editors is **RStudio**.

RStudio is a free open source editor developed for **R**. The download and all information about its running is available on the website

<http://www.rstudio.com/>.

Once opened, RStudio presents 4 windows (see image below):

- in the top left window, there is the file in which the instructions are written (and saved) - the **script**
- the window at the bottom left corresponds to the **console**, where instructions are executed. In most cases, F5 or Ctrl R send instructions from the script to the console, once the cursor is placed over the desired instruction.
- the top right window shows the objects that are in the current directory
- the window at the bottom right has several features: graphics (Plots), a Help window, and lists of available packages are the most used.



R: Basic Instructions

Remarks and/or annotations along a script, useful for further interpretation of that code, must be started with the # symbol.

Objects are accessed by name. The naming of objects (whether they are scalars, vectors, matrices, functions, ...) can contain letters, digits and/or periods (.) but the first character must not be a point, digit or symbol. For example, ".test" or "% test" should not be used.

Care must be taken with the names assigned since **R** is case-sensitive - distinguishes between upper and lowercase letters; for example, Weight $\neq$ weight.

The assignment of values or expressions to objects (variables, vectors, functions, ...) must be made using the symbol $<-$ (in most situations, the symbol $=$ can also be used). For example:

```
x <- 5
x
```

```
## [1] 5
```

```
y <- c(3.2, 1, 2.54)
y
```

```
## [1] 3.20 1.00 2.54
```

```
height <- c(1.76, 1.65, 1.54, 1.72)
height
```

```
## [1] 1.76 1.65 1.54 1.72
```

The function "c", used in the definition of vectors, comes from the word *concatenate*.

A vector can also be defined by reading one entry at a time.

After writing

```
height <- scan()
```

the user should type all values of 'height' and Enter at the end. Those values will be stored in the workspace, as the 'height' vector.

Many instructions in **R** call previously defined functions, such as *c(...)*, *cat(...)*, *mean(...)*, *plot(...)*, *sqrt(...)*, ... **Note that functions act on objects through round brackets (...).**

Vectors can be:

(i) numeric: coordinates are real numbers

(ii) character: coordinates are character strings of varying length. These are normaly entered and printed surrounded by double quotes, but single quotes can be used.

(iii) logical: coordinates are logical (true or false) values - less used by the user explicitly but recurrently used by internal routines.

(iv) integer: coordinates of (signed) integers.

(v) complex: coordinates of complex numbers.

(vi) list: a vector of R objects.

There is another object - NULL - which represents nothing, the empty set.

All objects in **R** have **classes** and the class of an object describes what the object contains and what many standard functions do with it.

'Every object in **R** has a class.'

```
u=c(4, 2.1, 3.0, 4.25)
class(u)  #numeric vector
```

```
## [1] "numeric"
```

```
names3 <- c("Maria", "Luísa", "Joana")
class(names3)  #character vector
```

```
## [1] "character"
```

When we want the result of a character vector to be a sentence, and not a set of words separated by quotes, the *cat(...)* function should be used. Its arguments may include text and previously defined objects (see below). The "\n" gives the instruction for a line break.

```
cat("Do you like dark chocolate from \"Nestlé\" ?\n")
```

```
## Do you like dark chocolate from "Nestlé" ?
```

```
yog <- c(2,3)
cat("Luis eats", yog[1], "yogurts per day while Ana eats", yog[2], "\n", "And you?")
```

```
## Luis eats 2 yogurts per day while Ana eats 3
##  And you?
```

An alternative solution, but producing a sentence within quotes, is

```
paste("Ana eats", yog[2], "yogurts per day")
```

```
## [1] "Ana eats 3 yogurts per day"
```

The **choice of specific coordinates from a previously defined vector uses square brackets** with the following structure:

```
vetorname[coordinate.number/s]
```

For example,

```
weight <- c(47.1, 57.2, 58.3, 78.1, 65.2, 61.4)
weight
```

```
## [1] 47.1 57.2 58.3 78.1 65.2 61.4
```

```
weight[2]
```

```
## [1] 57.2
```

```
weight[2:4] #same as weight[c(2:4)] or weight[c(2,3,4)]
```

```
## [1] 57.2 58.3 78.1
```

```
weight[c(1,2,5)]
```

```
## [1] 47.1 57.2 65.2
```

```
weight[-3]  # removes 3rd coordinate from vector
```

```
## [1] 47.1 57.2 78.1 65.2 61.4
```

```
weight[-c(1,3,4)]
```

```
## [1] 57.2 65.2 61.4
```

```
weight[seq(1,6,2)]  #seq(1,6,2) is vector 1, 3, 5
```

```
## [1] 47.1 58.3 65.2
```

It is possible to multiply a vector by a scalar (multiplying each of its coordinates by the scalar) and to sum vectors of the same size (summing up coordinate values, one by one).

Suppose, for instance, that the amount (in g) of fatty and lean fish consumed by a particular individual on each of the 7 days of a week is recorded.

```
lean.fish <- c(120, 80, 0, 0, 0, 0, 100)
fatty.fish <- c(0, 100, 0, 0, 150, 0, 0)
```

The total amount of fish consumed by that person on each day of the week is

```
fish <- lean.fish+fatty.fish
fish
```

```
## [1] 120 180   0   0 150   0 100
```

Warning: **R** sums vectors regardless of their sizes. For that, the shorter vector is repeated the necessary number of times and sum is performed coordinate wise. The user receives a warning message. For example:

```
u <- c(1,0)
lean.fish
```

```
## [1] 120  80   0   0   0   0 100
```

```
u+lean.fish
```

```
## Warning in u + lean.fish: longer object length is not a multiple of shorter
## object length
```

```
## [1] 121  80   1   0   1   0 101
```

The total amount of fish consumed by the individual within a week is

```
sum(fish)
```

```
## [1] 550
```

The total amount of fish consumed by the individual within a month (4 weeks) is

```r
4*sum(fish)
```

```
## [1] 2200
```

It is also possible to "glue" vectors by rows or columns. In the previous example, we can do, for example,

```r
rbind(lean.fish, fatty.fish)
```

```
##            [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## lean.fish   120   80    0    0    0    0  100
## fatty.fish    0  100    0    0  150    0    0
```

or

```r
cbind(lean.fish, fatty.fish)
```

```
##      lean.fish fatty.fish
## [1,]       120          0
## [2,]        80        100
## [3,]         0          0
## [4,]         0          0
## [5,]         0        150
## [6,]         0          0
## [7,]       100          0
```

These instructions *rbind()* and *cbind()* derive, respectively, from the expressions *row bind* and *column bind*.

R Factors

In Statistics, **factors** are categorical variables that take on a finite number of values (corresponding to each of its categories). In **R**, factors are sets of labelled observations with a pre-defined set of labels, not all of which need to occur.

Examples:

- sex/gender (2 categories: female, male)
- case/control (2 categories: case, control)
- the degree of severity of a disease (3 categories: light, medium, high)
- the level of pain perceived by patients (4 categories: none, reduced, medium, high)

```r
sex <- factor(c("male", "male", "female", "male", "female"))
levels(sex)
```

```
## [1] "female" "male"
```

```r
sex[3:5]
```

```
## [1] female male   female
## Levels: female male
```

By default, the levels of a factor are sorted into alphabetical order.

Function 'relevel' can be used to change the ordering of the levels.

Using a factor indicates to many of statistical functions that this is a categorical variable, and so it is treated specially. Categories of a factor can be coded numerically or textually. For example, the variable representing the classification of individuals into case or control can take the values 0 (control) and 1 (case), or "control" and "case". **It is advisable that factors in R take only numeric values.**

To perform calculations and statistical analyses, it is essential to distinguish a factor represented numerically from a numeric variable that takes on the same values. For example, a variable that takes the values

$$0, \quad 1, \quad 2, \quad \text{or} \quad 3$$

can represent:

- the residence area of a portuguese population (1-North, 2-Centre, 3-South, 0-Madeira/Açores)
- the number of yogurts eaten per day, per individual (assuming that is is always below 4).

Now, although it makes sense to sum the number of yogurts, considering several days, it is nonsense to sum portuguese residence areas.

On the other hand, **by default, R classifies all variables taking on numerical values as numeric**; e.g.:

```
class(c(3,2,0,1))
```

```
## [1] "numeric"
```

Some useful informations:

- class(object) – returns the class of the object
- is.factor(variablename) – returns a logical value indicating whether the variable is a factor or not;
- is.numeric(variablename) – returns a logical value indicating whether the variable is numeric or not;
- variablename=as.factor(variablename) – requires that the variable in question be interpreted as a factor;
- levels(factorname) – returns the levels of a factor;
- levels(factorname)=c("Feminino", "Masculino") – defines or redefines the levels of a factor.

R Control Structures

if( )

if (condition) {instruction 1}

if (condition) {instruction 1} else {instruction 2}

Here, "condition" has to correspond to a single logical value.

```
mark <- c(12.4, 9.7, 9.2, 16.3, 17.4, 4.9)
result <- as.character()
if (mark[1]>10) {result[1]<- "approved"}
result
```

```
## [1] "approved"
```

```
# if (mark>10) {result<- "approved"} #doesn't run
```

ifelse( )

ifelse(logical condition vector, vector-yes, vector-no)

```
mark <- c(12.4, 9.7, 9.2, 16.3, 17.4, 8.1)
result <- ifelse(mark>=9.5, "approved", "not approved")
result
```

```
## [1] "approved"     "approved"     "not approved" "approved"     "approved"
## [6] "not approved"
```

The instruction 'ifelse' returns a vector of exactly the same length as the logical condition vector. Missing values in test give missing values in the result.

Check that the returned value has to be of the same size as the logical test vector:

```
result <- ifelse(mark>=9.5, c(mark, mark/20*100), "not approved")
result  #not works
```

```
## [1] "12.4"         "9.7"          "not approved" "16.3"         "17.4"
## [6] "not approved"
```

If such an output is required, one should use

```
ff <- function(x){if (x>=9.5){c(x, x/20*100)} else ("not approved")}
sapply(mark, ff)
```

```
## [[1]]
## [1] 12.4 62.0
##
## [[2]]
## [1]  9.7 48.5
##
## [[3]]
## [1] "not approved"
##
## [[4]]
## [1] 16.3 81.5
##
## [[5]]
## [1] 17.4 87.0
##
## [[6]]
## [1] "not approved"
```

for( )

for (variable in vector) {instructions}

```
mark <- c(12.4, 9.7, 9.2, 16.3, 17.4, 8.1)
n.students <- length(mark)
mark.100 <- numeric()
mark.integer <- integer()
# does not work: mark.cat <- factor()
mark.cat <- factor(levels=c(1,2,3),
labels=c("insuff", "suff", "good"))

for (i in 1:n.students){
mark.cat[i] <- ifelse(mark[i]<9.5, "insuff", ifelse(mark[i]>14, "good", "suff"))
mark.100[i] <- mark[i]/20*100
mark.integer[i] <- round(mark[i])
}
mark.cat
```

```
## [1] suff   suff   insuff good   good   insuff
## Levels: insuff suff good
```

```
mark.100
```

```
## [1] 62.0 48.5 46.0 81.5 87.0 40.5
```

```
mark.integer
```

```
## [1] 12 10  9 16 17  8
```

```
## equal to
mark.cat <- ifelse(mark<9.5, "insuff", ifelse(mark>14, "good", "suff"))
```

while( )

while(condition){instructions}

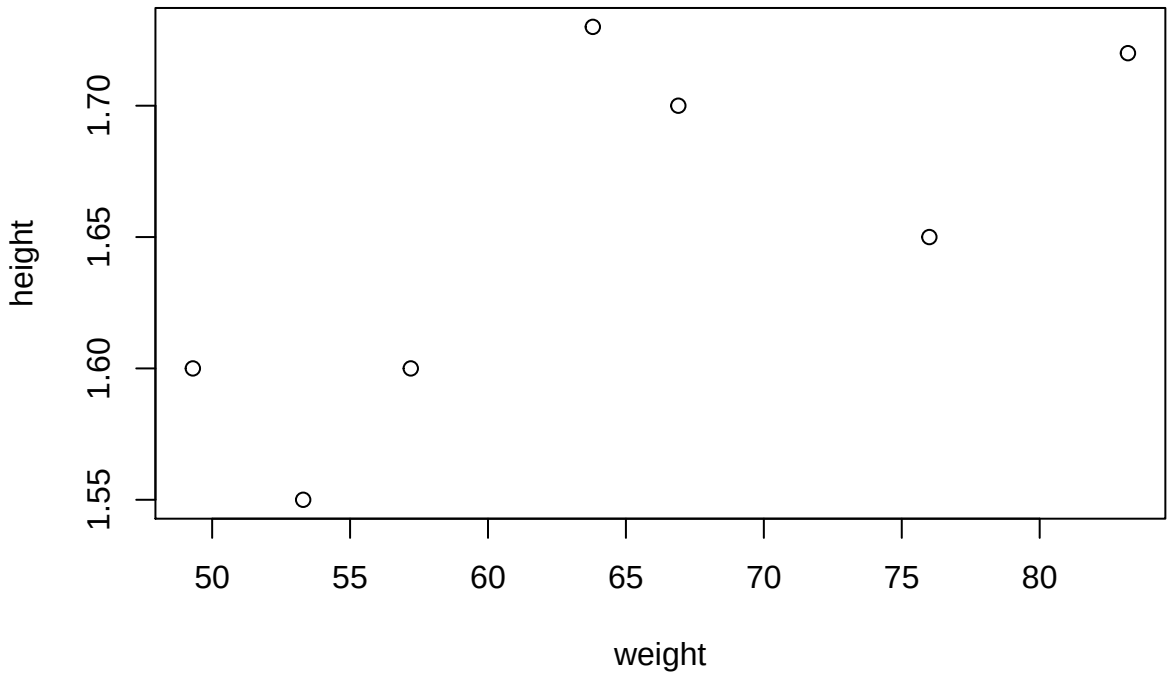
```
w<- 0
while(w<5){w<-w+1; print(w)}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Graphical System in R: basic notions

The simplest function for producing graphs in **R** is the **plot(...)** function.

```
weight <- c(76.0, 57.2, 49.3, 53.3, 83.2, 66.9, 63.8)
height <- c(1.65, 1.60, 1.60, 1.55, 1.72, 1.70, 1.73)
plot(weight, height)
```

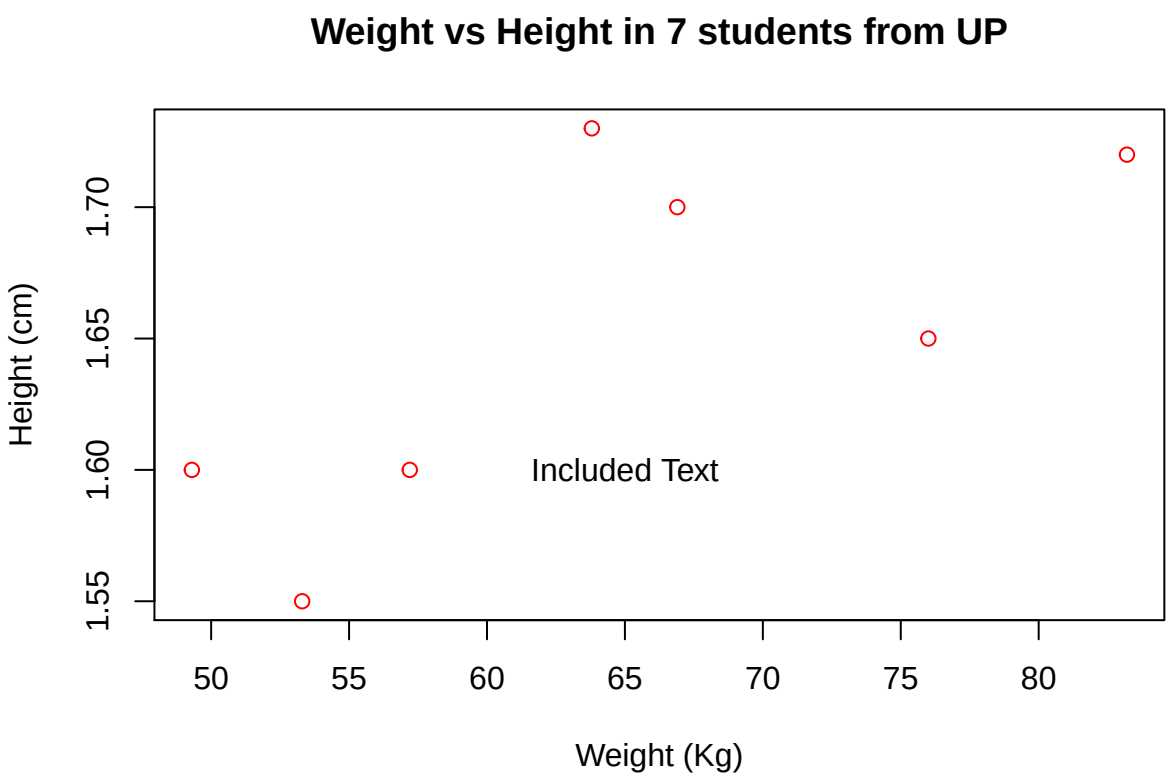


By default, the first variable in the plot instruction is represented in the x-axis while the second variable is represented on the y-axis. In doubt, one can specify the place of each variable in the graph, as in:

```
plot(y=altura, x=peso)
```

A more detailed and personalized graph would be, for instance, the following:

```
plot(weight, height,
 main="Weight vs Height in 7 students from UP",
 xlab="Weight (Kg)", ylab="Height (cm)",
 col="red")
text(65, 1.60, "Included Text")
```



The text has been centered at the coordinates (65, 1.60). It is also possible to change graphic parameters like the line thickness, line type (continuous, dashed, perforated, ...), character size and font, color, ... using the *par* function. Whenever this is necessary, consult the online help pages, namely through the following commands:

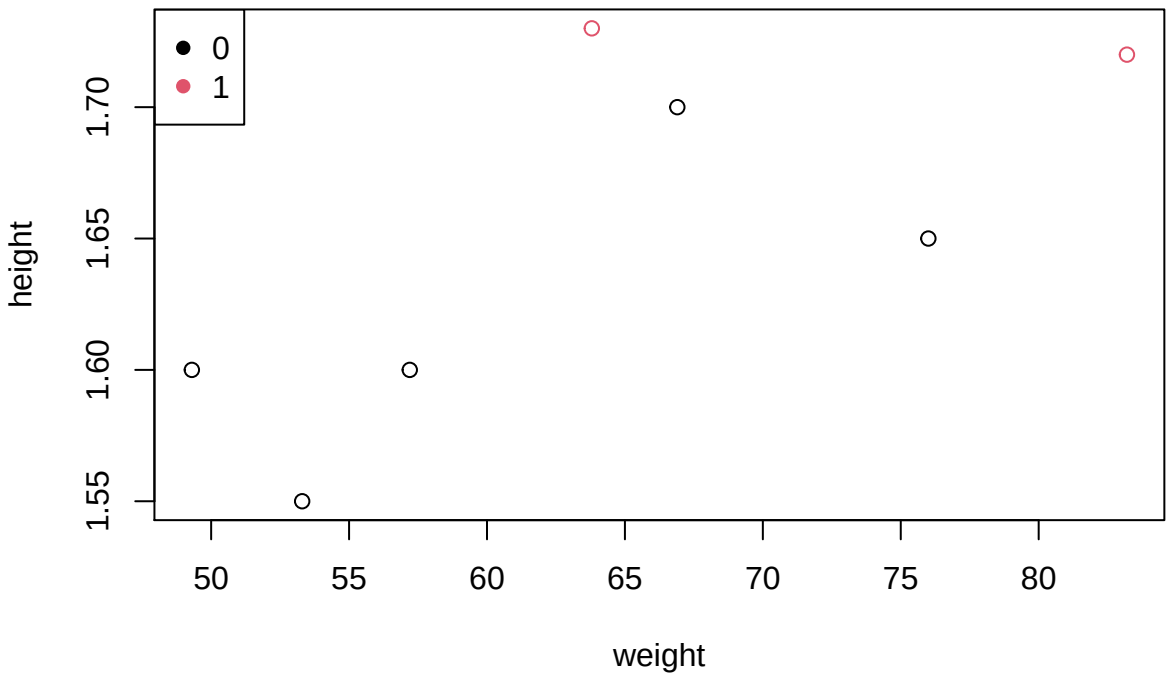
```
args(plot.default)
```

```
## function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
##     log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
##     ann = par("ann"), axes = TRUE, frame.plot = axes, panel.first = NULL,
##     panel.last = NULL, asp = NA, xgap.axis = NA, ygap.axis = NA,
##     ...)
## NULL
```

```
# ?plot.default  # opens page about plot
```

There could also be an interest in obtaining points with different colors depending on the levels of a factor. For example, if we know the gender of the individuals, we might want to have

```
sex <- c(0,0,0,0,1,0,1) #1=Man, 0=Woman
plot(weight, height, col=as.factor(sex))
legend("topleft", legend=levels(as.factor(sex)), pch=16, col=unique(as.factor(sex)))
```



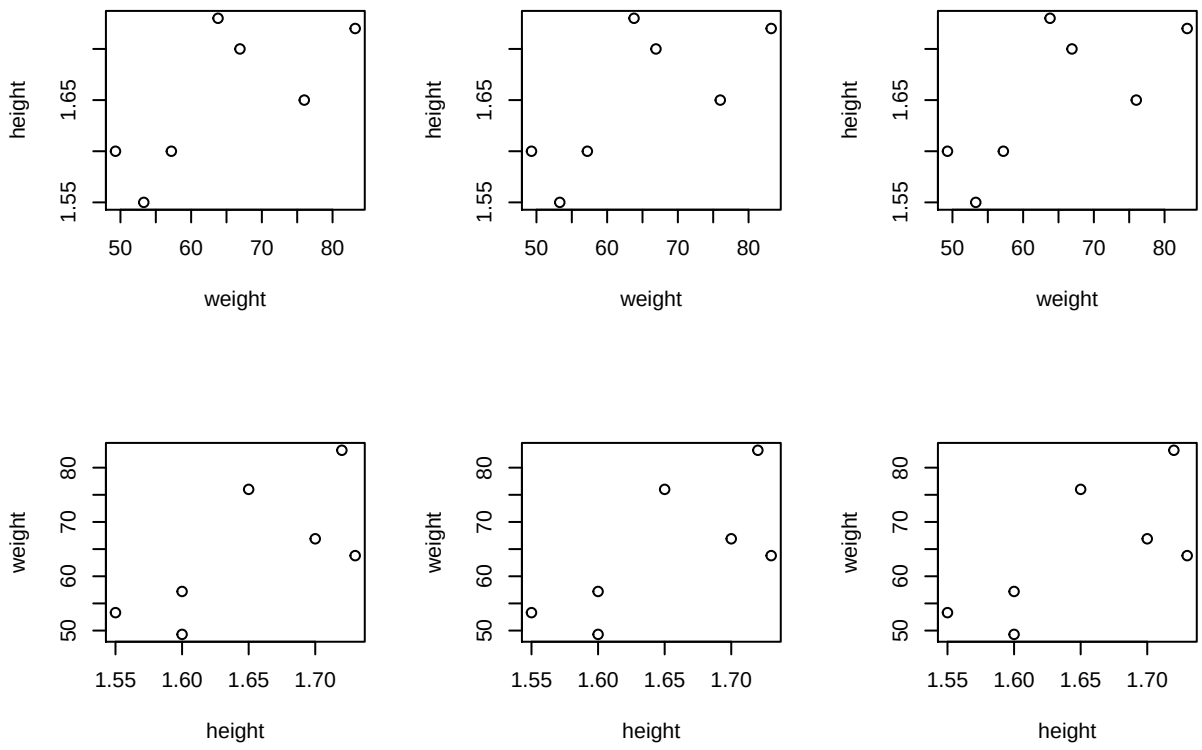

```
#pch: symbol to use
```

Note that here the black color corresponds to level 0 (women) and red corresponds to level 1 (men).

For more sophisticated graphs, explore the package **ggplot2**.

There are also situations in which you are interested in obtaining several plots on the same page. The following instruction organizes 6 graphs in a matrix of 2 rows by 3 columns:

```
par(mfrow=c(2,3))
plot(weight, height)
plot(weight, height)
plot(weight, height)
plot(height, weight)
plot(height, weight)
plot(height, weight)
```



R: Data frames

A **data frame** is an object consisting of rows and columns in which all lines are of the same length and data corresponding to the same line come from the same experimental unit (individual, for example). It is the type of object normally used in **R** to store a data matrix. In other statistical softwares, equivalent designations for data frame are *data matrix* or *data set*.

A data frame can be created from pre-existing variables (with the same length but possibly of different types) or by importing a file from a particular directory.

Suppose we want to create a data frame consisting of the variables weight and height, and corresponding to the values presented above for the 7 UP students:

```
weight <- c(76.0, 57.2, 49.3, 53.3, 83.2, 66.9, 63.8)
height <- c(1.65, 1.60, 1.60, 1.55, 1.72, 1.70, 1.73)
XX <- data.frame(weight, height)
XX
```

```
##   weight height
## 1   76.0   1.65
## 2   57.2   1.60
## 3   49.3   1.60
## 4   53.3   1.55
## 5   83.2   1.72
## 6   66.9   1.70
## 7   63.8   1.73
```

Manipulations of the observations in XX make use of matrix notation: the entry of XX that is in the ith row and jth column is $XX[i, j]$.

```
XX[3,1]
```

```
## [1] 49.3
```

```
XX[3,]
```

```
##   weight height
## 3   49.3    1.6
```

```
XX[,1]
```

```
## [1] 76.0 57.2 49.3 53.3 83.2 66.9 63.8
```

```
XX$weight
```

```
## [1] 76.0 57.2 49.3 53.3 83.2 66.9 63.8
```

```
XX$height[1:6]
```

```
## [1] 1.65 1.60 1.60 1.55 1.72 1.70
```

```
dim(XX)
```

```
## [1] 7 2
```

```
length(XX$weight)
```

```
## [1] 7
```

```
XX[-2,]
```

```
##   weight height
## 1   76.0   1.65
## 3   49.3   1.60
## 4   53.3   1.55
## 5   83.2   1.72
## 6   66.9   1.70
## 7   63.8   1.73
```

```
XX[,-2]
```

```
## [1] 76.0 57.2 49.3 53.3 83.2 66.9 63.8
```

```
XX[-c(3,6,7),]
```

```
##   weight height
## 1   76.0   1.65
## 2   57.2   1.60
## 4   53.3   1.55
## 5   83.2   1.72
```

```
XX[XX$height>1.60,]
```

```
##   weight height
## 1   76.0   1.65
## 5   83.2   1.72
## 6   66.9   1.70
## 7   63.8   1.73
```

```
XX$peso[XX$height>1.60]
```

```
## NULL
```

The columns of a data frame always have names and correspond to sample values of variables. When these names are not specified by the user, internal encoding $V1$, $V2$,.. is used.

```
names(XX)
```

```
## [1] "weight" "height"
```

Comparison operators and **logical operators** for comparisons:

- $<$ (less than), $>$ (greater than), $==$ (equal to)
- $<=$ (les than or equal to), $>=$ (greater than or equal to), $!=$ (different from)
- & (and), | (or), ! (no)

```
XX[XX$weight>80 | XX$weight<=55,]
```

```
##   weight height
## 3   49.3   1.60
## 4   53.3   1.55
## 5   83.2   1.72
```

```
XX[XX$height != 1.60 & XX$height <= 1.70,]
```

```
##   weight height
## 1   76.0   1.65
## 4   53.3   1.55
## 6   66.9   1.70
```

Analogous example but using factors as some of the variables:

```
BMI <- XX$weight/(XX$height^2)
XX$BMI #BMI does not exist within XX
```

```
## NULL
```

```
XXX <- data.frame(XX, BMI)
XXX    #now BMI does exist within XXX
```

```
##   weight height      BMI
## 1   76.0   1.65 27.91552
## 2   57.2   1.60 22.34375
## 3   49.3   1.60 19.25781
## 4   53.3   1.55 22.18522
## 5   83.2   1.72 28.12331
## 6   66.9   1.70 23.14879
## 7   63.8   1.73 21.31712
```

Now suppose we need to categorize BMI in the following way:

$$\begin{aligned} BMI \leq 20 &\implies BMI.F = 1 \\ 20 < BMI \leq 27 &\implies BMI.F = 2 \\ BMI > 27 &\implies BMI.F = 3 \end{aligned}$$

We can use several instructions:

```
BMI.cat <- ifelse(BMI<=20, 1, ifelse(BMI>27, 3, 2))
table(BMI.cat)
```

```
## BMI.cat
## 1 2 3
## 1 4 2
```

```
# or
BMI.F=rep(2,7)
for (i in 1:nrow(XXX)){
if (BMI[i] <= 20) BMI.F[i]=1
 else
if (BMI[i] > 27) BMI.F[i]=3
}
table(BMI.F)
```

```
## BMI.F
## 1 2 3
## 1 4 2
```

```
# or
BMI.c <- cut(BMI, breaks=c(0, 20, 27, max(BMI)))
levels(BMI.c)
```

```
## [1] "(0,20]"    "(20,27]"   "(27,28.1]"
```

```
levels(BMI.c)<-c("1", "2", "3")
table(BMI.c)
```

```
## BMI.c
## 1 2 3
## 1 4 2
```

```
# BMI.F, BMI.cat e BMI.c coincide!
```

Joining BMI.F to the data frame XXX:

```
XXX$BMI.F <- BMI.F
names(XXX)
```

```
## [1] "weight" "height" "BMI"    "BMI.F"
```

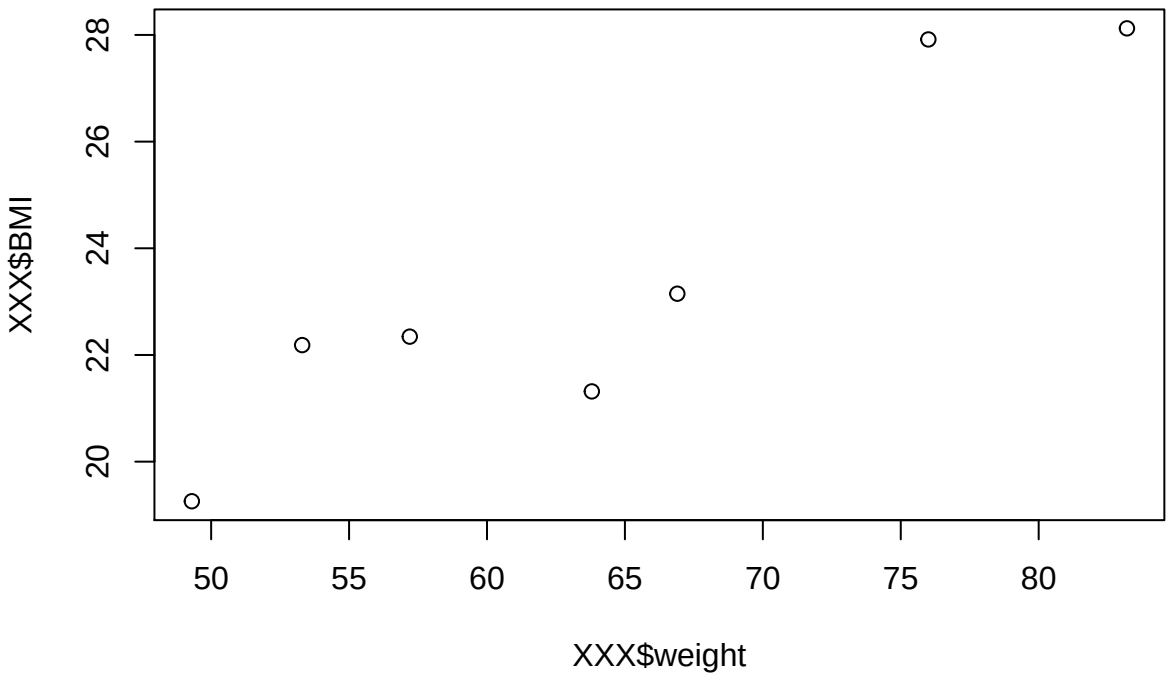
```
# weight e height of students with BMI.F=2:
XXX[XXX$BMI.F=="2",]
```

```
##   weight height      BMI BMI.F
## 2   57.2   1.60 22.34375     2
## 4   53.3   1.55 22.18522     2
## 6   66.9   1.70 23.14879     2
## 7   63.8   1.73 21.31712     2
```

```
# XXX[XXX$BMI.F="2",]     # error
# heights of students with BMI.F=2:
height.2 <- XXX$height[BMI.F=="2"]
height.2
```

```
## [1] 1.60 1.55 1.70 1.73
```

```
plot(XXX$weight, XXX$BMI)
```



It is also possible to re-designate the levels of a factor, for example by tranforming BMI.F into a text variable:

```
levels(BMI.F)=c("Low", "Medium", "High")
levels(BMI.F)
```

```
## [1] "Low"    "Medium" "High"
```

```
levels(XXX$BMI.F) #BMI.F does not exist in XXX
```

```
## NULL
```

Note that the variable that has been altered was the one in the workspace and not the one belonging to the data frame.

It is also possible to group the levels of a factor, defining a new factor with fewer levels. For example, consider the factor BMI.F12, only with two levels, defined by

$$\text{BMI.F12} = \begin{cases} 1 & \text{se BMI.F}= 1 \text{ or BMI.F}= 2 \\ 2 & \text{se BMI.F}= 3 \end{cases}$$

A possible instruction in **R** is

```
XXX$BMI.F12 <- ifelse(XXX$BMI.F==3, 2, 1)
table(XXX$BMI.F12)
```

```
##
## 1 2
## 5 2
```

```
table(XXX$BMI.F)
```

```
##
## 1 2 3
## 1 4 2
```

Sub-data frames can be defined choosing the appropriate variables and/or observations:

```
Y=XXX[,c(2:4)]
Y
```

```
##   height      BMI BMI.F
## 1   1.65 27.91552     3
## 2   1.60 22.34375     2
## 3   1.60 19.25781     1
## 4   1.55 22.18522     2
## 5   1.72 28.12331     3
## 6   1.70 23.14879     2
## 7   1.73 21.31712     2
```

```
Z=XXX[c(1:5),]
Z
```

```
##   weight height      BMI BMI.F BMI.F12
## 1   76.0   1.65 27.91552     3       2
## 2   57.2   1.60 22.34375     2       1
## 3   49.3   1.60 19.25781     1       1
## 4   53.3   1.55 22.18522     2       1
## 5   83.2   1.72 28.12331     3       2
```

```
W=XXX[c(4,7), c(2,3)]
W
```

```
##   height      BMI
## 4   1.55 22.18522
## 7   1.73 21.31712
```

Several Remarks:

1. Variables in a dataframe are only accessible through the instruction:

```
dataframe$variablename
```

For example:

```
XX$height2 <- XX$height^2
# >height2    error
```

For ease of notation, one can use *attach(dataframe)*. From then onwards, variables can already be called directly:

```
attach(XX)
```

```
## The following objects are masked _by_ .GlobalEnv:
##
##     height, weight
```

```
height2 #works already
```

```
## [1] 2.7225 2.5600 2.5600 2.4025 2.9584 2.8900 2.9929
```

The instruction attach(...) may give rise to some comments; it is convenient to know that *.GlobalEnv* is the workspace.

To stop accessing variables directly by their names, use *detach(dataframe)*:

```
detach(XX)
```

This instruction does not eliminate the data frame. It only implies that variables are not directly accessed by their names any longer.

2. List of all created objects:

```
ls()
```

```
##  [1] "BMI"          "BMI.c"        "BMI.cat"      "BMI.F"        "fatty.fish"
##  [6] "ff"           "fish"         "height"       "height.2"     "i"
## [11] "lean.fish"    "mark"         "mark.100"     "mark.cat"     "mark.integer"
## [16] "n.students"   "names3"       "result"       "sex"          "u"
## [21] "w"            "W"            "weight"       "x"            "XX"
## [26] "XXX"          "y"            "Y"            "yog"          "Z"
```

lists the objects (vectors, data frames, ...) in the *workspace*.

3. Removal of objects

One can use the instruction

```
rm(variablename)
```

Clicking on the brush symbol in the upper right box of RStudio removes **all** objects from the environment.

4. Identification of the working directory

```
getwd()
```

5. Help in **R**

If you know the name of the function for which more information is sought, just write ? followed by the function's name. For example, each of the instructions

?par or ?mean

opens a window with the description of the contents of the corresponding function.

When the name of the function is only partially known, it is worth trying

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

For example,

```
help.search("par") # opens a window listing all functions having 'par' in their designation
??"par"            # equivalent
```

6. Saving of the work

The procedure

File → Save History

allows saving all instructions used during a session to a file. To read those instructions again, one can use

File → Load History.

All session (with the results and created objects) can be saved in

File → Save Workspace

(extension .RData). and then recovered using

File → Load Workspace.

7. Pre-existing data frames

There are several data frames available in **R** (often used by library authors in their examples).

The instruction

```
data()
```

opens a window listing all existent data frames in the current loaded packages. Any data frame in the list can be directly accessed by its name.

R: Matrices

A data frame is not a matrix, although it may look like a matrix. Matrices, like vectors, have all their entries of the same type. Matrices are generated by the function 'matrix'. By default, all entries, are laid out down columns.