BioSys PhD | Earthsystems PhD

Statistics 1

Introduction to R

Lisete Sousa
lmsousa@fc.ul.pt
Room: 6.4.25

Departamento de Estatística e Investigação Operacional
Centro de Estatística e Aplicações

Ciências ULisboa

CEAUL
Centro de Estatística e Aplicações
Universidade de Lisboa

2018

Summary

1. Installing and Configuring R Studio

R Studio

R Studio *Website*

https://www.rstudio.com/products/rstudio/

The beggining of R:

CRAN - *Comprehensive R Archive Network*

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

Information

RStudio

rstudio::conf Products Resources Pricing About Us Blogs

Take control of your R code

RStudio is an integrated development environment (IDE) for R. It includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging and workspace management. Click here to see more RStudio features.

RStudio is available in open source and commercial editions and runs on the desktop (Windows, Mac, and Linux) or in a browser connected to RStudio Server or RStudio Server Pro (Debian/Ubuntu, RedHat/CentOS, and SUSE Linux).

CLICK HERE TO SEE ADDITIONAL FEATURES

Desktop

Run RStudio on your desktop

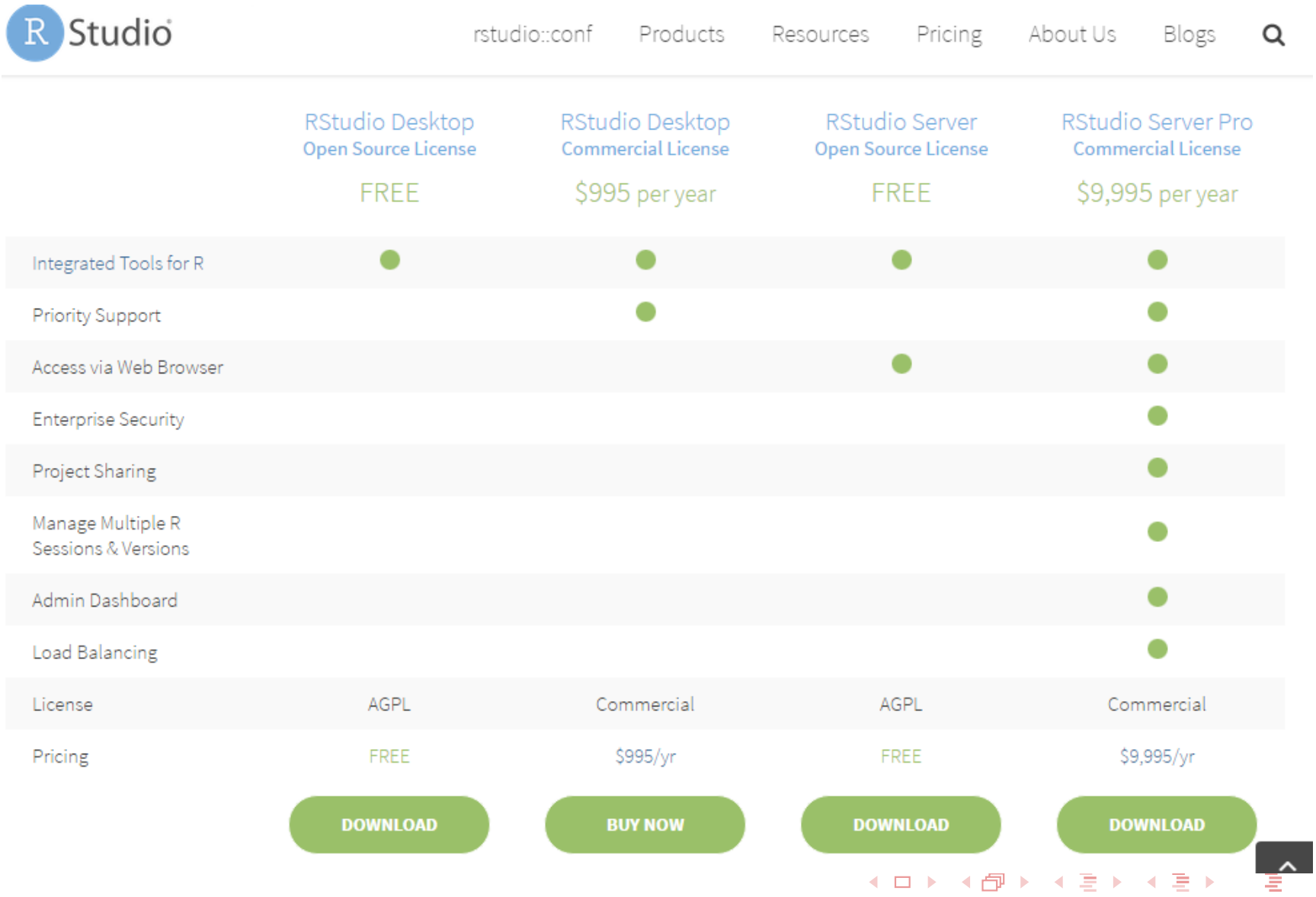
RStudio Desktop ›

Server

Centralize access and computation

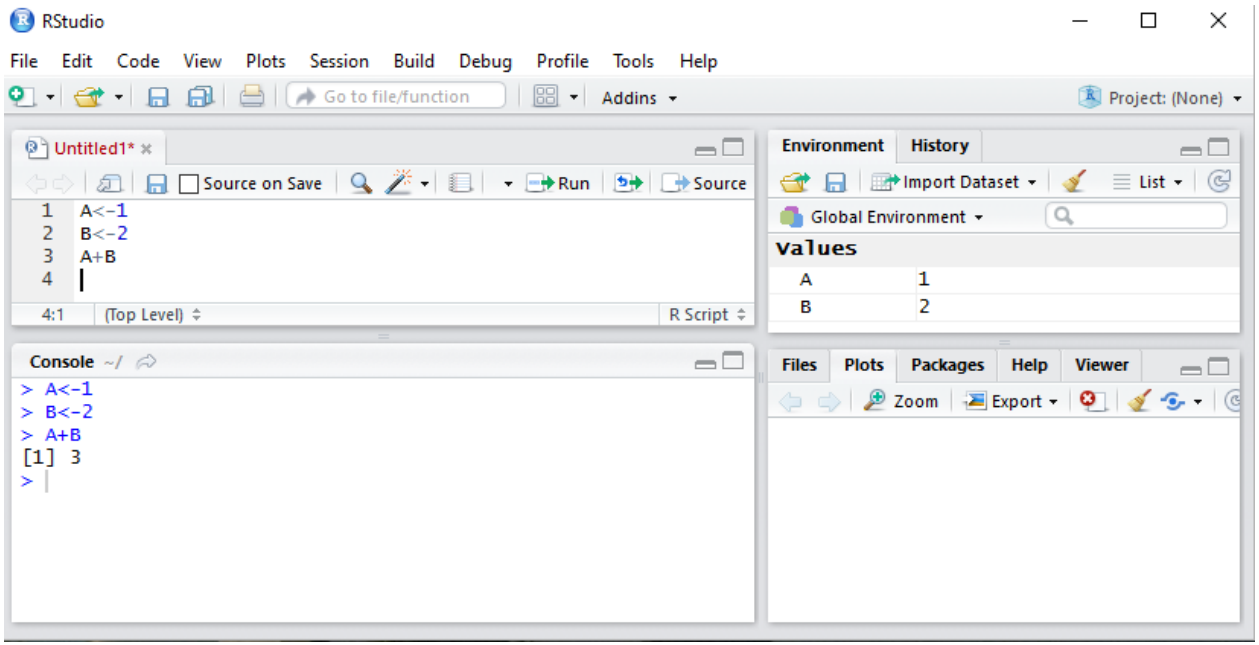
RStudio Server ›

Download


RStudio
rstudio::conf
Products
Resources
Pricing
About Us
Blogs
RStudio Desktop
Open Source License
FREE
RStudio Desktop
Commercial License
$995 per year
RStudio Server
Open Source License
FREE
RStudio Server Pro
Commercial License
$9,995 per year
Integrated Tools for R
Priority Support
Access via Web Browser
Enterprise Security
Project Sharing
Manage Multiple R Sessions & Versions
Admin Dashboard
Load Balancing
License
AGPL
Commercial
AGPL
Commercial
Pricing
FREE
$995/yr
FREE
$9,995/yr
DOWNLOAD
BUY NOW
DOWNLOAD
DOWNLOAD

Before using R Studio

In RConsole window we can observe a waiting alert command '>' that we call prompt. It is in this space that we edit command lines of R.



Type, after *prompt*, the command R.version followed by the Enter key. The R.version command when running sends information about the R version and the computer where R is running.

PRELIMINARY NOTES:

- R software differentiates the the lower and the upper case letters.
- Symbols like ',' , '.' , ';', have specific functions within the language.
- The assignment operator is '<-' or '='.
  Example: `x<-2` or `2->x`
  If an expression is used without attribution then the information is lost, that is, only one result is returned on the screen. Instead of the operator '<-', we can use the command `assign()`.
  Example: `assign('x',2)`

  The symbol $<-$ is the most widely used.

- Any expression preceded by the symbol # is not executed and is only informative.
  Example:

```
# the following operations allow
# the construction of a three-dimensional graph.
```

- The results (*output*) from the executed commands come after the straight brackets, with the line number [1].

WORKING DIRECTORY:

- It is recommended the use of different directories for the various analyses carried out in R. To know the directory you are working in, just type `getwd()`.
- To change the directory you may use the option
  `Session - Set Working Directory - Choose Directory`
  `[ctrl + shift + H]`
  in the menu, or use the function `setwd()`.

  Example:

  ```
  setwd("C:/My Documents/My.Docs")
  ```
  or
  ```
  setwd("C:\\My Documents\\My.Docs")
  ```

GETTING HELP IN R:

- Through the menu *Help* you will have access to manuals and other types of available help.
- Through the `help.start()` function, you will have access to a help system in *html* format.
- In order to get information about a particular function, you can use the function `help()`.

  Example:

  `> help(sqrt)` or `> ?sqrt`
- Go to HELP window and type *sqrt*.

R packages:

- Packages are aggregations of functions that were created by someone and made available to the community for free.
- Anyone can build a package and submit it to R.
- R already has a set of installed packages, however you can install additional packages available on the R webpage.

  For example, the package `lattice` has many functions for the construction of graphs.
  (1) To install this package go to PACKAGES window, press INSTALL and type *lattice*.
  (2) To access this package use: `> library(lattice)`

2. Basic Operations

Operators and Arithmetic Functions

R can operate as a calculator, having the following operators:

+	addition
-	subtraction
*	multiplication
%*%	matrix multiplication
/	division
^	exponentiation
sqrt()	square root
abs()	absolute value
log10()	logarithm to base 10
log(x,base=a)	logarithm of x, to base a
log()	natural logarithm
exp()	exponential function
pi	π
sin() cos() tan()	trigonometric functions
asin() acos() atang()	inverse trigonometric functions

Simple Calculations

See the following examples:

```
> 3+7
[1] 10
> 89*1234
[1] 109826
> 67/4
[1] 16.75
> 3^4
[1] 81
> log(2)
[1] 0.6931472
> log10(34)  # or log(34,base=10)
[1] 1.531479
> exp(5)
[1] 148.4132
> sin(pi)
[1] 1.224606e-16
> cos(pi)
[1] -1
> tan(pi/4)
[1] 1
> sqrt(36)+4^(1/2)
[1] 8
```

Function seq()

In R there are several functions to generate successions or sequences of numbers. For example `1:10`, generates a sequence of values from 1 to 10. The ':' operator takes precedence in an expression in which it is used.

Exercise

Create the object `n<-10` and compare the sequences `1:n-1` e `1:(n-1)`.

The expression `30:1` allows you to build a descending sequence.

Function seq()

Allows you to generate more complex sequences. The four most commonly used arguments are:

from=the starting value of the sequence

to=the last value of the sequence

by=number: increment of the sequence

length.out=desired length of the sequence

See the following examples:

```
> seq(1:10)
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(from=1,to=10)
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(to=10,from=1)
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(1,5,by=0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
> seq(from=2.3,by=0.2,length=20)
 [1] 2.3 2.5 2.7 2.9 3.1 3.3 3.5 3.7 3.9 4.1 4.3 4.5 4.7 4.9
[15] 5.1 5.3 5.5 5.7 5.9 6.1
```

Logical Vectors

The elements of a logical vector are **FALSE** and **TRUE** and can be abbreviated to **F** and **T**, respectively. The logical operators are:

<	-
>	-
<=	-
>=	-
==	exact equality
!=	inequality
a & b	intersection
a|b	union
!a	negation

Logical vectors allow the evaluation of conditions.

Examples:

```
> x<-190
> cholesterol <- x<200
> cholesterol
[1] TRUE
> mode(cholesterol)
[1] "logical"
> y <- x>150
> y
[1] TRUE
> y&cholesterol
[1] TRUE
> y|cholesterol
[1] TRUE
```

The function `cholesterol <- x>200` creates `cholesterol` vector with the length of **x**, whose elements are **FALSE** (not satisfying the condition) or **TRUE** (meeting the condition of being over 200).

Alphanumeric Vectors

- Alphanumeric or character vectors, are often used for example in tags (*labels*) of graphics.
- Are defined in double quotation marks ("xxx").
- The `paste()` function converts its arguments in *strings* separated by user-defined tab (`sep=' '`).

```
> paste("A",1:10)
 [1] "A 1"  "A 2"  "A 3"  "A 4"  "A 5"  "A 6"  "A 7"  "A 8"  "A 9"  "A 10"
> paste(c("A","B"),1:10,sep=)
 [1] "A1"  "B2"  "A3"  "B4"  "A5"  "B6"  "A7"  "B8"  "A9"  "B10"
> paste("Today is",date(),1:3,sep=",")
[1] "Today is,Sat Apr 19 01:36:23 2014,1"
[2] "Today is,Sat Apr 19 01:36:23 2014,2"
[3] "Today is,Sat Apr 19 01:36:23 2014,3"
```

3. Vectors, Matrices and Data Frames

Creating vectors in R

R is an object-based language. These objects are characterized by their names and by their contents. All objects have two attributes: mode and length.

mode: *numeric*, *character*, *complex* or *logical*

length: number of elements in the object

Perform the following operations in R:

```
x<-1
mode(x)
length(x)
A <- "Iron Level"
compar<-TRUE
z<-1i
mode(A); mode(compar); mode(z)
```

Vectors

- If the object has *missing data*, it is represented as `NA` (*not available*).
- A huge numerical value may be represented in scientific notation (e.g., 2.1e23).

Function c()

Generic function which combines its arguments (of the same type (mode)).

Exercises:

```
v<-c(3,7,9,15)
v
length(v)
mode(v)
s<-c("Name","Blood type","Pathology")
s
mode(s)
```

If a certain element is unknown:

```
> y<-c(35,56,NA,18,65)
```

Picking up a certain element:

```
> y[3]
```

element on the third position of the vector

Changing an element of the vector:

```
> y[4]<-70
```

the fourth element is substituted by 70

Exercises:

```
# Including an extra element
y[6]<-76
y

# Extracting elements
y<-c(y[2],y[4])
y

# Operations with vectors
v<-c(1,2,3,4,5,6)
x<-sqrt(v)
x
v1<-c(1,2,3)
v2<-c(4,5,6)
v1+v2
```

Exercise: Check what the following operations do.

```
x<-c(0,-3,2,7,9,15,20,-2,-1)
x
x>0
y<-x>0
x[y] # or x[x>0]
x[x<=-2|x>5]
x[x<15 & x>2]
x[c(3,7)]
x[1:3]
x[-c(2,8)]
x[-(1:3)]
```

Factors

Function factor()

Encoding a vector as a factor (for qualitative variables).

Sometimes it is preferable to keep the categorical information as factor instead of string.

```
# Let us assume we intend to save the results of a diagnostic
test of
# 10 patients
p<-c("sick","notsick","sick","notsick","sick",
"sick","sick","notsick","sick","sick")
p
p<-factor(p)
p
# Suppose now that all patients are females
gender.p<-factor(c("f","f","f","f","f","f","f","f","f","f"),
levels=c("f","m"))
gender.p
```

The factors allow us to use specific functions, such as counting the number of occurrences of each level. The function `table()` allows to obtain frequency tables for factors of equal length. If the factor has k levels, the result will be a table with the frequencies of each level.

Exercises:

```
table(p)
table(gender.p)
table(p,gender.p)
t<-table(p,gender.p)
margin.table(t,1)
margin.table(t,2)
prop.table(t)
100*prop.table(t)
```

Assume we have a sample of 15 health professionals from various provinces of Portugal. The vector of provinces contains the initials of each of the elements of this sample:

```
> province<-c("tmd","bl","min","rib","rib","ba","alt","alt","min","alg",
+ "rib","alg","min","min","ba")
> province
 [1] "tmd" "bl"  "min" "rib" "rib" "ba"  "alt" "alt" "min" "alg" "rib" "alg
[13] "min" "min" "ba"
> province.f<-factor(province)
> province.f
 [1] tmd bl  min rib rib ba  alt alt min alg rib alg min min ba \\
Levels: alg alt ba bl min rib tmd
> levels(province.f)
[1] "alg" "alt" "ba"  "bl"  "min" "rib" "tmd"
> length(province.f)
[1] 15
```

Assume now we have an array with the number of diabetes patients, attended in one week by each of the health professionals:

```
> nr.patients<-c(60,49,40,61,64,60,59,62,69,70,42,56,61,61,58)
> length(nr.patients)
[1] 15
```

To calculate the average number of patients with diabetes attended in one week in each of the provinces, we can use the function `tapply()`:

```
> mean.patients<-tapply(nr.patients,province.f,mean)
> mean.patients
     alg      alt       ba       bl      min      rib      tmd
63.00000 60.50000 59.00000 49.00000 57.75000 55.66667 60.00000
```

Matrices

Matrices are rectangular arrays arranged in rows and columns. In R we can construct a matrix in several ways:

```
 y<-1:12; y
 dim(y)<-c(4,3); y
# or
y<-1:12
 x<-matrix(y,ncol=3); x
# or
z<-matrix(y,nrow=4); z
# or
m<-matrix(c(1,2,3,4,5,6,7,8,9,10,11,12),4,3); m
```

There is also the possibility of naming rows and columns:

```
colnames(m)<-c("c1","c2","c3")
rownames(m)<-c("r1","r2","r3","r4")
m

# We can access individual elements of the matrix
m[2,2]
# or
m["r2","c2"]
```

Some operations involving matrices:

```
A<-matrix(c(5,3,4,8,9,10),3,2,2,5,4);A
B<-matrix(c(-1,3,-2,7,0,4,-2,4,5),2,3);B
A+B
A%*%B
```

... and some more:

`t(B)`
Transpose of a matrix

`det(A)`
Determinant of a matrix

`solve(B)`
Inverse matrix

`diag(A)`
Diagonal of a matrix

`rowMeans(A)`
Vector containing the mean of each row of a matrix

Lists

A list is an ordered collection of objects known as components of a list collection. These components do not need to be the same type, size or so. The components of a R list are always numbered and may have a name associated with each.

```
> lt<-list(name="Mary",marital.st="married",nr.kids=3,age.kids=c(4,7,9))
> lt
$name
[1] "Mary"
$marital.st
[1] "married"
$nr.kids
[1] 3
$age.kids
[1] 4 7 9
> length(lt)
[1] 4
> mode(lt)
[1] "list"
> names(lt)
[1] "name"      "marital.st"    "nr.kids"     "age.kids"
```

The components of the lists are numbered, so we can access them through the order they occupy or through their names.

```
> lt[[1]]
[1] "Mary"
> lt[[3]]
[1] 3
>
> # or
>
> lt$name
[1] "Mary"
> lt$nr.kids
[1] 3
```

We can change the names of the components;

```
> names(lt)<-c("name","m.state","chil.nr","child.age")
> names(lt)

[1] "name"      "m.state"     "child.nr"     "child.age"
```

add new components;

```
> lt$age<-35
> lt$parents<-c("Paul","Susan")
> names(lt)

[1] "name"      "m.state"     "child.nr"     "child.age"     "age"     "parents"
```

or concatenate lists

```
> lt.2<-list(blood.type="A",sex="F")
> lt.3<-c(lt,lt.2)
```

Data frames

Data frames are data tables. A data frame is very similar to an array, but the columns are named and may contain different types of data, as opposed to an array. Each row is a record and each column corresponds to a variable.

A data frame can be created as follows:

```
a<-c(3124,2136,2157,3589,1980)
b<-c("obese","normal","normal","obese","normal")
c<-c("m","f","m","f","f")
patients.inform<-data.frame(cal.cons=a,group=b,gender=c)
patients.inform
cal.cons  group gender
    3124  obese      m
    2136 normal      f
    2157 normal      m
    3589  obese      f
    1980 normal      f
```

The elements of a data frame may be accessed in the same way as for a matrix:

```
> patients.inform[2,2]
[1] normal
Levels: normal obese
```

The columns can be accessed in its entirety by using their names:

```
> patients.inform$group
[1] obese  normal normal obese  normal
Levels: normal obese
```

We can do more complex queries to databases, such as for example:

```
> patients.inform[patients.inform$cal.cons<3000,]
  cal.cons  group gender
2     2136 normal      f
3     2157 normal      m
5     1980 normal      f
> patients.inform[patients.inform$cal.cons<3000,
+ "group"]
[1] normal normal normal
Levels: normal obese
> patients.inform[patients.inform$gender=="f",
+ c("cal.cons","group")]
  cal.cons  group
2     2136 normal
4     3589  obese
5     1980 normal
```

We can directly access the columns' values without referring to the data frame:

```
> attach(patients.inform)
> patients.inform[cal.cons<3000,]
  cal.cons  group gender
2     2136 normal      f
3     2157 normal      m
5     1980 normal      f
> group
[1] "obese"  "normal" "normal" "obese"  "normal"
> detach(patients.inform)
> gender
Error: object 'gender' not found
```

We can add new columns to the data frame. The only restriction is that the new column must have as many rows as the previous data frame:

```
> nrow(patients.inform)
[1] 5
> ncol(patients.inform)
[1] 3
> patients.inform$weight<-c(100,62,80,75,55)
> patients.inform
  cal.cons  group gender weight
1     3124  obese      m    100
2     2136 normal      f     62
3     2157 normal      m     80
4     3589  obese      f     75
5     1980 normal      f     55
```

4. Import and Export Data

Access to the database of a specific package

R has several datasets available, in addition to the databases accompanying the libraries (packages).

- For a list of data in current session use the function `data ()`; to load one of these datasets just enter the name as an argument: `data(BOD)`.

- For a database of a certain package, type in:

```
> data(package="lattice")
```

For a specific dataset of that package, type in:

```
> data(ethanol,package="lattice")
```

• When the package has already been loaded through function `library()`, the database is automatically included:

```
> library(Biobase)
> data(ethanol)
> ethanol
> ?ethanol
```

The structure of an existing database can be changed by the function `data.entry()`, which is available in some versions of R.

Function read.table()

- There are several possibilities of reading external data in R, however, the best way of importing data into R is the format .txt. The file should be placed in the working directory.
- Function read.table() creates a data frame.

Example: data <- read.table("data.txt"), creates the data frame data from the imported file.

- This function reads files with file name extensions like: `.txt`, `.csv`, `.dat`.
- This function has many arguments, as for example:
  - `"namefile.txt"`, file name;
  - `header=T or F`, indicates whether the file includes, or not, the columns names;
  - `sep=" "`, columns separator;
  - `dec=","`, decimal separator.

• There are some restrictions concerning the external file, in order to `read.table()` to import it correctly:

1. The first line may contain the names of the fields or variables. One should use the argument `header=TRUE` to provide that information.
2. If the first element of the first row is empty, `R` assumes that this line corresponds to the column names. In this case, in each of the following lines, the first element is the label line, followed by the values of other variables.
3. You can add vectors of names for rows and columns, with the arguments `row.names` and `col.names`, respectively.

Import Data from Other Software

Function `read.xls()`

This function requires loading library `gdata`. This feature allows you to spell the name of the file and the number of the worksheet that contains the data we want to import.

Example: `data.excel <- read.xls('data.xls',sheet=3)`

Function `read.spss()`

R is able to import data from other software, requiring the loading of package `foreign`.

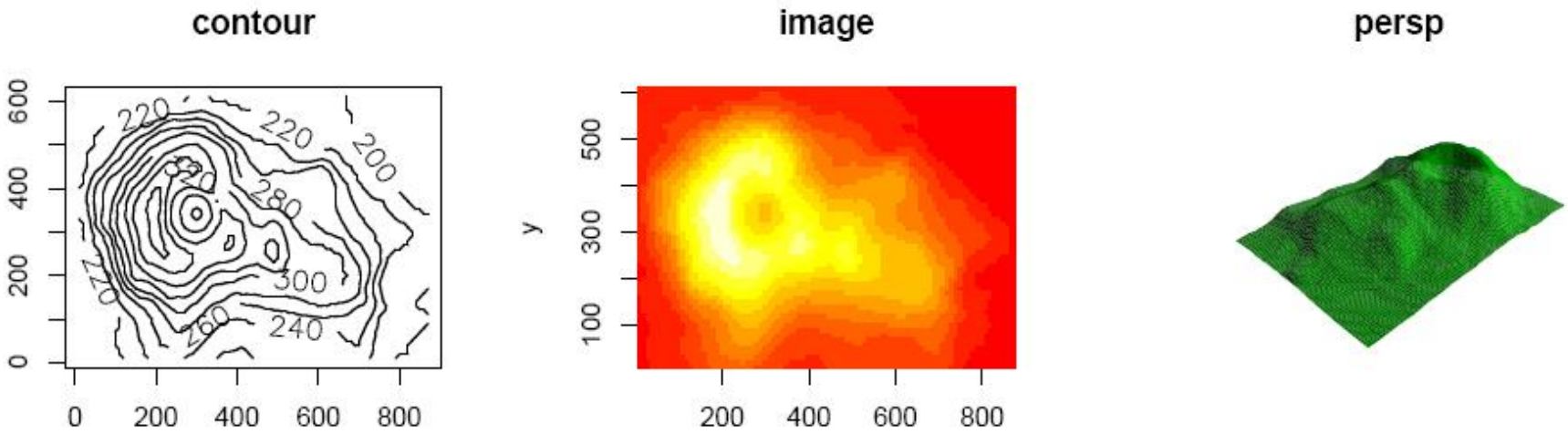
Example:

```
> data.spss<-read.spss(file="iris.sav",use.value.label=T,
+ to.data.frame=T)
```

The first argument is the file name; the second argument allows the existence of qualitative data; and the third argument will create a data frame.

5. Graphical Functions

The R is extremely versatile and produces high-quality graphics.

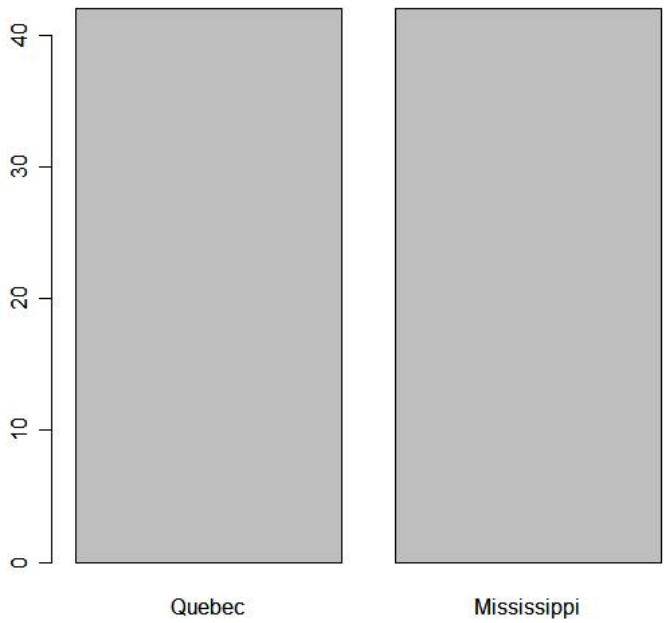


To see some of the possibilities that R offers, do `demo(graphics)` followed by Enter to show each chart.

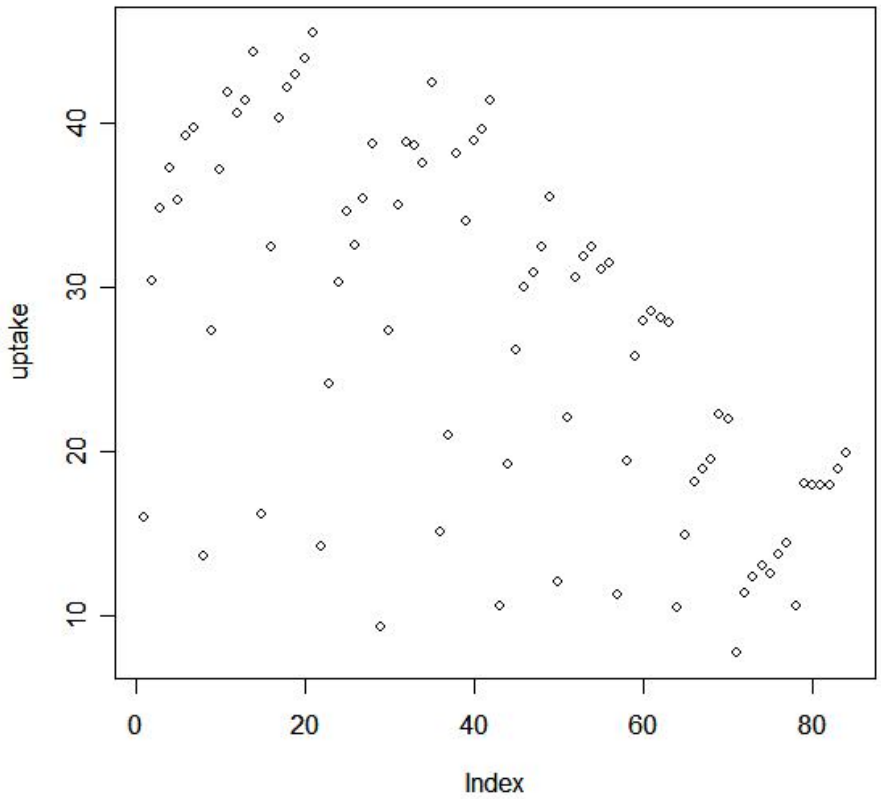
Function plot()

The function plot() is the more generic graphic function, and the type of generated chart depends on the first argument of the function:

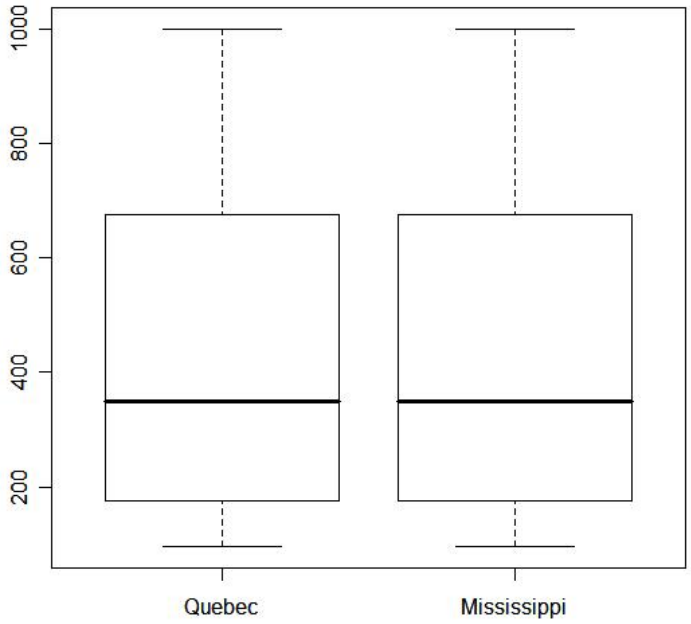
```
> data(CO2); names(CO2)
> attach(CO2)
> plot(Type)
```



```
> plot(uptake)  # dispersion diagram
```



```
> plot(Type,conc)  # box plot
```



The following arguments are used in most graphical functions:

main=	title
xlab=, ylab=	axes legend
xlim=, ylim=	limits of the axes
type=	"l" lines "p" points "o" overplotted points and lines "h" vertical lines "s" steps; ...

Multiple Graphics in One device

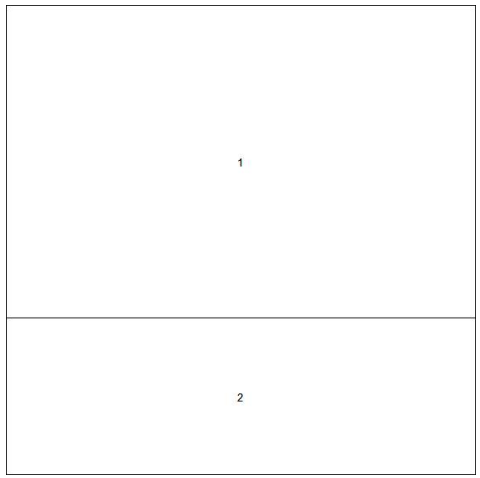
The mfrow argument can be used to configure the graphics device, setting line by line. If you want to set column by column should use mfcol.

Exercise: Proceed as follows in R:

```
library(MASS)
data(Animals)
names(Animals)
attach(Animals)
par(mfrow=c(2,2),pch=16)
plot(body, brain)
plot(sqrt(body),sqrt(brain))
plot((body)^0.1,(brain)^0.1)
plot(log(body),log(brain))
```

- We can divide the area of the graphics window in different sizes. The `layout()` function receives as main argument an array and its dimensions define how the `device` will be split.
- Function `layout.show()` can be used to illustrate visually the information.
- The `heights` argument serves to clarify the different sizes between the rows of the matrix, defining the areas in which the `device` is divided.

```
> layout(matrix(1:2, 2, 1), heights = c(2, 1))
> layout.show(2)
```

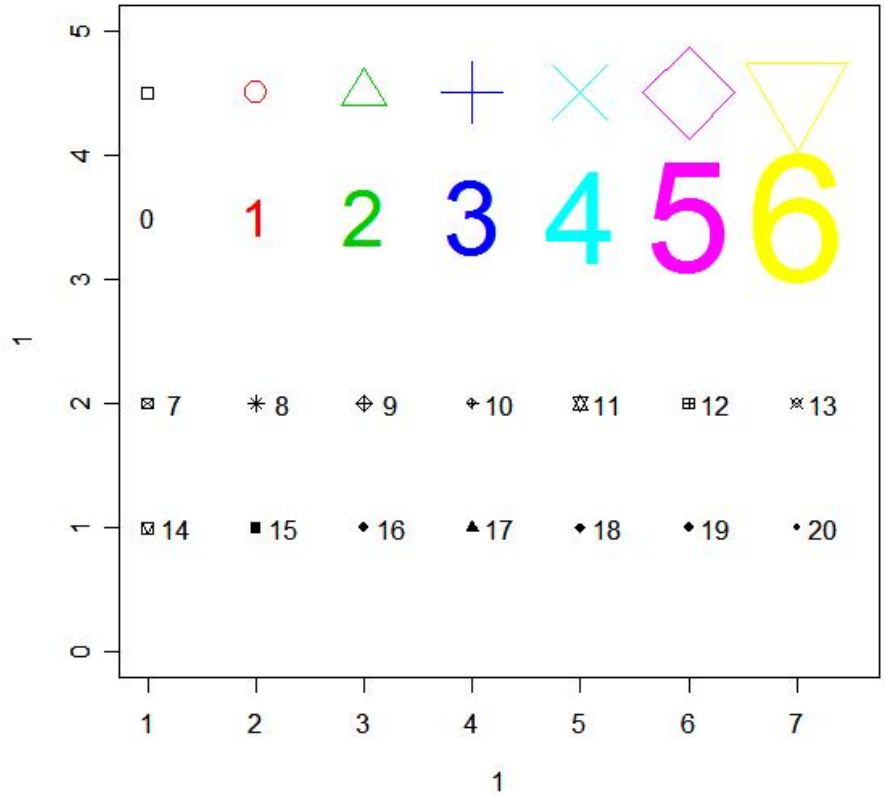


Argument `cex`, for functions `plot()` and `points()`, controls the dimension of the symbols and the text. Argument `col` controls the colors, and `pch` defines the type of symbol.

Exercise:

```
# no points in the graph
plot(1, 1, xlim=c(1, 7.5), ylim=c(0,5), type="n")
points(1:7, rep(4.5, 7), cex=1:7, col=1:7, pch=0:6)
text(1:7,rep(3.5, 7), labels=paste(0:6), cex=1:7, col=1:7)
points(1:7,rep(2,7), pch=(0:6)+7)
# Labels corresponding to symbols' numbers
text((1:7)+0.25, rep(2,7), paste((0:6)+7))
# Symbols 14 to 20
points(1:7,rep(1,7), pch=(0:6)+14)
text((1:7)+0.25, rep(1,7), paste((0:6)+14))
```

The resulting graph is:



6. Loops and Functions

LOOPS

A loop is something that is always running up to the moment when certain condition is verified.

Example:

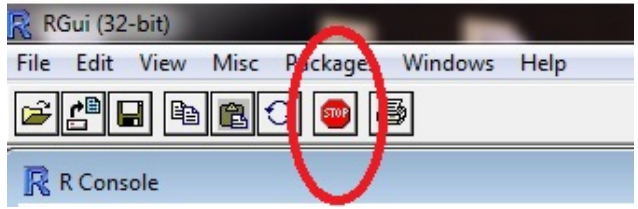
```
While A > B do
print "A is greater than B"
```

In this example, the loop stops printing *A is greater than B*, when the condition $A \leq B$ is satisfied.

No R loops may be executes by using the following loop functions:

- `for()`
- `while()`
- `repeat()`
- `apply()` family (slides 39–40)

Any loop function may be interrupted by pressing the button **STOP** in the menu, as indicated in the figure.



Function `for()`

Function `for()` syntax

```
for(counter in vector)   { instructions }
```

Example:

```
> x<-c(3,5,6,1,2,4)
> z <- NULL
> for(i in 1:length(x)) {
+    if(x[i] < 5)  z <- c(z, x[i] - 1)
+    else          z <- c(z, 100)      }
> z
[1] 2 100 100   0   1   3
```

Function while()

Similar to function for(), but the iterations are controlled by one condition.

Function while() syntax

```
while(condition)   { instructions }
```

Example:

```
> z <- 0
> while(z < 5)  {
+     z <- z + 2
+     print(z)  }
[1] 2
[1] 4
[1] 6
```

Function repeat()

The loop is repeated until the stop condition ins satisfied. The stop condition is simply a statement that allows you to validate the flow interruption. The stop command is given by the function break().

Function repeat() syntax

```
repeat { instructions
         stop condition }
```

Example:

```
> z <- 0
> repeat { z <- z + 1
+          print(z)
+          if(z > 2) break() }
[1] 1
[1] 2
[1] 3
```

FUNCTIONS

A very useful feature of R is the possibility of expanding the existing functions and writing custom functions easily.

Constructing a function:

```
func <- function(arg1,arg2,...) { structure of the function }
```

The function returns a given object by the values assigned to the arguments, `arg1`, `arg2`, `...` by calling the function as follows:

```
func(arg1=..., arg2=...)
```

Example:

```
> func <- function(x1, x2=5) {
+   z1 <- x1/x2
+   z2 <- x1*x2
+   vec <- c(z1, z2)  # This line and the next one
+   return(vec)       # may be substituted by  c(z1,z2)
+ }
> func                # Prints the script of function 'func'
> func(x1=2, x2=5)    # Applies the function to the pair (2,5)

[1]  0.4   10.0



# or func(2,5) - The name of the arguments is not necessary
#                if they are placed by the established order
# or func(2) - We may omit the value of x2 since there
#              is a value established by default
```

Acknowledgements:

Carina Silva-Fortes (ESTeSL – IPL), for allowing the use of some material produced by both of us in previous courses.

Bibliography:

Adler, J. (2010). *R in a Nutshell: a Desktop Quick Reference*. O'Reilly Media.