A brief introduction to R

DATA SCIENCE
M.José Ramírez-Quintana
(English version with modifications by José Hernández-Orallo)
ETSINF
Universitat Politècnica de València

September 16, 2015

Contents

1 The R environment

R is an open-source language and associated software for data analysis. One of the advantages of R is that we can understand what the data look like, what we can do with them or how they can be transformed. R is an object-oriented language (although it also includes other features that can be considered functional, imperative or multiparadigm). R is very powerful, flexible and especially extensible. It works with a command-line interface, although there are graphical interfaces for R. R is an interpreted language, which means that the commands are executed on the fly, without the need of building an executable file. In addition, R is a matricial language with a simple and intuitive syntax.

R is available for multiple platforms (see http://cran.r-project.org/)[1]. Since R has become very popular, there are numerous books, videos and tutorials that can be very helfpul both for the installation and learning the language. For instance, there are tutorials at http://cran.r-project.org/, or at http://www.r-tutor.com/r-introduction or wikibooks (https://en.wikibooks.org/wiki/R_Programming).

Figure 1 shows an R console. After some welcoming messages, the command line starts with the symbol ">", which indicates that R is ready to accept commands. R uses different colours to distinguish the commands that are introduced by the user (in red or blue, depending on the version and platform) from the results given by these commands (in blue or black, depending on the version and platform). In the figure, observe R's result to the command $2*5$. Before the result, 10, there is a 1 between squared brackets. This number indicates the line of the result and it is useful when there are many lines in the result (e.g., when we show the content of a dataset). This also allows us to go back.

Since R is object-oriented, the variable, data, functions, results, etc., are stored in main memory as objects, with a specific name. The name of an object must start with a letter (A–Z and a–z) and can include more letters, digits (0–9), and dots (.). R is case-sensitive for objects, so x and X refer to different objects. The dot (.) has no subobject interpretation. For instance, `a.b` is just the name of an object, unrelated to `a`. Subproperties are usually accessed with the dollar symbol $, as we will see.

The symbol # comments out the rest of a line. We can ask for the documentation (or help) of any function by using a single question mark before it, such as "`?sort`", or the function "`help`", as in "`help(sort)`". An .html document will be sent to a browser, which will display the help about that function or term. If we want to search the whole R documentation, we

[1] If you are using your own computer, you can download and install it from that link. You can also install an IDE for R, such as http://www.rstudio.com. If you are in the lab, you can use the installations already there, although you will not be able to install additional packages.

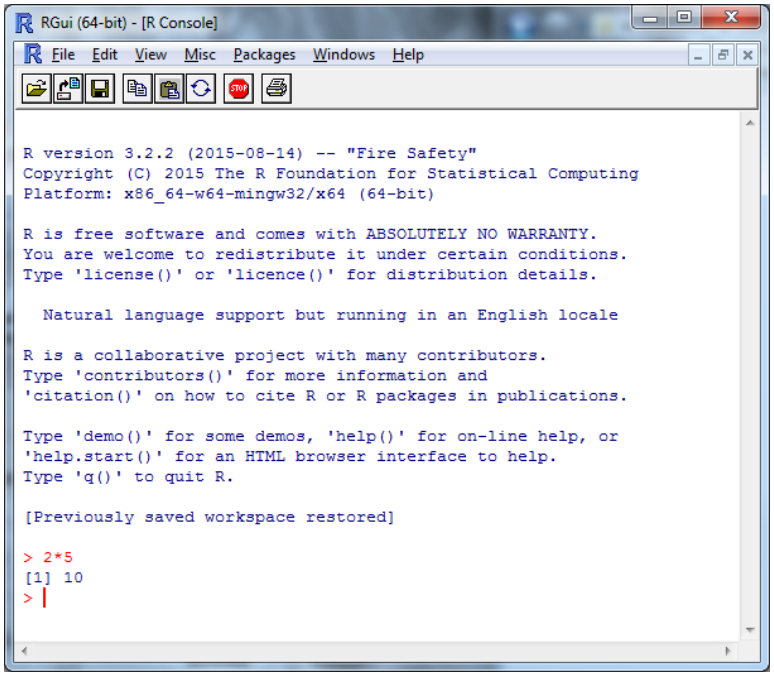


Figure 1: R environment interface.

can use a double question mark, such as "`??sort`".

2 Object creation. Assignment. Attributes

An object can be created with the operator "`->`" or "`-<`" depending on the direction in which the object is assigned:

```
> x<-5  # assignment example
> x
[1] 5
> y<-3*3
> y
[1] 9
```

If the object already exists, its previous value will be replaced after the assignment. Alternatively, we can use "=" or the command `assign`.

```
> assign("X",3.4)
> X
[1] 3.4
> y=2
> y
[1] 2
```

The objects in R, apart from name and content, also have attributes that specify the datatypes that are represented by the object. Every object has two intrinsic attributes: type and length. The type refers to the class of the elements in the object (numeric, character, complex and logical). The length is simply the number of elements in the object. In order to see the type and length of an object, we can use the functions `mode` (or alternatively `class()`) and `length`, respectively:

```
> A<-"Data Science"; h=17; l<-TRUE
>  mode(A); mode(h); mode(l)
[1] "character"
[1] "numeric"
[1] "logical"
```

3 Vectors

The simplest object in R is the vector. Even when we write `x<-3` we are actually creating a vector with one single element! Of course we can create vectors with more than one element. Vectors are used to store values of the same atomic datatype (character, logical, numeric and complex). In R, vectors are created using the function `c( )`:

```
> v<-c("AVS", "CDA", "DIM", "TVD")
> v
[1] "AVS" "CDA" "DIM" "TVD"
> length(v)
[1] 4
> mode(v)
[1] "character"
```

If the types are different, coercion (i.e., conversion) takes place, if possible.

```
 > u<-c(1,2,4,6.0)
> u
[1] 1 2 4 6
> w<-c(1.3,2.5,3.9,5)
> w
[1] 1.3 2.5 3.9 5.0
```

We can have explicit coercion by using the functions `as.*` as we show in the following table:

Function	Conversion
as.numeric	FALSE → 0
	TRUE → 1
	"1", "2", . . . → 1,2,. . .
	"A",. . . → NA
as.logical	0 → FALSE
	other numbers → TRUE
	"FALSE", "F"→ FALSE
	"TRUE", "T"→ TRUE
	other characters → NA
as.character	1,2, . . . → "1", "2",. . .
	FALSE → "FALSE"
	TRUE → "TRUE"

```
>  u<-c(1,2,3,4)
> class(u)
[1] "numeric"
> as.numeric(u)
[1] 1 2 3 4
> as.logical(u)
[1] TRUE TRUE TRUE TRUE
> as.character(u)
[1] "1" "2" "3" "4"
> x<-c("a","b","c")
> mode(x)
[1] "character"
> as.numeric(x)
[1] NA NA NA
Warning message:
NAs introduced by coercion
```

We can also create a vector with the function `vector()`, indicating its mode and length. The vector is created with a by-default value (for instance, 0 if it is numeric, "" if it is a character, FALSE if it is logical, etc.).

```
> x <- vector("numeric", length = 10)
> x
 [1] 0 0 0 0 0 0 0 0 0 0
```

To access an element of a vector, we put an index between simple square brackets:

```
> v[2]
[1] "CDA"
> v[5]<-"MIT"
> v
```

```
[1] "AVS"   "CDA"   "DIM"   "TVD"   "MIT"
> t<-c(v[2],v[4])
> t
[1] "CDA" "TVD"
```

R features several functions that work with vectors. Some of them are shown below:

Usual operators	$+, -, *, /, \hat{}$
Arithmetic functions	log, exp, sin, cos, tan, sqrt, etc
Maximum, minimum and range	max, min, range
Length	length
Product and sum	prod, sum
Mean, median and variance	mean, median, var
Sorting	sort

4 Factors and contingency tables

A factor is a vector that is used to specify discrete values over the elements of another vector of the same length. In R there are nominal and ordinal factors. Factors have *levels*, which are the possible values they can take (although R stores these values as numerical codes because of efficiency reasons).

Let us see what factors and levels are more precisely. Suppose that we have a sample of 8 people for whom we know what pet they have.

```
> pet<-c("cat","dog","dog","cat","cat","snake",
+ "parrot","cat")
> pet
[1]"cat""dog" "dog" "cat""cat" "snake" "parrot" "cat"
> Fpet<-factor(pet)
> Fpet
[1] cat  dog  dog  cat  cat  snake  parrot cat
Levels: cat dog parrot snake
> levels(Fpet)
[1] "cat"   "dog"   "parrot"  "snake"
> mode(Fpet)
[1] "numeric"
```

We can use the factors to count the occurrence of each value (level). In order to do this we use the function `table()`:

```
> table(Fpet)
Fpet
   cat    dog   parrot  snake
     4      2        1      1
```

This function can also cross several factors, leading to a contingency table. For instance, we can create another factor with the information of whether the pet's owner is a boy or a girl.

```
> own<-factor(c("girl","boy","boy","girl","girl",
+ "boy","boy","boy"))
> own
[1] girl boy  boy  girl girl boy  boy  boy
Levels: boy girl
> table(own)
own
 boy girl
   5    3
> table(Fpet,own)
        own
Fpet     boy girl
  cat      1    3
  dog      2    0
  parrot   1    0
  snake    1    0
```

If we are only interested in the marginal frequencies, we use the function `margin.table(table,dimension)`:

```
> t<-table(Fpet,own)
> margin.table(t,1)
Fpet
   cat    dog parrot  snake
     4      2      1      1
> margin.table(t,2)
own
 boy girl
   5    3
```

and if we are interested in the relative frequencies then we can use the function: `prop.table(table,dimension)`:

```
> prop.table(t,1)
        own
Fpet      boy girl
  cat    0.25 0.75
  dog    1.00 0.00
  parrot 1.00 0.00
  snake  1.00 0.00
> prop.table(t,2)
        own
```

```
Fpet      boy girl
  cat     0.2  1.0
  dog     0.4  0.0
  parrot  0.2  0.0
  snake   0.2  0.0
```

The order of the levels can be established using the argument `levels` of function `factor()`:

```
> x <- factor(c("yes", "yes", "no", "yes", "no"),
+ levels = c("yes", "no"))
> x
[1] yes yes no  yes no
Levels: yes no
```

5 Sequences

R provides several functions to generate sequences. For instance, we can define a numeric sequence as follows:

```
> x<-1:15
> x
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
> y<-5:1
> y
[1] 5 4 3 2 1
```

The function `seq()` can also generate sequences of real numbers:

```
> seq(1, 5, 0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

where the first argument indicates the start of the sequence, the second one sets the stop value and the third one is the increment used to generate the secuence. An alternative way of expressing the same is:

```
> seq(length=9, from=1, to=5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

Observe how we use the name of the arguments for clarity.

Another function that is useful to generate sequences is `rep()`:

```
> rep(5,10)
[1] 5 5 5 5 5 5 5 5 5 5
> rep(c(1,2),4)
[1] 1 2 1 2 1 2 1 2
```

La function `gl()` makes it possible to generate sequences involving factors. `gl(k,n)` generates a sequence where `k` is the number of levels of the factor and `n` is the number of repetitions of each level:

```
> gl(2,4,labels=c("boy","girl"))
[1] boy  boy  boy  boy  girl girl girl girl
Levels: boy girl
```

We can also generate random sequences. In order to do that with several distribution, R provides several functions with the general form `Xfunc(n, p1, p2,...)` where `X` can take the values `d, p, q, r` to obtain the probability density, the cumulative probability, the quartile value and the random generation of values respectively; `func` indicates the distribution (`norm` for normal, `binom` for binomial, `beta` for beta, `unif` for uniform, ...); `n` is the number of data to be generated and `p1, p2, ...` are the values for the parmaters of the distribution. The following table just shows some of them (there are many more, embedded in R or in additional packages) and their parameters:

Distribution	rfunc
Gaussian (normal)	`rnorm(n, mean=0, sd=1)`
exponential	`rexp(n, rate=1)`
gamma	`rgamma(n, shape, scale=1)`
Poisson	`rpois(n, lambda)`
beta	`rbeta(n, shape1, shape2)`
binomial	`rbinom(n, size, prob)`
geom	`rgeom(n, prob)`
uniform	`runif(n, min=0, max=1)`

```
> runif(5)
[1] 0.9111758 0.4921438 0.3238787 0.4402546 0.1792024
> rnorm(10, mean=5, sd=1)
 [1] 4.481318  4.319482  4.105529  7.498646  5.668317
4.046477  6.629298  4.627495  6.276711  4.497717
```

6 Matrices and arrays

In mathematics a matrix is a bidimensional structure. However in R this is really a vector with an additional attribute (`dim`), which is actually a numeric vector of length 2 that defines the number of rows and columns of the matrix, i.e., its 'structure'. Matrices are defined as follows:

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

where `byrow` indicates whether the values fill the matrix by columns (`TRUE`) or Rows (`FALSE`), and `dimnames` allows us to give name to rows and columns. Let us see an example:

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

Matrices can be created from vectors using the appropriate values for the attribute `dim`:

```
> x<-1:10
> x
[1]  1  2  3  4  5  6  7  8  9 10
> dim(x) <- c(2, 5)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

We can traspose a matrix with `t`:

```
> t(x)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
[5,]    9   10
```

We can also construct matrices by joining vectors or matrices by rows (`rbind`) or columns (`cbind`):

```
> x<-1:3
> y<-10:12
> cbind(x,y)
     x  y
[1,] 1 10
[2,] 2 11
[3,] 3 12
> rbind(x,y)
  [,1] [,2] [,3]
x    1    2    3
y   10   11   12
```

Observe the notation for indices as `[i,j]`. We can specify both, one or none of the indices:

```
x<-matrix(1:4,2,2)
```

```
> x[1,2]
[1] 3
> x[1,]
[1] 1 3
> x[,2]
[1] 3 4
> x[,]
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Finally, arrays are multidimensional generalisations of matrices, with more than 2 dimensions. If the provided data are not enough to fill all cells, they are replicated (what R refers to as 'recycling').

```
> a<-array(1:10,c(3,2,2))
> a
, , 1
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

, , 2
     [,1] [,2]
[1,]    7   10
[2,]    8    1
[3,]    9    2
```

7 Lists

Lists are a special kind of data type that can contain a series of elements of different classes. They can be indexed by names:

```
> animal <-list(%class='mammal',
order='carnivore', family='feline',
+ species='lynx')
> animal
%$class
%[1] "animal"
$order
[1] "carnivore"
$family
[1] "feline"
```

```
$species
[1] "lynx"
```

We can extract the elements of a list using the following indexed schema:

```
> animal[[1]]
[1] "carnivore"
```

Observe that we have used double square brackets to access the element. If we use simple square brackets we do not get down to the element and get a different result:

```
> animal[1]
$order
[1] "carnivore"
```

Actually, `animal[1]` extracts a sublist that comprises the first component of `animal`, whereas `animal[[1]]` extracts the value of that first component. We can also access the values using the name of the component:

listname$componentname

```
> animal$order
[1] "carnivore"
```

We can add elements dynamically to a list using names. We can start with an empty list, as follows:

```
> l <- list()
> l$a <- 3
> l$b <- 5
> l$c <- 8
> l
$a
[1] 3
$b
[1] 5
$c
[1] 8
```

Lists can be without names and composed of various kinds of elements, as follows:

```
> mylist <- list(3,'a')
> mylist[[1]]
[1] 3
> mylist[[2]]
```

```
[1] "a"
> mylist[[3]] <- 8.5
[[1]]
[1] 3
[[2]]
[1] "a"
[[3]]
[1] 8.5
```

We can even have vectors as elements of a list:

```
> animal$class<-c('vertebrate','mammal')
> animal
$order
[1] "carnivore"
$family
[1] "feline"
$name
[1] "lynx"
$class
[1] "vertebrate" "mammal"
```

If the classes of all elements are compatible (or coerceable) we can convert a list to a vector with the function
`unlist(x, recursive=TRUE, use.names=TRUE)`

```
> unlist(animal)
       order       family      species       class1
 "carnivore"     "feline"       "lynx" "vertebrate"
      class2
    "mammal"
```

8 Avoiding loops: apply and other commodities

For vectors, matrices, arrays and lists, we have the power of applying a function to all the elements, without the need of a loop. This is not only very handy but usually much more efficient than doing a loop.

The main function is `lapply`, but there are several variants for making its use easier: `sapply`, `vapply`, and others. The easiest one is `sapply(x, fun, ...)`, where `x` is a list (or something that can be coerced to a list if it is not, such as a vector), `fun` is the name of the function and ... other arguments. The result is usually rendered as the simplest datatype compatible with the resuting data. For instance, we can round all the elements of a vector of real numbers and get a vector of integer numbers as the result.

```
> myvector <- c(3.5, 7.8, 4.2, 2.5)
> sapply(myvector, round)
[1] 4 8 4 2
```

With lists, it is preferrable to use `lapply(x, fun, ...)`, because `lapply` always returns a list.

```
> lapply(animal,length)
$order
[1] 1
$family
[1] 1
$species
[1] 1
$class
[1] 2
```

Below we see more examples of the type of the result depending on whether we use `lapply` or `sapply`. The second example being a list of vectors:

```
> sapply(animal,length)
  order  family species   class
      1       1       1       2
```

```
> x <- list(a = 1:4, b = rnorm(10),
+ c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5
$b
[1] -0.06742888
$c
[1] 1.182268
$d
[1] 5.039349

> sapply(x, mean)
         a          b          c          d
2.50000000 -0.06742888  1.18226849  5.03934889
```

In a similar way we can apply operators to structures. For instance, we can count how many elements there are in a vector meeting a condition in various ways, without the use of loops:

```
> myvector
```

```
[1] 3.5 7.8 4.2 2.5
> myvector + 1
[1] 4.5 8.8 5.2 3.5
> myvector < 5
[1]  TRUE FALSE  TRUE  TRUE
> sum(myvector < 5)
[1] 3
> myvector[myvector < 5]
[1] 3.5 4.2 2.5
```

Also, there is a special function `which` that returns the indices of the elements meeting a condition:

```
> which(myvector < 5)
[1] 1 3 4
```

With a little bit of imagination, we can nest several things in very compact expressions. For instance, the following expression adds 1 to those elements that are lower than 5.

```
> myvec <- myvector
> myvec[which(myvec < 5)] <- myvec[myvec < 5] + 1
> myvec
[1] 4.5 7.8 5.2 3.5
```

One can think about ways of making the previous expression even shorter. For instance, we can use the function `ifelse`

```
> myvec <- myvector
> ifelse(myvec < 5, myvec +1, myvec)
> myvec
[1] 4.5 7.8 5.2 3.5
```

In any case, it is usually much faster to use these expressions than using loops.

9 Data frames

A *data frame* is the most convenient data structure to store and work with data tables (tables such as those used in relational databases, with their rows and columns). They are similar to matrices (since they are bidimensional) but, unlike matrices, data frames can store data with columns of different types. Each row in a matrix represents an instance, case, observation or example, whereas a column represents an attribute. For instance, we could create a data frame as follows:

```
> d<-data.frame(name=c('Anne','Joe','Mario','Rose','
+ Mary'),age=c(21,34,54,27,41))
```

```
> d
   name age
1  Anne  21
2   Joe  34
3 Mario  54
4  Rose  27
5  Mary  41
```

We can add columns to the data frame. The only restriction is that the column must have the same number of rows as the data frame where it is inserted:

```
> city<-c('Valencia','Barcelona','Madrid','Valencia',
+ 'Valencia')
> job<-c('student','dealer','engineer',
+'physician','journalist')
> d2<-cbind(city,job,d)
> d2
       city         job  name age
1  Valencia     student  Anne  21
2 Barcelona      dealer   Joe  34
3    Madrid    engineer Mario  54
4  Valencia   physician  Rose  27
5  Valencia  journalist  Mary  41
```

We can access the values in rows or columns, or even use data that meet several conditions. For instance, we can query the values of the attribute "city":

```
> d2$city
[1] Valencia  Barcelona Madrid    Valencia  Valencia
Levels: Barcelona Madrid Valencia
> d2$job
[1] student  dealer  engineer  physician  journalist
Levels: dealer engineer journalist physician student
```

Observe that, since they are nominal values, R has converted the results to factors. Similar to lists, double square brackets are useful to extract the values of the data frame while the simple square brackets are useful to extract elements:

```
> d2[[1]]
```

```
[1] Valencia  Barcelona Madrid    Valencia  Valencia
Levels: Barcelona Madrid Valencia
> d2[1,]
[1] Valencia  Barcelona Madrid    Valencia  Valencia
Levels: Barcelona Madrid Valencia
> d2[1]
     city
1  Valencia
2 Barcelona
3    Madrid
4  Valencia
5  Valencia
```

Additionally, we can also select the names and age of those people who live in "Valencia":

```
> d2[d2$city=='Valencia',c("name","age")]
  name age
1 Anne  21
4 Rose  27
5 Mary  41
```

or select those people who are more than 35 years old:

```
> d2[d2$age>35,]
      city         job  name age
3   Madrid   engineer Mario  54
5 Valencia journalist  Mary  41
```

To select a subset of elements of the data frame we can also use the function `subset()`:

```
> subset(d2,city=='Valencia',job:age)
         job name age
1    student Anne  21
4  physician Rose  27
5 journalist Mary  41
```

The functions `ncol(), nrow()` indicate the number of rows and columns of a data frame (or an array or matrix):

```
> nrow(d2)
[1] 5
> ncol(d2)
[1] 4
```

In order to transform some columns or create several columns o new variables, we can do it redundantly as follows:

```
> d3 <- d2
> d3[,'age'] <- -d3[,'age']
> d3
       city         job  name age
1  Valencia     student  Anne -21
2 Barcelona      dealer   Joe -34
3    Madrid    engineer Mario -54
4  Valencia   physician  Rose -27
5  Valencia journalist  Mary -41
```

But we can use the function `transform()` to make it more smoothly and compact:

```
> d4<-transform(d2, age=-age)
> d4
       city         job  name age
1  Valencia     student  Anne -21
2 Barcelona      dealer   Joe -34
3    Madrid    engineer Mario -54
4  Valencia   physician  Rose -27
5  Valencia journalist  Mary -41
```

10 Changing names and levels

We can assign or change the names of the components of a vector, list, matrix or data frame with the function `names()`:

```
> x <- 1:3
> names(x)
NULL
> names(x) <- c("first", "second", "third")
> x
 first second  third
     1      2      3
> names(x)
[1] "first"  "second" "third"
```

It is also important to be able to change levels, otherwise some values cannot be modified, for instance:

```
> d
   name age
1  Anne  21
2   Joe  34
```

```
3 Mario  54
4  Rose  27
5  Mary  41
> d[5,1]
[1] Mary
Levels: Anne Joe Mario Mary Rose
> d[5,1] <- "Susan"
Warning message:
In `[<-.factor`(`*tmp*`, iseq, value = "Susan") :
  invalid factor level, NA generated
> levels(d[,1])[levels(d[,1])== "Mary"] <- "Susan"
> d
   name age
1  Anne  21
2   Joe  34
3 Mario  54
4  Rose  27
5  <NA>  41
> d[5,1] <- "Susan"
> d
   name age
1  Anne  21
2   Joe  34
3 Mario  54
4  Rose  27
5 Susan  41
```

We have to be very careful with these things if a level is used twice since whenever the level is renamed, all the values using that level will be lost.

11 Infinite, undefined, missing and empty values

R has infinite, undefined (not a number), missing and empty values. It is important to distinguish them. Infinite values are perfectly fine, and we can operate with them:

```
> a <- 1/0
> a
[1] Inf
> a + 3
[1] Inf
> -a
[1] -Inf
```

The symbol `NaN` denotes a numeric number that is *not a number*. For instance,

```
> b <- 0/0
> b
[1] NaN
> b + 3
[1] NaN
> b*0
[1] NaN
```

This is very different to a missing value, denoted by `NA` (*not available*), which is independent of the data type.

```
> t<-c(T,F,NA,FALSE)
> t
```

Finally, the value `NULL` denotes an empty object, and it must not be confused with `NA` or `NaN`. Note that in SQL, NULL represents a missing value but not in R. Empty objects are useful to define variables that are enlarged or assigned afterwards.

```
> a <- NULL
> a
> NULL
> a <- c(a,3)  # works because 'a' has been previously defined
> a
[1] 3
> a <- c(a,5)
> a
[1] 3 5
```

But be careful that NULL is skipped (i.e., removed) for vectors, as NULL means nothing.

```
> s<-c(T,F,NULL,FALSE)
> s
[1]  TRUE FALSE FALSE
```

At any point we can check the value of an object using `is.infinite()`, `is.finite()`, `is.na()`, `is.nan()` or `is.null()`:

```
> t<-c(T,F,NA,FALSE)
> is.na(t)
[1] FALSE FALSE  TRUE FALSE
> is.nan(t)
[1] FALSE FALSE FALSE FALSE
> is.nan(0/0)
```

```
[1] TRUE
> is.null(0/0)
[1] FALSE
> is.infinite(0/0)
[1] FALSE
```

A common task in some datasets is to delete missing values

```
> x<- c(1, 2, NA, 4, NA, 5)
> bad <- is.na(x)
> x[!bad]
[1] 1 2 4 5
```

In the following example, we use `complete.cases()` to select those pairs (or rows in a dataframe) free of missing values:

```
> x <- c(1, 2, NA, 4, NA, 5)
> y <- c("a",NA,NA,"d","e","f")
> good <- complete.cases(x, y)
> good
[1] TRUE FALSE FALSE TRUE FALSE TRUE
> x[good]
[1] 1 4 5
> y[good]
[1] "a" "d" "f"
> mydf <- data.frame(x,y)
> good2 <- complete.cases(mydf)
> good2
[1]  TRUE FALSE FALSE  TRUE FALSE  TRUE
```

12 Sorting data

The function `sort()` is able to order numbers or alphanumeric characters (using the ASCII order).

```
> x<-c("auu","1uu","Auu")
> sort(x)
[1] "1uu" "Auu" "auu"
```

Other functions for sorting or related to it are:

- `order()`: returns the indices of the sorted vector.
- `duplicated()`: returns a logical vector with the values that are duplicated.
- `unique()`: returns the values that are different.

For instance, we can use these functions to get the three youngest people in our small dataset:

```
> d
   name age
1  Anne  21
2   Joe  34
3 Mario  54
4  Rose  27
5  Mary  41
> d[order(d[2])[1:3],1]
[1] Anne Rose Joe
Levels: Anne Joe Mario Mary Rose
```

13 Loading and saving data in R

Given a text file, we can load data using functions such as `read.table()`, `read.csv()` or `read.xls()` (the latter may depend on the version of the Excel file, so it is recommended to export the file into a standard such as csv):

- If the file meets the following conditions, then we can load it directly with the function `read.table()` (using the default values):
  - The first line must contain the name of each attribute.
  - Each of the following lines must start with the row label followed by the values of each attribute.
  - The values are separated by an empty space: " ".

  If the file has one element less in the first line than in the rest, then the above schema must be used. By default, the numeric elements (except the row labels) are stored as numeric variables; and non-numeric as factors. The by-default character for comments is #. All this can be changed using the following attributes:
  - `file`: the name of the file.
  - `header`: logical value indicating whether there is a header row in the file.
  - `sep`: the character (or string) that separates columns.
  - `colClasses`: a character vector with the class (datatype) of each column.
  - `nrows`: number of files to be read.

  - `comment.char`: the character (or string) used for comments. If we do not expect comments, we should set this value to "".
  - `skip`: number of lines to skip from the beginning.
  - `stringsAsFactors`: TRUE to code the variables of type character as factor.
  - `dec`: the character for the decimal point.

- `read.csv()` is like the previous one but the columns are separated by commas.

Note that in some countries in continental Europe (such as Spain or France) and Latin America the decimal points are indicated with a comma, and columns are usually indicated with semicolons. Consequently, some files generated by spreadsheets can be in this format. If this is the case, `dec` and `sep` should be specified.

Similar to `read.X()` there are several `write.Y()` functions for writing files. You can check the help of these functions or a manual of R for more functions about reading and writing files.

It is important that you define the working directory where you read or write files, using:

```
> WORKDIR <- "C:/MyFiles/"
> setwd(WORKDIR)
```

You can see where you are with `getwd()`.

14 R graphics

R provides an extensive battery of graphical tools, which can be used to do standard or customised plots. In order to see a short demo of the kind of graphics R is able to do, you can execute `demo(graphics)`.

The following table summarisse some of the most important graphics in R.

function	description
`plot(...)`	a scatterplot for isolated points or curves
`pie(...)`	a circular pie graphic
`boxplot(...)`	a *box-and-whiskers* plot
`barplot(...)`	a value histogram
`hist(...)`	a frequency histogram

Some of the arguments that can be used in the previous functions are described in the following table:

attribute	description	
`add=`	if changed to `TRUE`, it superpose the graph to an existing one.	
`axes=`	if change to `FALSE`, it does not draw the axes nor the graphic box.	
`type=`	type of graphic:	
	`''p''`	points (default value).
	`''l''`	line.
	`''b''`	points linked with lines.
	`''o''`	points with lines over them.
	`''h''`	vertical lines.
	`''s''`	stairs, data on the top
	`''S''`	stairs, data on the bottom
`xlim=, ylim=`	lower and upper axis limits.	
`xlab=, ylab=`	axis titles.	
`main=`	main titule of the plot.	
`sub=`	subtitle.	

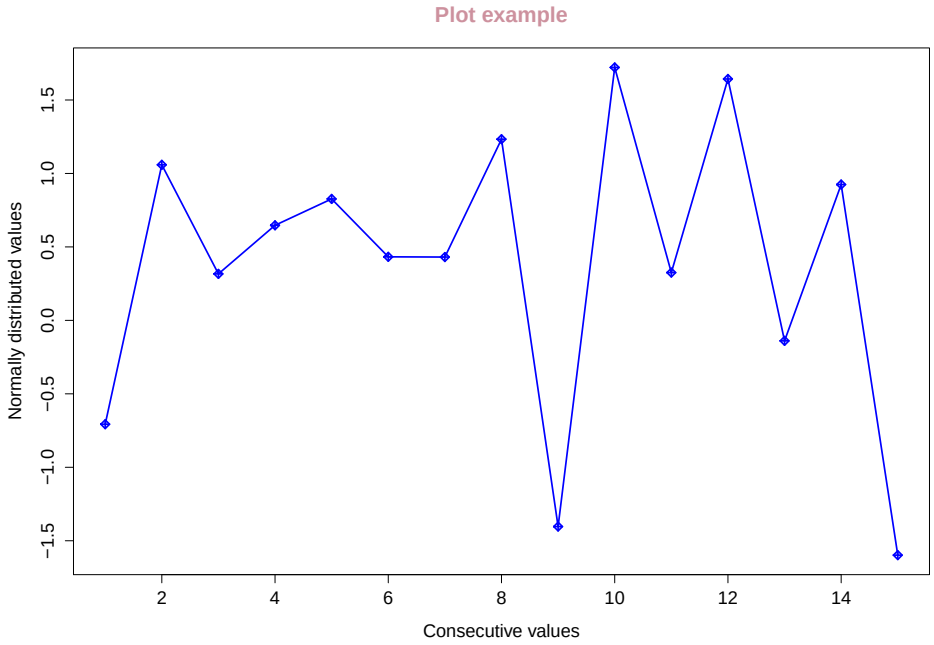
Apart from the previous modifier, one can always add low-level elements to a plot, such as lines, points, text, etc. The most common functions are:

function	description
`lines`	adds lines to a plot.
`points`	adds points to a plot.
`text`	adds text to a plot.
`title`	adds title to a plot.
`mtext`	adds texts on the margins of a plot.
`axis`	adds axes: 1=below, 2=left, 3=above y 4=right.

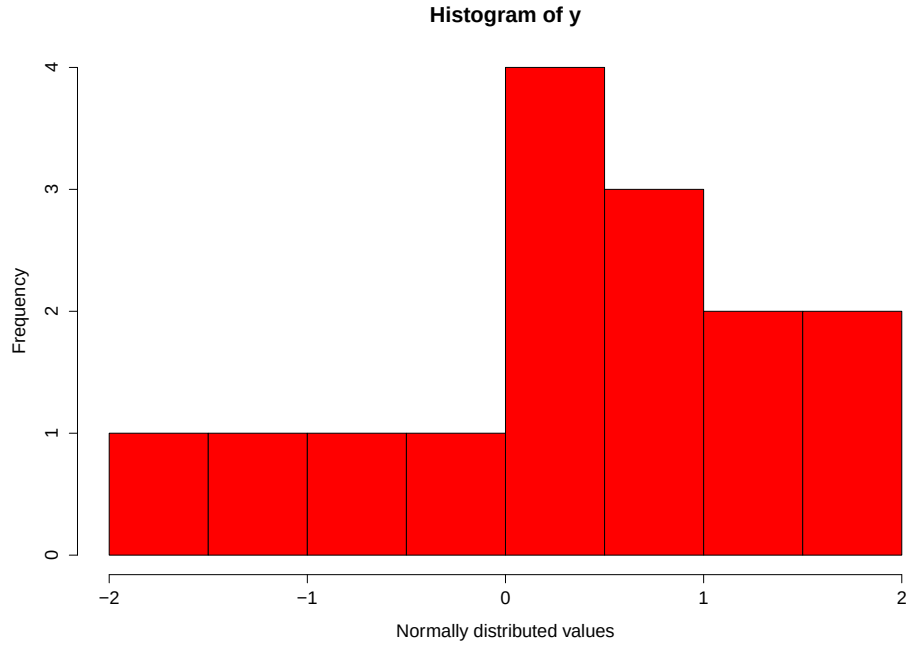
The function `par()` is used to establish parameters of the plot. These parameters can be set for the whole R session (until the parameters are changed) or can be included in one of the functions seen above (only affecting that function momentarily). The most common parameters are:

parameter	description
`pch`	point character or symbol.
`lty`	Line type
`lwd`	Line width.
`col`	Colour (of elements in the plot).
`bg`	Colour (of background).

Let us now see some examples. We include the code and the plot it generates.

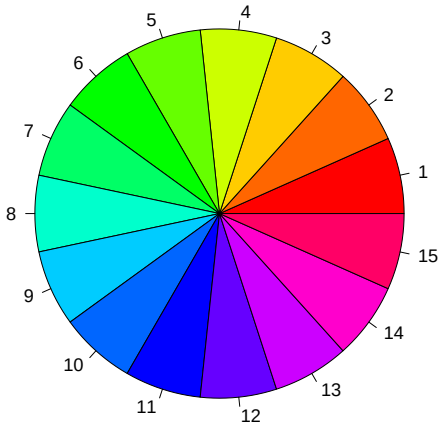



```
> x <- 1:15
> y <- rnorm(15)
> plot(x,y,type="o",pch=9,xlab="Consecutive values",
+ylab="Normally distributed values",col="blue",lwd=2)
> title("Plot example",col.main="pink3")
```




```
> hist(y,col="red",xlab="Normally distributed values",
+ ylab="Frequency")
```

Pie chart




```
> z<-rep(1,15)
> pie(z,col=rainbow(15))
> title("Pie chart",col.main="purple3")
```

There are many more functions for plots, and many external packages.

15 Exporting R graphics

If instead of showing the chart on the screen, we want to save it to a file, we have several options. The easiest option is from the R console. There, by clicking on the right button of the mouse, we can save the plot as a metafile, as postscript or a bitmap. Another easy options is when using Rstudio. In the "Plots" window, the chart can be saved as an image or as a PDF file.

If we want to have more control about how to do this, one can open a file before executing the functions to create the plot, using the functions `postscript()`, `pdf()`, `png()`, ... Another option is to use any screen device `x11()` (for Unix/Linux/Windows), `quartz()` (for MAC OS X) or the default device (the function `dev.cur()` shows the active device).

With any of the above options, we need to execute `dev.off()` at the end. The following example outputs a very simple plot to a file:

```
> WORKDIR <- "C:/MyFiles/"
> setwd(WORKDIR)
> pdf("myfile.pdf", width=7, height=7)
> plot(1:5)
> dev.off()
```

Other options are based on using the default device and export the chart using functions such as `dev.copy`, `dev.copy2pdf` or `dev.copy2eps`.

16 Control structures

In R we use curly brackets {} to group expressions. The most common control structures are:

- `if, else`: conditional

  ```
  if(<condition>) {
            ## do something
  } else {
            ## do something else
  }
  if(<condition1>) {
            ## do something
  } else if(<condition2>) {
            ## do something different
  } else {
            ## do something different
  }
  ```

  The `else` part is optional.

- Loops
  - `for`: executes a loop a given number of times. It uses a variable (the *iterator*) and assigns values successively using a sequence or vector. The syntax is : `for (name in values) expression`

```
> for(i in 1:10) cat(i,sep="\n")
1
2
3
4
5
6
7
8
9
10
```

  - `while`: executes a loop while the condition is true. The condition is checked first and if so then it executes the body of the loop, then the condition is checked and so on.

```
> count <- 0
> while(count < 10) {
+ print(count)
+ count <- count + 1
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
```

  - `repeat`: executate a loop infinitely. It can only be exited with `break`.
  - `break`: exits a loop.
  - `next`: goes to the next iteration of a loop.
  - `return`: returns from a function and whatever loops inside the function we may be in.

```
>count<-0
> repeat{
+        count<-count+1
+        if (count%%2==0){
+                next
+        } else {
+                print(count)
+                }
+        if (count>=9) {break}
+        }
[1] 1
[1] 3
[1] 5
[1] 7
[1] 9
```

Many of these structures are more commonly used in functions than in command-line interactions with R.

17 Creating functions and arranging code

Functions in R can be defined in the following way:

`function(<arguments>) {## body}`.

In R, functions are objects and, as we saw above, they can be passed as argument to other functions (R allows higher-order functions). As they are objects we can just see their definition at any moment:

```
> myfun <- function(x, y)
+ { if (x>y) return(-x) else return(y)}
> myfun(5,3)
[1] -5
> myfun
function(x, y) { if (x>y) return(-x) else return(y)}
```

We can also focus on the argument of a function with `args(name)`:

```
> args(myfun)
function (x, y)
NULL
```

Some functions can have a variable number of attributes. For instance the function `paste`:

```
> args(paste)
```

```
function (..., sep = " ", collapse = NULL)
NULL
```

The ... is used in R to denote any variable number of arguments. In this case `paste` is a function that allows the user to concatenate any number of strings, or even combine vectors of string, as shown in the following example:

```
> paste(c('a','b','c'),c(1,2),sep=":")
[1] "a:1" "b:2" "c:1"
>
> paste(c('a','b','c'),c(1,2))
[1] "a 1" "b 2" "c 1"
```

We can define functions with this variable number of arguments. For instance:

```
> myfun2<-function(x,y,...){paste(x,y,sep=":",...)}
> myfun2(c('a','b','c'),c(1 ,2))
[1] "a:1" "b:2" "c:1"
```

If we wanted to do the same for a variable number of vectors:

```
> myfun3<-function(...){paste(...,sep=":")}
> myfun3(c('a','b','c'),c(1,2),rep('AB',4))
[1] "a:1:AB" "b:2:AB" "c:1:AB" "a:2:AB"
```

As programs get larger than snippets of a few lines of code, we should arrange them modularly. We can use several files, and some can include other files. Typically, R code uses the extension .R. In order to include a file into the console (or into another file), we can use `source()`.

18 Packages

Finally, one of the strengths of R is its large number of libraries. Libraries are standardised under the CRAN system. There are many mirrors that can be used to download any of the official packages. This can be done on the menu of the R console or through code, as we see below.

First, if we want to use a package that we have not downloaded, then we will get an error:

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

It does not work as it is not installed. It can be installed manually on the R menu ("Package" -¿ "Install Packages") or it can be done automatically with `install.packages()`. For instance:

```
> install.packages("rocc")
Installing package into 'C:/Users....
also installing the dependencies 'bitops', ...

trying URL 'https://cran.rstudio.com/....
Content type 'application/zip' length 36009 ...
downloaded 35 KB
...

package 'bitops' successfully unpacked and MD5 ...
...

The downloaded binary packages are in
        C:\Users\...
```

Now it works!

```
> library(rocc)
Loading required package: ROCR
Loading required package: gplots

Attaching package: 'gplots'
The following object is masked from 'package:stats':
    lowess
```