Chapter 3
Basic Concepts and Data Organisation

Goals of this chapter

This chapter introduces the basic concepts of the R software (calculator mode, assignment operator, variables, functions, arguments) and the various data types and structures which can be handled by R.

SECTION 3.1

Your First Session

Launch R by double-clicking its icon on the Windows Desktop (or from the Start menu). At the end of the text displayed in the *R console*, you can see the **prompt symbol** >, inviting you to type in your first instruction in the R language.

```
R version 2.14.1 (2011-12-22)
Copyright (C) 2011 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

P. Lafaye de Micheaux et al., *The R Software: Fundamentals of Programming and Statistical Analysis*, Statistics and Computing 40, DOI 10.1007/978-1-4614-9020-3_3,

For example, type `"R is my friend"`, then validate by hitting the `ENTER` key (or `RETURN`). You will then get

```
> "R is my friend"
[1] "R is my friend"
```

As you can see, R is well behaved and kindly proceeds with your request. This will usually be the case—maybe R is trying to compensate for its lack of conviviality. We shall explain later on why R's reply starts with *[1]*.

3.1.1 R *Is a Calculator*

Like many other similar languages, R can easily replace all the functionalities of a (very sophisticated!) calculator. One of its major strengths is that it also allows calculations on arrays. Here are a few very basic examples.

```
> 5*(-3.2)          # Careful: the decimal mark must be a point (.)
[1] -16
> 5*(-3,2)          # otherwise, the following error is generated:

Error : ',' unexpected in "5*(-3,"

> 5^2               # Same as 5**2.
[1] 25
> sin(2*pi/3)
[1] 0.8660254
> sqrt(4)           # Square root of 4.
[1] 2
> log(1)            # Natural logarithm of 1.
[1] 0
> c(1,2,3,4,5)      # Creates a collection of the first five
                    # integers.
[1] 1 2 3 4 5
> c(1,2,3,4,5)*2    # Calculates the first five even numbers.
[1]  2  4  6  8 10
```

Tip

Any R code after the symbol "#" is considered by R as a comment. In fact, R does not interpret it.

You can now exit the R software by typing the following instruction: `q()`.

You are asked whether you wish to save an image of the session. If you answer yes, the commands you typed earlier will be accessible again next time you open R, by using the "up" and "down" keyboard arrows.

3.1.2 Displaying Results and Variable Redirecting

As you have probably noticed, R responds to your requests by displaying the result obtained after evaluation. **This result is displayed, then lost.** At first, this might seem sensible, but for more advanced uses, it is useful to redirect the R output to your request, by storing it in a variable: this operation is called **assigning the result to a variable**. Thus, an assignment evaluates an expression but does not display the result, which is instead stored in an object. To display the result, all you need to do is type the name of that object, then hit `ENTER`.

To make an assignment, use the **assignment arrow** `<-`. To type the arrow `<-`, use the lesser than symbol (<) followed by the minus symbol (-).

To create an object in R, the syntax is thus
`Name.of.the.object.to.create <- instructions`

For example,

```
> x <- 1        # Assignment.
> x             # Display.
[1] 1
```

We say that the value of `x` is 1, or that we have assigned 1 to `x` or that we have stored in `x` the value 1. Note that the assignment operation can also be used the other way around `->`, as in

```
> 2 -> x
> x
[1] 2
```

Warning

The symbol = can also be used, but its use is less general and is therefore not advised. Indeed, mathematical equality is a symmetrical relation with a specific meaning, very different to assignment. Furthermore, there are cases where using the symbol = does not work at all.

Tip

Note that a pair of brackets allows you to assign a value to a variable and display the evaluation result at the same time:

```
> (x <- 2+3)
[1] 5
```

If a command is not complete at the end of a line, R will display a different prompt symbol, by default the plus sign (+), on the second line and on following lines. R will continue to wait for instructions until the command is syntactically complete.

```
> 2*8*10+exp(1)
[1] 162.7183
> 2*8*
+ 10+exp(
+ 1)
[1] 162.7183
```

Warning

Here are the **rules for choosing a variable name** in R: a variable name can only include alphanumerical characters as well as the dot (.); variable names are *case sensitive*, which means that R distinguishes upper and lower case; a variable name may not include white space or start with a digit, unless it is enclosed in quotation marks "".

3.1.3 Work Strategy

- Take the habit of storing your files in a folder reserved to this effect (you could call it `Rwork`). We also advise you to type all your R commands in a script window called *script* or *R editor*, accessible through the "File/New script" menu. Open a new script window, click on the "Windows/Side by side" menu, then copy the script below:

```
x <- 5*(-3.2)
5^2
sin(2*pi/3)
sqrt(4)
c(1,2,3,4,5)
z <- c(1,2,3,4,5)*2
```

Mac

On a Mac, the menu is "File/New Document", and it is not possible to lay the windows side by side.

At the end of your session, you can save this script in the folder `Rwork`, for example, as `myscript.R`, and reopen it during a later session from the menu "File/Open a script" (or on a Mac "File/Open Document").

- You can then use the key combinations `CTRL+A` (`COMMAND+A` on a Mac) to select all the instructions, then `CTRL+R` (`COMMAND+ENTER` on a Mac) to paste and execute them in one step in the R console. You can also execute a single line of R instructions from the script by hitting `CTRL+R` when the blinking cursor is on the relevant line of the script window.

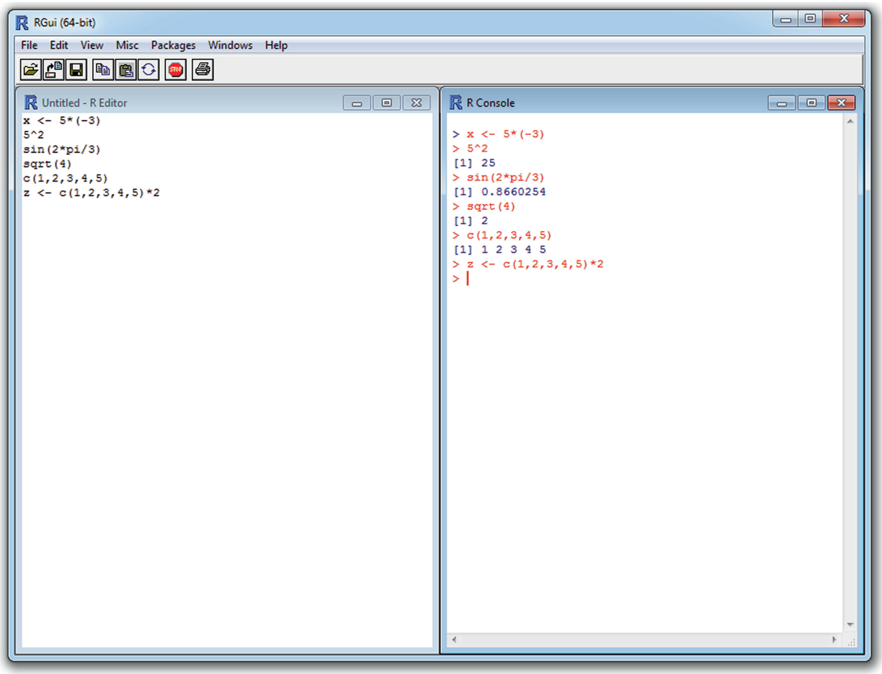

Fig. 3.1: The script window and the command console

Tip

Note in Fig. 3.1 the presence of the red STOP button that lets you interrupt a calculation that would last too long.

You can also use the function `source()` from the `R` console to read and execute the content of your file. This helps prevent overloading the console, as we will see later. You may find it useful to proceed as follows:

(a) Clicking once in the *R console* window.
(b) Going to the menu "File/Change current directory" ("Misc/Change work directory" on a Mac).
(c) Exploring your file system and selecting the folder `Rwork`.
(d) Typing in the console `source("myscript.R")`. Note that for the above example, the use of this instruction will not produce any output. The following Do it yourself will clarify this point.

Do it yourself

Begin to create a folder called Rwork in your home directory. Then, type in and save in an R script the preceding instructions. The file containing the R script will be called myscript.R and will be put in Rwork. Now close then reopen R. Next, type the following instructions in the R console:

```
rm(list=ls()) # Delete all existing objects.
ls()          # List existing objects.
source("myscript.R")
ls()
x
z
```

Note that the `source()` function has permitted to execute the preceding instructions. You may have noticed that the computations which have not been redirected into variables have not been printed. So their result is lost. Change your script and add the following instructions at the end of it:

```
print(2*3)
print(x)
```

Save it, then source it. What happened?

- Take the habit of using the online R help. The help is very complete and in English. You can reach it with the function `help()`. For example, type `help(source)` to get help about the function `source()`.

See also

All these notions will be examined in further detail in Chaps. 6 and 9.

Tip

Two good code editors are RStudio, available at `http://www.rstudio.com`, and Tinn-R (Windows only), available at `http://www.sciviews.org/Tinn-R`. They offer a better interaction between a script's code and its execution. They also provide syntactic colouring of the code.

Linux

Under Linux, note that the editors JGR and Emacs/ESS are available.

See also

You can consult the list of R editors on the webpage `http://www.sciviews.org/_rgui/projects/Editors.html`.

Do it yourself

The body mass index (BMI) is used to determine a person's corpulence. It is calculated using the formula

$$\text{BMI} = \frac{\text{Weight (kg)}}{\text{Height}^2 \text{ (m)}}.$$

Calculate your BMI. You simply need to type the following lines in your script window:

```
# You can type 2 instructions
# on the same line thanks to the symbol ;
My.Weight <- 75 ; My.Height <- 1.90
My.BMI <- My.Weight/My.Height^2
My.BMI
```

Execute this script by using the work strategy mentioned earlier. You can then modify this script to calculate your own BMI.

We propose a function to visualize your corpulence type. Execute the following instructions:

```
source("http://www.biostatisticien.eu/springeR/BMI.R",
encoding="utf8")
display.BMI(My.BMI)
```

You will learn how to program this kind of result in later chapters.

3.1.4 Using Functions

We have already encountered a few functions: `sin()`, `sqrt()`, `exp()` and `log()`. The base version of R includes many other functions, and thousands of others can be added (by installing packages or by creating them from scratch).

Note that a function in R is defined by its **name** and by the list of its **parameters**. Most functions output a **value**, which can be a number, a vector, or a matrix.

Using a function (or **calling** or **executing** it) is done by typing its name followed, in brackets, by the list of (formal) arguments to be used. Arguments are separated by commas. Each argument can be followed by the sign = and the value to be given to the argument. This value of the formal argument will be called effective argument, call argument or sometimes entry argument.

We will therefore use the instruction

```
functionname(arg1=value1,arg2=value2,arg3=value3)
```

where `arg1, arg2, ...` are called the arguments of the function, whereas `value1` is the value given to the argument `arg1`, etc. Note that you do not necessarily need to indicate the names of the arguments, but only the values, as long as you follow their order.

For any `R` function, some arguments must be specified and others are optional (because a default value is already given in the code of the function).

Warning

Do not forget the brackets when you call a function. A common mistake for beginners is forgetting the brackets:

```
> factorial
function (x)
gamma(x + 1)
<environment: namespace:base>
> factorial(6)
[1] 720
```

The output to the first instruction gives the code (i.e. the recipe) of the function, whereas the second instruction executes that code. This is also true for functions which do not require an argument, as shown in the following example:

```
> date()
[1] "Wed Jan  9 16:04:32 2013"
> date
function ()
.Internal(date())
<environment: namespace:base>
```

Obviously, this is not the place to comment the code of these functions.

To better understand how to use arguments, take the example of the function `log(x,base=exp(1))`. It can take two arguments: `x` and `base`.

The argument `x` must be specified: it is the number of which we wish to calculate the logarithm. The argument `base` is optional, since it is followed with the symbol = and the default value `exp(1)`.

Tip

An argument which is not followed with the symbol = must be specified. A parameter is optional if it is followed with =.

In the following code, R will calculate the *natural* logarithm of the number 1, since the base argument is not specified:

```
> log(1)
[1] 0
```

Note

For some functions, no argument needs to be specified, for example, `matrix`, which we shall encounter later on.

One last important note is that **you can call a function by playing with the arguments in several different ways**. This is an important feature of R which makes it easier to use, and you will find it useful to understand this principle. To calculate the natural logarithm of 3, any of the following expressions can be used:

```
log(3)                      log(3,base=exp(1))
log(x=3)                    log(3,exp(1))
log(x=3,base=exp(1))        log(base=exp(1),3)
log(x=3,exp(1))             log(base=exp(1),x=3)
```

Warning

Note that calling

```
log(exp(1),3)
```

will calculate the logarithm of `exp(1)` in base 3.

Finally, recall that we have been able to see the code for the function `factorial()`:

```
> factorial
function (x)
gamma(x + 1)
<environment: namespace:base>
```

This function was defined by the R developers with the following instructions:

```
> factorial <- function(x) gamma(x+1)
```

It is very easy to code a new function in R, by using the function `function()`. For example, here is how to code a function which takes two arguments n and p and calculates the binomial coefficient $\binom{n}{p} = \frac{n!}{p!(n-p)!}$:

```
> binomial <- function(n,p) factorial(n)/(factorial(p)*
+                             factorial(n-p))
```

You can then use this new function as any other R function:

```
> binomial(4,3)
[1] 4
```

We shall study in much further detail how to create more elaborate functions in Sect. 5.8 and in Chap. 8.

> **Note**
>
> In fact, there already exists an R function to compute the Newton binomial coefficient. This is the function `choose()` that works more efficiently, especially for big numbers.

SECTION 3.2

Data in R

R, like most computer languages, can handle classical data types. R is actually able to automatically recognize data types according to the format of the input. One of the main strengths of R is its ability to organize data in a structured way. **This will turn out to be very useful for many statistical procedures we will study later on**.

3.2.1 Data Nature (or Type, or Mode)

Data "types" can be handled using the functions `mode()` and `typeof()`, which only differ in very subtle ways which we shall ignore.

> **Note**
>
> The function `class()` is more general: it is used to handle both data type and structuring. We shall study it later on. For ease of understanding, we shall use the command `typeof()`.

The various types (or modes) of data are now presented.

3.2.1.1 Numeric Type (`numeric`)

There are two numeric types: integers (`integer`) and real numbers (`double`).

If you enter

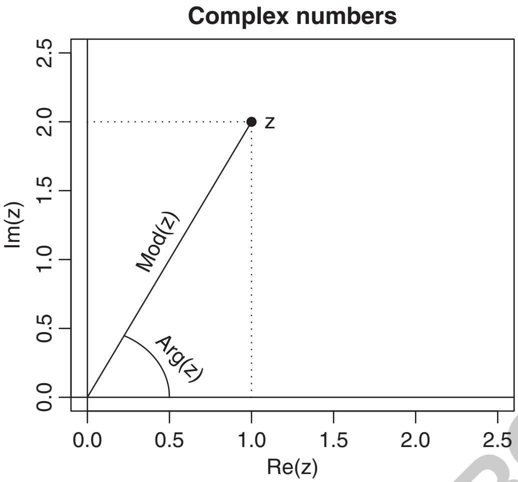

Fig. 3.2: Characteristics of a complex number

```
> a <- 1
> b <- 3.4
> c <- as.integer(a)
> typeof(c)
[1] "integer"
```

the variables `a` and `b` are of the type `"double"`, and the variable `c` has the same value as `a`, except that it has been forced to be of the type `"integer"`. This is useful because a vector of `"integer"`s takes up less memory space than a vector of `"double"`s of the same length. Instructions starting with `as.` are very common in R to convert data into a different type. We will see in the Sect. 3.2.2.1 how to check that an object's type is numeric.

3.2.1.2 † Complex Type (`complex`)

A complex number is created, thanks to the letter `i`. The functions `Re()` for real part, `Im()` for imaginary part, `Mod()` for modulus and `Arg()` for argument can be used (Fig. 3.2).

Here are a few examples:

```
> 1i
[1] 0+1i
> z <- 1+2i
> typeof(z)
[1] "complex"
> is.complex(z)  # To know whether an object is of the complex
                 # type.
[1] TRUE
> Re(z)
[1] 1
```

```
> Im(z)
[1] 2
> Mod(z)
[1] 2.236068
> Arg(z)
[1] 1.107149
```

3.2.1.3 Boolean or Logical Type (`logical`)

The type `logical()` is the result of a logical operation. It can take the values `TRUE` or `FALSE`. Here are a few instructions to create logical values:

```
> b>a
[1] TRUE
> a==b
[1] FALSE
> is.numeric(a)
[1] TRUE
> is.integer(a)
[1] FALSE
> x <- TRUE
> is.logical(x)
[1] TRUE
```

Warning

`TRUE` and `FALSE` can also be entered in a more condensed form by typing `T` and `F`, respectively. But this should not be encouraged.

When needed, this data type is naturally converted to numeric without having to specify the conversion: `TRUE` is worth `1` and `FALSE` is worth `0`. The following example illustrates this point:

```
> TRUE + T + FALSE*F + T*FALSE + F
[1] 2
```

3.2.1.4 Missing Data (`NA`)

A missing or undefined value is indicated by the instruction `NA` (for *non-available*). Several functions exist to handle this data type. In fact, R considers this data type as a constant logical value. Strictly speaking, it is therefore not a data type. Here are a few examples which use the instruction `NA`:

```
> x <- c(3,NA,6)
> is.na(x)
[1] FALSE  TRUE FALSE
> mean(x)                # Trying to calculate the mean of x.
```

```
[1] NA
> mean(x,na.rm=TRUE)    # The na.rm argument means that NA's
                        # should be ignored (NA.remove).
[1] 4.5
```

This is a very important notion when it comes to reading statistical data files. We shall examine it in further detail in Chap. 5.

Warning

Do not mistake NA for the reserved word NaN, which means *not a number*:

```
> 0/0
[1] NaN
```

Note also that the following instruction does not output NaN but infinity, represented in R with the reserved word Inf.

```
> 3/0
[1] Inf
```

3.2.1.5 Character String Type (character)

Any information between quotation marks (single ' or double ") corresponds to a character string:

```
> a <- "R is my friend"
> mode(a)
[1] "character"
> is.character(a)
[1] TRUE
```

Conversions into a character string from another type are possible. Converting a character string into another type is possible as long as R can correctly interpret the content inside the quotations marks. Note that some conversions are done automatically. Here are a few examples:

```
> as.character(2.3)         # Conversion into a character string.
[1] "2.3"
> b <- "2.3"
> as.numeric(b)             # Conversion from a character string.
[1] 2.3
> as.integer("3.4")         # Conversion from a character string.
[1] 3
> c(2,"3")                  # Automatic conversion.
[1] "2" "3"
> as.integer("3.four")      # Impossible conversion.
[1] NA
```

Note

The differences between single and double quotation marks are given in Chap. 5.

3.2.1.6 † Raw Data (`raw`)

In R, it is possible to work directly with bytes (displayed in hexadecimal format). This can sometimes be useful when reading certain files in binary format. We shall see examples in Chap. 7.

```
> x <- as.raw(15)
> x
[1] 0f
> mode(x)
[1] "raw"
```

Summary

Table 3.1: The various data types in R

Data type	Type in R	Display
Real number (integer or not)	`numeric`	`3.27`
Complex number	`complex`	`3+2i`
Logical (true/false)	`logical()`	`TRUE` or `FALSE`
Missing	`logical()`	`NA`
Text (string)	`character`	`"text"`
Binary	`raw`	`1c`

Tip

The function `storage.mode()` get or set the type or storage mode of an object.

3.2.2 Data Structures

In R, you can organize (structure) the various data types defined above (Table 3.1). The structures we are about to present can be accessed or created with the function `class()` (Table 3.2).

3.2.2.1 Vectors (`vector`)

This is the simplest data structure. It represents a **sequence of data points of the same type**. A vector can be created with the function `c()` (for collection or concatenation). Other functions such as `seq()` or a colon `:` can also be used to create a vector. Note that when creating a vector, it is possible to mix data of different types. R will then make an implicit conversion into the more general data type, as shown in the following example:

```
> c(3,1,7)
[1] 3 1 7
> c(3,TRUE,7)
[1] 3 1 7
> c(3,T,"7")
[1] "3"    "TRUE" "7"
> seq(from=0,to=1,by=0.1)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> seq(from=0,to=20,length=5)
[1]  0  5 10 15 20
> vec <- 2:36
> vec
 [1]  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
[20] 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
```

Warning

The indications *[1]* and *[20]* give the rank in the vector `vec` of the element they precede.

Note that it is possible to "name" the elements of a vector using the function `names()`.

```
> vec <- c(1, 3, 6, 2, 7, 4, 8, 1, 0)
> names(vec) <- letters[1:9] # 9 first letters of the alphabet.
> vec
a b c d e f g h i
1 3 6 2 7 4 8 1 0

> is.vector(vec)
[1] TRUE
> x <- 1:3
> x
[1] 1 2 3
> y <- c(1,2,3)
> y
[1] 1 2 3
> class(x)
[1] "integer"
> class(y)
[1] "numeric"
```

One would actually expect to see appear "vector of doubles" or "vector of integers" instead of "numeric" or "integer", but no software is perfect!

Advanced users

Note that the instructions c() and : give the same output, but that x and y are stored internally in different ways. The type integer uses less memory than the type numeric.

3.2.2.2 Matrices (matrix) and Arrays (array)

These two notions are generalizations of the vector notion: they represent sequences with two indices for matrices and with multiple indices for arrays. As with vectors, **elements must be of the same type, otherwise implicit conversions will occur**.

The following instruction

```
> X <- matrix(1:12,nrow=4,ncol=3,byrow=TRUE)
> X
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
```

creates (and stores in the variable X) a matrix with four rows and three columns, filled by row (byrow =TRUE) with the elements of the vector 1:12 (e.g., the twelve first integers).
Similarly, a matrix can be filled by column (byrow=FALSE).

```
> Y <- matrix(1:12,nrow=4,ncol=3,byrow=FALSE)
> Y
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> class(Y)
[1] "matrix"
```

The function array() is used to create multidimensional matrices with more than two dimensions, as shown in the following figure (for a three-dimensional array) (Fig. 3.3):

```
> X <- array(1:12,dim=c(2,2,3))
> X
, , 1
     [,1] [,2]
[1,]    1    3
```

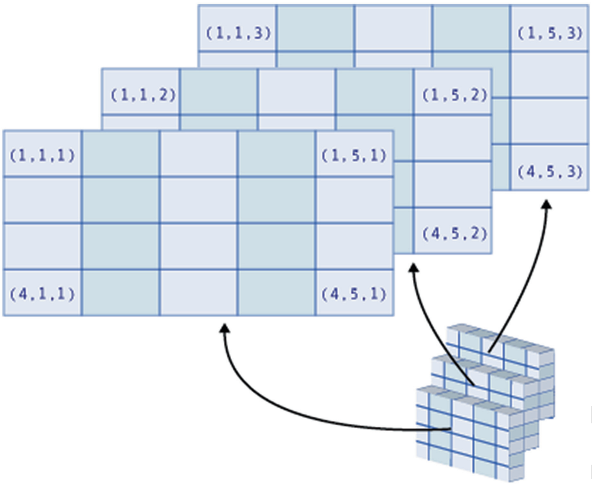

Fig. 3.3: Illustration of an array

```
[2,]    2    4
, , 2
     [,1] [,2]
[1,]    5    7
[2,]    6    8
, , 3
     [,1] [,2]
[1,]    9   11
[2,]   10   12
> class(X)
[1] "array"
```

Warning

Arrays with more than three dimensions can be created, thanks to the argument `dim`, which can be of length greater than 3.

3.2.2.3 Lists (`list`)

The most flexible and richest structure in R is the list. Unlike the previous structures, lists can **group together in one structure data of different types** without altering them. Generally speaking, each element of a list can thus be a vector, a matrix, an array or even a list. Here is a first example:

```
> A <- list(TRUE,-1:3,matrix(1:4,nrow=2),c(1+2i,3),
+           "A character string")
> A
```

```
[[1]]
[1] TRUE
[[2]]
[1] -1  0  1  2  3
[[3]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4
[[4]]
[1] 1+2i 3+0i
[[5]]
[1] "A character string"
> class(A)
[1] "list"
```

In such a structure, with heterogeneous data types, element ordering is often completely arbitrary. Elements can therefore be explicitly named, which makes the output more user-friendly. Here is an example:

```
> B <- list(my.matrix=matrix(1:4,nrow=2),
+             my.complex.numbers=c(1+2i,3))
> B
$my.matrix
     [,1] [,2]
[1,]    1    3
[2,]    2    4
$my.complex.numbers
[1] 1+2i 3+0i
> list1 <- list(my.complex.number=1+1i,my.logical.value=FALSE)
> list2 <- list(my.string="I am learning R",my.vector=1:2)
> C <- list("My first list"=list1,My.second.list=list2)
> C
$`My first list`
$`My first list`$my.complex.number
[1] 1+1i
$`My first list`$my.logical.value
[1] FALSE
$My.second.list
$My.second.list$my.string
[1] "I am learning R"
$My.second.list$my.vector
[1] 1 2
```

See also

Naming elements will make it easier to extract elements from a list (see Chap. 5, p. 106).

3.2.2.4 The Individual×Variable Table (`data.frame`)

The individual×variable table is the quintessential structure in statistics. In R, this notion is expressed by a data.frame. Conceptually speaking, it is a matrix with each line corresponding to an individual and each column corresponding to a variable measured on the individuals. **Each column represents a single variable, which must be of the same type across all individuals.** The columns of the data matrix can have names. Here is an example of a data.frame creation:

```
> BMI <- data.frame(Gender=c("M","F","M","F","M","F"),
+      Height=c(1.83,1.76,1.82,1.60,1.90,1.66),
+      Weight=c(67,58,66,48,75,55),
+      row.names=c("Jack","Julia","Henry","Emma","William","Elsa"))
> BMI
        Gender Height Weight
Jack         M   1.83     67
Julia        F   1.76     58
Henry        M   1.82     66
Emma         F   1.60     48
William      M   1.90     75
Elsa         F   1.66     55
> is.data.frame(BMI)
[1] TRUE
> class(BMI)
[1] "data.frame"
> str(BMI)
'data.frame':        6 obs. of  3 variables:
 $ Gender: Factor w/ 2 levels "F","M": 2 1 2 1 2 1
 $ Height: num  1.83 1.76 1.82 1.6 1.9 1.66
 $ Weight: num  67 58 66 48 75 55
```


Note

The `str()` function enables one to display the structure of each column of a data.frame.

Advanced users

A data.frame can be seen as a list of vectors of identical length. This is actually how R stores a data.frame internally.

```
> is.list(BMI)
[1] TRUE
```

3.2.2.5 Factors (`factor`) and Ordinal Variables (`ordered`)

In R, character strings can be organized in a more astute way, thanks to the function `factor()`:

```
> x <- factor(c("blue","green","blue","red",
+                   "blue","green","green"))
> x
[1] blue  green blue  red   blue  green green
Levels: blue green red
> levels(x)
[1] "blue"  "green" "red"
> class(x)
[1] "factor"
```

Tip

The function `cut()` enables one to recode a continuous variable into a factor.

```
> Poids <- c(55,63,83,57,75,90,73,67,58,84,87,79,48,52)
> cut(Poids,3)
 [1] (48,62] (62,76] (76,90] (48,62] (62,76] (76,90] (62,76]
 [8] (62,76] (48,62] (76,90] (76,90] (76,90] (48,62] (48,62]
Levels: (48,62] (62,76] (76,90]
```

Factors can of course be used in a data.frame.
R indicates the different *levels* of the factor. The function `factor()` should thus be used to store qualitative variables. For ordinal variables, the function `ordered()` is better suited:

```
> z <- ordered(c("Small","Tall","Average","Tall","Average",
+        "Small","Small"),levels=c("Small","Average","Tall"))
> class(z)
[1] "ordered" "factor"
```

The `levels` argument of the function `ordered` is used to specify the order of the variable's modalities.

See also

Examples of uses of these two functions are given in Chap. 11, pp. 341 and 342.

Tip

The function gl() generates factors by specifying the pattern of their levels:

```
> gl(n = 2,k = 8,labels = c("Control", "Treat"))
 [1] Control Control Control Control Control Control Control
 [8] Control Treat   Treat   Treat   Treat   Treat   Treat
[15] Treat   Treat
Levels: Control Treat
```

In the above instruction, n and k are two integers, the first one giving the number of levels and the second one the number of replications.

Advanced users

A vector of character strings can be organized in a more efficient way by taking into account repeated elements. This approach allows better management of the memory: each element of the factor or of the ordinal variable is in fact coded as an integer.

3.2.2.6 Dates

R can be used to structure the data representing dates, using the as.Date() function for example.

```
> dates <- c("92/27/02", "92/02/27", "92/01/14",
+            "92/02/28", "92/02/01")
> dates <- as.Date(dates, "%y/%m/%d")
> dates
[1] NA           "1992-02-27" "1992-01-14" "1992-02-28"
[5] "1992-02-01"
> class(dates)
[1] "Date"
```

We will return in detail on the functions for manipulating dates in Chap. 5.

3.2.2.7 Time Series

When data values are indexed by time, it may be useful, using the ts() function, to organize them into an R structure that reflects the temporal aspect of these data.

```
> ts(1:10, frequency = 4, start = c(1959, 2)) # 2nd Quarter of
                                              # 1959.
     Qtr1 Qtr2 Qtr3 Qtr4
1959         1    2    3
1960    4    5    6    7
1961    8    9   10
```

See also

The reader may consult with profit the book [40] which outlines the basic techniques for modelling time series, present the R functions to use for these models and give applications of these functions on several real data sets.

Summary

Table 3.2: The various data structures in R

Data structure	Instruction in R	Description
Vector	`c()`	Sequence of elements of the same nature
Matrix	`matrix()`	Two-dimensional table of elements of the same nature
Multidimensional table	`array()`	More general than a matrix; table with several dimensions
List	`list()`	Sequence of R structures of any (and possibly different) nature
Individual×variable table	`data.frame()`	Two-dimensional table where a row represents an individual and a column represents a variable (numerical or factor). The columns can be of different natures, but must have the same length
Factor	`factor()`, `ordered()`	Vector of character strings associated with a modality table
Dates	`as.Date()`	Vector of dates
Time series	`ts()`	Time series, containing the values of a variable observed at several time points

Memorandum

`<-`, `->`: variable assignment arrows
`mode()`, `typeof()`: gives the nature of an object
`is.numeric()`: determine whether an object is numerical
`TRUE`, `FALSE`, `is.logical()`: True, False, determine whether an object is a Boolean
`is.character()`: determine whether an object is a character string
`NA`, `is.na()`: missing value, determine whether a value is missing
`class()`: determine the structure of an object
`c()`: create a sequence of elements of the same nature
`matrix()`, `array()`: create a matrix, a multidimensional table
`list()`: create a list (collection of different structures)
`data.frame()`: create an individual×variable table
`factor()`: create a factor

Exercises

3.1- What is the output of this instruction: `1:3^2` ?
3.2- What is the output of this instruction: `(1:5)*2` ?
3.3- What is the output of these instructions: `var<-3`? `Var*2`?
3.4- What is the output of these instructions: `x<-2`? `2x<-2*x`?
3.5- What is the output of these instructions: `root.of.four <- sqrt(4)`? `root.of.four`?
3.6- What is the output of these instructions: `x<-1`? `x< -1`?
3.7- What is the output of this instruction: `An even number <- 16`?
3.8- What is the output of this instruction: `"An even number" <- 16`?
3.9- What is the output of this instruction: `"2x" <- 14`?
3.10- What is the output of this instruction: `An even number`?
3.11- Two symbols have been removed from this R output. What are they?

```
> 2
+
[1] 6
```

3.12- What is the output of this instruction: `TRUE + T +FALSE*F + T*FALSE +F`?
3.13- Name the five data types in R.
3.14- Give the R instruction which gives the following output:

```
> X
     [,1] [,2]  [,3]
[1,]    1    5     9
[2,]    2    6    10
[3,]    3    7    11
[4,]    4    8    12
```

3.15- Name the data structures (classes) available in R.

Worksheet

Study of Body Mass Index

We wish to analyze the characteristics of a sample of children. These children went through a medical examination in their first year of kindergarten in 1996–1997 in schools in Bordeaux (South West France). The sample below contains information on ten children between the ages of 3 and 4.

The following information is available for each child:

- gender: G for girls and B for boys;
- whether their school is in a ZEP (*zone d'éducation prioritaire*: area targeted for special help in education, recognized as socially deprived): Y for yes and N for no;
- age in years and months (two variables: one for years and one for months);
- weight in kg, rounded to the nearest 100 g;
- Height in cm, rounded to the nearest 0.5 cm.

Name	Edward	Cynthia	Eugene	Elizabeth	Patrick	John	Albert	Lawrence	Joseph	Leo
`Gender`	G	G	B	G	B	B	B	B	B	B
`ZEP`	Y	Y	Y	Y	N	Y	N	Y	Y	Y
`Weight`	16	14	13.5	15.4	16.5	16	17	14.8	17	16.7
`Years`	3	3	3	4	3	4	3	3	4	3
`Months`	5	10	5	0	8	0	11	9	1	3
`Height`	100.0	97.0	95.5	101.0	100.0	98.5	103.0	98.0	101.5	100.0

In statistics, it is of the utmost importance to know the type of the variables under study: qualitative, ordinal or quantitative. These types can be specified in `R`, thanks to the structure functions we introduced earlier in this chapter.

Try the following manipulations under `R`. Remember to use the work strategy we presented at the beginning of the chapter.

3.1- Choose the best `R` function to save the data from each variable in vectors which you will call `Individuals`, `Weight`, `Height` and `Gender`.

3.2- Where possible, calculate the mean of the variables.

3.3- Calculate the BMI of the individuals. Group the results in a vector called `BMI` (be careful of the units).

3.4- Group these variables in the R structure which seems most appropriate.
3.5- Use R's online help to get information on the `plot()` function.
3.6- Make a scatter plot of `Weight` as a function of `Height`. Remember to add a title to your graph and to label your axes.