An Introduction to Calibration Techniques in Sample Surveys
May 23-26 2016, ADB, Metro Manila, Philippines

An introduction to R: Data objects manipulation

Diego Zardetto
World Bank STC for research

The R objects used to store data structures are:

- `vector`: vector (NOT algebraic)
- `factor`: (vector for categorical variables)
- `matrix`: matrix
- `array`: multi-dimensional data object
- `list`: list
- `data.frame`: list organized as a matrix (units x variables)

Vector

The R `vector` is NOT the algebraic vector (column vector).
In R a `vector` is the simplest data object: it is an ordered collection of elements of the same type (numbers, character strings, logicals…).
There are five types of `vector`, according to the values they store:

Values in the vector	Attributes		
	mode	**class**	**storage mode**
Integers numbers	"numeric"	"integer"	"integer"
Real numbers	"numeric"	"numeric"	"double"
Txt string	"character"	"character"	"character"
Logical	"logical"	"logical"	"logical"
Complex numbers	"complex"	"complex"	"complex"

vector of numbers (integer or real numbers)

The function `c()` combines (concatenates) the arguments to form an R vector

```
> x <- c(1, 2, 3, 4, 5, 6)

> x
[1] 1 2 3 4 5 6

> length(x)
 6

> mode(x)
[1] "numeric"
```

c() admits an arbitrary number of objects; as arguments can have others R vectors

```
> y <- c(x, 10, x)   #x then a "10" and then x again
> y
 [1]  1  2  3  4  5  6 10  1  2  3  4  5  6
```

c() can be used to concatenate two existing vectors (of the same type)

```
> x <- c(1,2,3,4)

> y <- c(10, 20, 30)

> z <- c(y, x)
> z
[1] 10 20 30  1  2  3  4
```

Sequences of integers

```
> x <- 1:4    #the same as  x<-c(1,2,3,4)

> x <- 4:1    #  x<-c(4,3,2,1)
```

The function `rep()` is very useful

```
> rep(3, 6)          #repeats 6 times the value 3

> rep(1:3, 2)        #repeats 2 times the vector 1:3

> rep(1:3, each=2)   #repeats 2 times each elem. of 1:3

> rep(1:3, 4:2)      #repeats 1 4 times, 2 3 times and 3 2
                      times
```

Sequences of real numbers can be created with `seq()`

```
> x <- seq(from=1, to=10, by=0.5)
> x
 [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0

> x <- seq(1,3,0.2)
> x
 [1] 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0
```

vector of logicals

It is a vector whose elements are "TRUE" (T or 1) and/or "FALSE" (F or 0)

```
> x <- c(1:3, NA)
> x
[1]  1  2  3 NA

> y <- is.na(x)
> y
[1] FALSE FALSE FALSE  TRUE

> length(y)
[1] 4

> mode(y)
[1] "logical"
```

vector of TXT strings

```
> x <- c("a", 'b')
> x
[1] "a" "b"

> length(x)
[1] 2

> mode(x)
[1] "character"
```

The function `nchar()` counts the number of characters of each string in the vector

```
> x <- c("Mick","Jagger","a", '', ' ')
> nchar(x)
[1] 4 6 1 0 1
```

The function `paste()` combines character/numbers to form complex strings

```
> paste("VAR", 1:4, sep="")
[1] "VAR1" "VAR2" "VAR3" "VAR4"

> paste("VAR", 1:4, sep=".")
[1] "VAR.1" "VAR.2" "VAR.3" "VAR.4"
```

Some useful functions to deal with txt strings:

```
substring()     #selects pieces of string using position

tolower()       #converts to lower cases

toupper()       #converts to UPPER CASES

match()         #searches for a string in a vector

grep()          #as before but admits partial matches

pmatch()        #search for a (even partial) match starting
                #from the beginning of the strings

sub()           #substitutes a piece of a string
```

%in% returns a logical vector indicating the presence of a string in a vector

Es.

```
> x <- c("R", "course", "in", "Manila")
> match("R",x)
[1] 1

> match("Man",x)
[1] NA

> match("Manila",x)
[1] 4

> pmatch("Man",x)
[1] 4

> pmatch("ila",x)
[1] NA
```

```
> grep("ila",x)
[1] 4


> "R" %in% x
[1] TRUE


> c("R","course","Philippines") %in% x
[1]  TRUE  TRUE FALSE


> x<-c("January","February")
> substr(x,1,3)

[1] "Jan" "Feb"
```

Selecting elements of a vector

The elements of a `vector` can be selected by referring to their position in the vector:

name_vector[*position*]

```
> mon <- month.abb
> mon
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
 [10] "Oct" "Nov" "Dec"

> mon[6]  #element in pos. 6
[1] "Jun"

> mon[-6]   #the element in pos. 6 is dropped
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jul" "Aug" "Sep" "Oct"
 [10] "Nov" "Dec"
```

```
> mon[1:6]     # first 6 elements of mesi
[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun"

> mon[-(1:6)]    #excludes the first 6 elements of mon
[1] "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"

> mon[c(2,4,6)]  #2nd,4th and 6th element from mon
[1] "Feb" "Apr" "Jun"

> mon[-c(2,4,6)]
[1] "Jan" "Mar" "May" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"

> mon[c(2,4,1,2)]
[1] "Feb" "Apr" "Jan" "Feb"
```

The selection of elements of a `vector` can be done according to given conditions

```
> x <- 1:8

> y <- x>6
> y
[1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE

> x[y]
[1] 7 8
```

The same result can be obtained by typing directly:

```
> x[x>6]
[1] 7  8
```

Logical operators:

"==" equality
">" greater
">=" greater or equal to
"<" lower
"<=" lower or equal to
"!" NOT

```
> x <- c(NA, 1:4, NA)
> x
[1] NA  1  2  3  4 NA

> y <- x[!is.na(x)]
> y
[1] 1 2 3 4
```

Arithmetic operations with numeric vectors

The arithmetic operation involving two vectors are carried out element by element

```
> y <- runif(4)
[1] 0.97125301 0.28373228 0.04466238 0.51412020

> y+10
[1] 10.97125 10.28373 10.04466 10.51412

> y*10
[1] 9.7125301 2.8373228 0.4466238 5.1412020
```

```
> x<-1:3; y<- c(1,10,100)
> x*y
[1]   1  20 300
```

* much care should be used when `vectors` of different length are involved

```
> x<-1:5
> length(x)
[1] 5
> y<-c(1,10,100)
> length(y)
[1] 3
> x*y
[1]   1  20 300   4  50
Warning message:
longer object length
      is not a multiple of shorter object length in: x * y
```

In practice, R completes the operation by recycling the shortest `vector` in order to obtain a `vector` of the same length of the other one.

Note that the warning message does NOT appear when the length of the `vector` is a multiple of the length of the other one

```
> x <-1:6
> y
[1]   1  10 100
> x*y
[1]   1  20 300   4  50 600
```

The same as:

```
> x*c(y, y)
```

Some useful mathematical functions are:

`log10, log, exp, sin, cos, tan, sqrt, abs, round, min, max`

others useful functions:

```
> sum(x)   #sum the elements of x
[1] 112.3

> prod(x)   #product of the elem. of x
[1] 121.2

> x <- c(1.2, 0.1, 10, 101)
> range(x)   #the same as  c(min(x), max(x))
[1]   0.1 101.0

> mean(x)   #average of the elements of x
[1] 28.075
```

Note that formulas in R can be written as they are found in the textbooks

Ex.: $\sum_{i=1}^{n}(x_i - \bar{x})^2$

```
> sum( (x-mean(x))^2)
[1] 7149.627


> var(x)    #sample variance
[1] 2383.209

> sd(x)   #standard deviation
[1] 48.81812

> median(x)   #median
[1] 5.6
```

```
> quantile(x, 0.25)  #1st Quartile
  25%
0.925

> quantile(x, c(0.25,0.75) )  #1st and 3rd Quartile
   25%    75%
 0.925 32.750


> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.100   0.925   5.600  28.070  32.750 101.00
```

factor objects

A `factor` is `vector` that handles categorical variables (e.g. sex, education level, …). It stores the observed values, the admissible values and the eventual labels. Notice that the order of the levels is important!

```
> sex <- factor(c("M","F","M","F","F"))
> sex
[1] M F M F F
Levels: F M
> sort(sex)
[1] F F F M M
Levels: F M

> sex <- factor(c("M","F","M","F","F"), levels=c("M","F"))
> sex
[1] M F M F F
Levels: M F
```

```
> sort(sex)
[1] M M F F F
Levels: M F

> sex <- factor(c("M","F","M","F","F"), levels=c("M","F"),
                labels=1:2)
> sex
[1] 1 2 1 2 2
Levels: 1 2
```

ordered factors

The `ordered factor` is the R structure appropriate to handle ordered categorical variables. Many functions (e.g. comparison operators or linear models treat `ordered` and `unordered` factors differently)

```
> education<- factor(c("phd","elementary","degree",
                       "high-school","bachelor","degree"))
> education
[1] phd  elementary  degree  high-school  bachelor  degree
Levels: bachelor degree elementary high-school phd


> education.ord <- ordered(education,
                     levels=c("elementary","high-school",
                              "bachelor","degree","phd"))
> education.ord
[1] phd  elementary  degree  high-school  bachelor  degree
Levels: elementary < high-school < bachelor < degree < phd

> education.ord[1] > education.ord[2]
[1] TRUE
```

```
> length(sex)
[1] 5

> mode(sex)
[1] "numeric"

> class(sex)
[1] "factor"

> levels(sex)
[1] "M" "F"

> nlevels(sex)
[1] 2
```

Remark.
By default NA (missing) is excluded from the levels of the factor:

```
> sex <- factor(c("M","F","M","F","F",NA),
levels=c("M","F",NA))

> sex
[1] M   F   M   F   F   <NA>
Levels: M F          # Levels are only M,F try to use table()

sex <- factor(c("M","F","M","F","F",NA),
levels=c("M","F",NA),exclude=NULL)

> sex
[1] M   F   M   F   F   <NA>
Levels: M F <NA>     # Now level NA is there too, try to use table()
```

A `factor` is the output of the function `cut()` used to categorize a continuous variable

```
cut(x, breaks, labels = NULL,
    include.lowest = FALSE, right = TRUE, ...)
```

default intervals are (argument `right = TRUE`): (a1,a2], (a2,a3], ...

(a1, a2] means a1 < x <= a2

```
> x <- 1:8
> fx <- cut(x, c(1,4,8))
> class(fx)
[1] "factor"

> fx
[1] <NA>  (1,4] (1,4] (1,4] (4,8] (4,8] (4,8] (4,8]
Levels: (1,4] (4,8]
```

```
> fx <- cut(x, c(1,4,8), include.lowest=T)
> fx
[1] [1,4] [1,4] [1,4] [1,4] (4,8] (4,8] (4,8] (4,8]
Levels: [1,4] (4,8]


> fx <- cut(x, c(1,4,8), include.lowest=T,
+               labels=c("1-4", "5-8") )
> fx
[1] 1-4 1-4 1-4 1-4 5-8 5-8 5-8 5-8
Levels: 1-4 5-8
```

The importance of a `factor` comes out in building contingency tables.

The `matrix` object

A `matrix` is a collection of elements, of the same type, organized in two dimensions, rows and columns.

The function `matrix()` is the common way to build a matrix. The following assignment defines a matrix of 0s, with 10 rows and 2 cols.

```
> x <- matrix(0, nrow=10, ncol=2 )
```

```
> mode(x)
[1] "numeric"

> class(x)
[1] "matrix"
```

```
> dim(x) #dimensions of the matrix
[1] 4 2

> length(x)  #no. of elements in x
[1] 8

> nrow(x)   # no. of rows
[1] 4

> ncol(x)   # no. of cols
[1] 2
```

```
> x <- matrix(1:8, 4, 2)
> x
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

NOTE the matrix is filled in by columns!!!

By using the argument `byrow=T`, the matrix is filled in by rows:

```
> x <- matrix(1:8, nrow=4, ncol=2, byrow=T)
> x
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
```

When the length of the `vector` is smaller than the length of the matrix, the matrix is filled by recycling the starting vector

```
>  x <- matrix(1:5, 4, 2)
Warning message:
data length [5] is not a sub-multiple or multiple of the
number of rows [4] in matrix
> x
     [,1] [,2]
[1,]    1    5
[2,]    2    1
[3,]    3    2
[4,]    4    3
```

In R the algebraic vector is a `matrix` object with only a column, that can be created by simply typing:

```
> X <- matrix(1:4)
> X
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4

> dim(X)
[1] 8 1

> class(X)
[1] "matrix"
```

The selection of the elements of a matrix can be done by referring to their position in rows and columns:

name_mat[*row_pos*, *col_pos*]

```
> x<- matrix(1:8, 4, 2)
> x
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8

> x[4,2]   #selects the element in 4th row and 2nd column
[1] 8
```

```
> x[4, ]  #all the elements in the 4th row
[1] 4 8

> x[ ,2]      #all the elements in the 2nd column
[1] 5 6 7 8

> x[c(1,4),]     #elements in the 1st and 4th row
     [,1] [,2]
[1,]    1    5
[2,]    4    8

> x[c(1,4),2]  #elements in 1st and 4th row of the 2nd col.
[1] 5 8
```

In the same way, by using the "-" it is possible to deselect a row/col:

```
> x[-2, ]    #drops the 2nd row
     [,1] [,2]
[1,]    1    5
[2,]    3    7
[3,]    4    8

> x[ ,-1]    #drops the 1st col
[1] 5 6 7 8
```

Operations with the **matrix** object

Adding, subtracting, multiplying or dividing a `matrix` by a constant is straightforward:

```
> x <- matrix(1:8,4)

> x+100
     [,1] [,2]
[1,]  101  105
[2,]  102  106
[3,]  103  107
[4,]  104  108
```

```
> x/10
     [,1] [,2]
[1,]  0.1  0.5
[2,]  0.2  0.6
[3,]  0.3  0.7
[4,]  0.4  0.8


> 1/x     #is NOT the matrix inversion
          [,1]      [,2]
[1,] 1.0000000 0.2000000
[2,] 0.5000000 0.1666667
[3,] 0.3333333 0.1428571
[4,] 0.2500000 0.1250000
```

In this case, a matrix of the same dimension of the original one is created; each element of the new matrix is the reciprocal of corresponding element in the origin matrix.

```
> x^2     #is NOT the matrix product xx
     [,1] [,2]
[1,]    1   25
[2,]    4   36
[3,]    9   49
[4,]   16   64
```

The same happens for the square root:

```
> sqrt(x)
         [,1]     [,2]
[1,] 1.000000 2.236068
[2,] 1.414214 2.449490
[3,] 1.732051 2.645751
[4,] 2.000000 2.828427
```

In operation involving two matrices (of the same dimension) the operation is performed element by element

```
> y <- x+100
> y
     [,1] [,2]
[1,]  101  105
[2,]  102  106
[3,]  103  107
[4,]  104  108

> x + y   #sum element by element
     [,1] [,2]
[1,]  102  110
[2,]  104  112
[3,]  106  114
[4,]  108  116
```

```
> y <- matrix(101:108, 2)
> y
     [,1] [,2] [,3] [,4]
[1,]  101  103  105  107
[2,]  102  104  106  108


> x+y
Error in x + y : non-conformable arrays


> t(x)+y
     [,1] [,2] [,3] [,4]
[1,]  102  105  108  111
[2,]  107  110  113  116
```

The function `t()` performs matrix transposition.

Care should be used in operations involving `matrices` and `vectors`

```
> y <- 1:4
> y
[1] 1 2 3 4

> x-y
     [,1] [,2]
[1,]    0    4
[2,]    0    4
[3,]    0    4
[4,]    0    4
```

The vector y is subtracted by each column of x.
Here again, R recycles the smaller object to obtain an object of the same length and dimension of the bigger one:

```
> x - matrix(y, nrow(x), ncol(x))
```

```
> y <- 1:5
> x-y
     [,1] [,2]
[1,]    0    0
[2,]    0    5
[3,]    0    5
[4,]    0    5
Warning message:
longer object length
      is not a multiple of shorter object length in: x - y
```

Matrix facilities

The operator to perform **matrix multiplication** is "%*%"

```
> x <- c(4, 7, 9, 2, 7, 1)
> dim(x) <- c(2,3)
> x
     [,1] [,2] [,3]
[1,]    4    9    7
[2,]    7    2    1

> y <- c(4, 1, 5, 7, 6, 8)
> dim(y) <- c(3,2)
> y
     [,1] [,2]
[1,]    4    7
[2,]    1    6
[3,]    5    8
```

```
> z <- x %*% y
> z
     [,1] [,2]
[1,]   60  138
[2,]   35   69
```

Remember: `z[i,j] = sum(x[i, ] * y[ ,j])`
for i=1,..,nrow(x); j=1,..,ncol(y) and ncol(x)=nrow(y).

Matrix inversion can be performed by using the `solve()` function.

```
> solve(z)
             [,1]         [,2]
[1,] -0.10000000  0.20000000
[2,]  0.05072464 -0.08695652
```

The function `diag()` gives different results:

a) when applied to an existing square matrix, it selects its main diagonal:

```
> z
     [,1] [,2]
[1,]   60  138
[2,]   35   69
> diag(z)
[1] 60 69
```

b) if its argument is a number k, a square identity matrix kxk is created:

```
> diag(2)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

c) if its argument is a vector, it returns a diagonal matrix with the elements of the vector as the main diagonal of the matrix

```
> diag(1:3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

Data matrix in statistics

The matrix dimension is *n* x *k*, *k* numeric variables observed on *n* units

```
colSums(x)      #sum of elements for each col of the matrix

colMeans(x)     #average for each column of the matrix

summary(x)      #summary stat. for each columns

var(x)          #Var-Cov. matrix

cor(x)          #correlation matrix
```

array object

An **array** is a collection of elements, all of the same type, organized in 2 or more dimensions.

`array()` is the main function to create it:

```
> x <- array(1:24, dim=c(4,3,2))
```

This assignment creates an array with numbers from 1 to 24, organized in 3 dimensions (length of the vector `dim(x)`: `length(dim(x))`).

```
> x
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
```

NOTE: the array is filled by column!!!!

```
> length(x)   #no. of elements of x
[1] 24

> mode(x)
[1] "numeric"

> class(x)
[1] "array"
```

The selection of elements of the array can be done by referring to their position in the dimensions:

name_array[*pos_dim1*, *pos_dim2*, …, *pos_dimN*]

Example:

```
> dim(x)
[1] 4 3 2
> x[1,2,1]   # 1st el. in dim 1, 2nd el. in dim 2, 1° el. in
dim 3
[1] 5

>x[1,2,]    # 1st el. in dim 1, 2nd el. in dim 2, all in dim 3
[1]  5 17
> x[1,,]    # 1st el. in dim 1, all in dim 2 and dim 3
     [,1] [,2]
[1,]    1   13
[2,]    5   17
[3,]    9   21
```

Dimension subscripts of arrays can be permutated through `aperm`. It can be useful whenever operations are required involving 2 or more arrays with the same dimensional structure.

Es.

```
> y <- aperm(x,c(2,1,3))
```

Changes array `x` with dimensions 4x3x2 into array `y` with dimensions 3x4x2:

```
> dim(y)
[1] 3 4 2
```

```
> y
, , 1

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

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   14   15   16
[2,]   17   18   19   20
[3,]   21   22   23   24
```

`list` object

A **`list`** is an ordered collection of others R data objects (vectors, matrices, factors, lists…).

Each single object is a component of the list. A list is created through the function `list()`

```
> fam <- list(name="Fred", wife="Mary", no.children=3,
+             child.ages=c(4,7,9))
```

This assignment creates a list called `fam`, consisting of 4 components.

```
> length(fam) #no. of components of the list
[1] 4
```

```
> mode(fam)
[1] "list"

> class(fam)
[1] "list"


> fam
$name
[1] "Fred"

$wife
[1] "Mary"

$no.children
[1] 3

$child.ages
[1] 4 7 9
```

The first component of the list is a character vector with only one element, "Fred", and it can be referred as `fam[[1]]`

```
> fam[[1]]
[1] "Fred"

> fam[[2]]
[1] "Mary"

> fam[[3]]
[1] 3

> fam[[4]]
[1] 4 7 9
```

In this case we have assigned a name to each component:

```
> names(fam)
[1] "name"         "wife"         "no.children" "child.ages"
```

These names can be used in selecting the components by using a structure:

name_list$*name_component*

```
> fam$wife
[1] "Mary"

> fam$no.children
[1] 3

> fam$no    #if unique, only part of the name can be used
[1] 3
```

When selecting the components of a list by referring to their position it is important to type the right number of square parenthesis:

```
> x <- fam[[4]]
> x
[1] 4 7 9
> mode(x)
[1] "numeric"

> x <- fam[4]
> x
$child.ages
[1] 4 7 9

> mode(x)
[1] "list"
> length(x)
[1] 1
```

Therefore structure of the type:

name_list[*pos_component*]

selects a "sub-list" from the starting list.

This is useful when selecting two or more components from a list:

```
> fam[1:2]    # selects the 1st and the 2nd component
$name
[1] "Fred"

$wife
[1] "Mary"
```

```
> fam[c(2,4)]  # selects the 2nd and the 4th component
$wife
[1] "Mary"

$child.ages
[1] 4 7 9

> fam[-2]  #drops the 2nd component
$name
[1] "Fred"

$no.children
[1] 3

$child.ages
[1] 4 7 9
```

Appending a new component to the list is easy:

```
>fam$child.sex<-factor(c("M","F","F"), levels=c("M","F"))
```

A new component named "child.sex" is appended to the list.

```
> length(fam)
[1] 5
> fam[[5]]
[1] M F F
Levels: M F
```

In alternative we can refer to the position of the new component:

```
> fam[[6]] <- c(0,1,1)
> length(fam)
[1] 6
```

```
> fam[[6]]
[1] 0 1 1
> names(fam)
[1] "name"         "wife"         "no.children" "child.ages"
"child.sex"    ""
```

Two lists can be concatenated by using the function `c()`:

```
> fam1 <- fam[1:2]
> mode(fam1); length(fam1)
[1] "list"
[1] 2

> fam2 <- fam[3:5]
> mode(fam2); length(fam2)
[1] "list"
[1] 3

> FAM <- c(fam1,fam2)

> mode(FAM); length(FAM)
[1] "list"
[1] 5
```

The function `unlist()` transforms a `list` into a `vector`

```
> x <- unlist(FAM)
> x
       name        wife no.children child.ages1
     "Fred"      "Mary"         "3"          "4"
child.ages2 child.ages3  child.sex1  child.sex2
        "7"         "9"         "1"          "2"
 child.sex3
        "2"

> mode(x)
[1] "character"
> length(x)
[1] 9
```

NOTE: Due to the presence of a character vector, all the elements of the other components of the list are coerced to character.

data.frame object

A **`data.frame`** is a particular `list`: it is a `list` in which each component of the list is a `vector`/`factor` of the same length. Each component of the `list` is the result of the observation of a variable on the statistical units.

Due to this feature, the `data.frame` is organized and printed as a `matrix`: each column contains the observed values for the variable on the statistical units.

A `data.frame` admits at the same time `numeric`, `logical` or `factor` components (columns).

A `data.frame` is created by means of the function `data.frame()`

```
> id <- c("001","002","003","004")
> sex <- factor(c("M","F","F","F"), levels=c("M","F"))
> age <- c(33, 25, 61, 21)

> x <- data.frame(id,sex,age)
> x
   id sex age
1 001   M  33
2 002   F  25
3 003   F  61
4 004   F  21
```

```
> length(x)   #return the no. of components
[1] 3

> mode(x)
[1] "list"

> class(x)
[1] "data.frame"
```

Note that, by default, `character` vectors are coerced to `factor`,

```
> class(id)
[1] "character"
> class(x$id)
[1] "factor"
```

Unless the argument `stringsAsFactors` is set to "FALSE"

```
> x <- data.frame(id,sex,age, stringsAsFactors=FALSE)

> class(x$id)
[1] "character"
```

Given that a `data.frame` is `list` organized as a `matrix`, we can treat it as a `list` or as a `matrix`:

```
> length(x)   #return the no. of components
[1] 3

> nrow(x)
[1] 4
> ncol(x)
[1] 3
> dim(x)
[1] 4 3
```

The same happens when we want to select some elements of the data.frame:

```
> x$age  #component named "age"
[1] 33 25 61 21

> x[,"age"]  #column with name "age"
[1] 33 25 61 21

> x[,3]  #3rd column
[1] 33 25 61 21

> x[2,]  #vales observed on the 2nd unit
   id sex age
2 002   F  25

> class(x[2,])  #NOTE it is still a data.frame
[1] "data.frame"
```

Adding a new column (component) to the `data.frame`

```
> q1 <- as.logical(c(1,0,1,1))
> q1
[1]  TRUE FALSE  TRUE  TRUE
> x$q1 <- q1

> x
   id sex age    q1
1 001   M  33  TRUE
2 002   F  25 FALSE
3 003   F  61  TRUE
4 004   F  21  TRUE
```

Adding a new row (unit) to the `data.frame`

Remember that each row of the `data.frame` is a `data.frame` itself.

```
> new.unit <- data.frame(id="005",sex="M",age=60,q1=FALSE)

> xx <- rbind(x, new.unit)  #row binding
> xx
   id sex age    q1
1 001   M  33  TRUE
2 002   F  25 FALSE
3 003   F  61  TRUE
4 004   F  21  TRUE
5 005   M  60 FALSE
```

A `data.frame` can be transformed in a `matrix` (usually a matrix with numeric elements) thought the function `data.matrix()`

```
> y <- data.matrix(xx)
> y
  id sex age q1
1  1   1  33  1
2  2   2  25  0
3  3   2  61  1
4  4   2  21  1
5  5   1  60  0

> class(y)
[1] "matrix"

> mode(y)
[1] "numeric"
```

Subsetting objects by logicals

which(): given an object, finds the indices of the object's elements satisfying a logical condition

Vectors

```
> z <- c(1,2,5,12,32,27,14)

> which(z %% 2 == 0)
[1] 2 4 5 7

> z[which(z %% 2 == 0)]
[1]  2 12 32 14
```

When the input object has an array structure (i.e. has a dim attribute) one can access the array indices of the elements:

```
> m <- matrix(sample(1:20,20,rep=TRUE),4,5)
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    6    5   15   19   19
[2,]   14   17   13   16    3
[3,]   11    9    5    3   10
[4,]   20   19    2   20    8

> which(m > 18, arr.ind=TRUE)
     row col
[1,]   4   1
[2,]   4   2
[3,]   1   4
[4,]   4   4
[5,]   1   5
```