R Laboratory
Introduction to R

Francesco Schirripa
francesco.schirripa@ec.unipi.it

March 14-15, 2019

What is R?

- "*R is a free software environment for statistical computing and graphics*" (`https://www.r-project.org/about.html`).
- R is based on the computer language S, which was developed by John Chambers and others at Bell Laboratories in 1976.
- R was initially designed in 1992 by Ross Ihaka and Robert Gentleman at the Department of Statistics of the University of Auckland in Auckland, New Zealand. However, many individuals has contributed to R by sending code and bug reports.
- It is open source and available at `https://www.r-project.org/`.
- It is *multiplatform* (Windows, Linux, MacOsX).

Why R?

- R is open source
  - Open source means that you do not have to pay for it...but it is much more: it provides full access to algorithms and their implementation; it gives you the ability to fix bugs and extend software; it promotes reproducible research . . .
- "*Created by statisticians for statisticians*".
- R is Flexible and powerful. R can handle complex and large data.
- R is Well Supported. R has a huge, active and constantly evolving users-based community (*community-supported*).
- R has amazing graphical capabilities.
- . . .

How does R work?

Once R is started, a console is displayed where commands can be written (at the prompt >).

```
> 3+5
[1] 8
> 3*5
[1] 15
> 5/5
[1] 1
> 6-2
[1] 4
```

However, it is a good practice to store all commands in a text (*script*) file with extension .R.

Two windows:

- Script (new or existing)
- Terminal – output and temporary input (unsaved)

R Studio

- RStudio (`http://www.rstudio.com`): integrated development environment for R. It includes in just one screen:
  1. Text editor with syntax highlighting, brackets matching (very useful) and buttons/keyboard combinations to submit code snippets to the console directly (i.e., no need for copy-and-paste)
  2. Console (output);
  3. Workspace browser, a data viewer and the commands history;
  4. Windows for graphics/packages/online help

References on R

- *Official* manuals on `https://cran.r-project.org/`
- Courses on Data Camp (`https://www.datacamp.com/`)
- *Use R!* series published by Springer, in which you can find specialized texts on variety of topics. On the Springer website (`https://www.springer.com/series/6991?detailsPage=titles`) you can find a list of the titles in this series. Some of these texts area available in the library (`https://www.sba.unipi.it/`)
- The funny guide *The R Inferno*, by Patrick Burns, available here: `http://www.burns-stat.com/pages/Tutor/R_inferno.pdf`
- The web is full of resources on R: blogs, websites, forums ... (such as `https://stackoverflow.com/`)

Getting help

The online help is a very useful tool to familiarise with R and its commands:

1. `help.start()` opens the html main page of R online help;
2. `help(cmd1)` (or `?cmd1`) provides details about how command cmd1 works; if you are interested in the help for operators you have to put it between apostrophes `help('%*%')`
3. `help.search("keyword")` (or `??keyword`) performs an online search based on keyword;
4. `apropos("keyword")` returns all the commands containing keyword in its name.

Getting help: R code

```
> help.start()

> help(sqrt)
> ?sqrt

> help.search("linear␣models")
> ??linearmodels

> apropos("mean")
```

The working directory: organize your work

- Once you open R you are inside the memory of the computer.
  The part of the memory in which you are currently working is called **working directory**.
  You can find out which is the current working directory by running the `getwd()` - get working directory - function.
- `dir()` displays all the files in the directory;
- `setwd("new path")`: change the current working directory.
  In RStudio you can set the working directory using the menu.
- There is no general rule on how to organize your working directories but may be convenient create a single directory for each project (for instance named "ProjectName") and create some sub-directories for data, script, documents . . .

R objects

R is an **object-oriented program**: every operation is made on and produces objects. All objects in R have a class, reported by the function `class()`. Possible class are: `numeric`, `logical`, `character`, `list`, `matrix`, `array`, `factor` and `data.frame`.
Assignment is performed by the <- or the = operator.

```
> x<-5
> x
[1] 5
> class(x)
[1] "numeric"
```

During an R session, objects are created and stored by name.
The R command `ls()` can be used to display the names of the objects which are currently stored within R.
The collection of objects currently stored is called the workspace.
`rm(list=ls())` cleans the whole workspace;

R commands and case sensitivity

R is **case sensitive**:

⇒ **A and a are different symbols** and would refer to different variables.

Commands are separated either by a **semi-colon (';')**, or by a **newline**

Comments can be put almost anywhere, starting with a hashmark ('#'), everything to the end of the line is a comment.

If a command is not complete at the end of a line, R will give the symbol '+'.

```
# This is a comment not a line-command
> a<-2
> A
Error: object 'A' not found  # error message

# I can write two commands on the same line
> A<-4; A
[1] 4
```

Rules for assignment

Names defining objects cannot contain spaces or mathematical operators/special characters (except for the dot .), nor can they begin with a number; Some peculiar values:

- `NA` (Not Available) is the code denoting a missing numerical or character element (warning: "NA" is a valid character string);
- `NaN` (Not a Number) is the result of impossible or undefined expressions like a division by zero;
- `Inf` and `-Inf` denote $\pm\infty$
- These and other R keywords (`for,while,if,TRUE,FALSE` etc.) are not available for assignment.

Vectors

- A vector is a sequence of data elements of the same basic type. To define a vector we use the function `c()`;
- `length()`: returns the length of a vector;
- `str()`: returns the internal structure of the vector (or of an R object in general);
- `x=seq(from=a,to=b,by=s)`: returns a vector with elements from `a` to `b` with step `s`;
- `x=rep(x,times=a)`: repeats `x` `a` times (`x` can be a vector);
- Specific elements can be selected using square brackets: `x[a]`;

Vectors (cont.)

```
> x<-c(2, 3, 5)
> #vector containing three numeric values 2, 3 and 5
> x
[1] 2 3 5
> str(x)
num [1:3] 2 3 5
> length(x)
[1] 3
>  x[2] #select a specific element
[1] 3
> # A vector can contain character strings.
> b=c("aa", "bb")
> class(b)
[1] "character"
> #A vector of 10 elements from 1 to 10
> x<-seq(1,10,1)
```

Vectors (cont.)

Basic mathematical functions can also be applied to vectors. Such functions are performed element-by-element, i.e. *elementwise*

```
> # Define two vectors
> a = c(1, 3, 5, 7)
> b = c(1, 2, 4, 8)

> #if we multiply a by 5, we would get a vector with
> # each of its members multiplied by 5.
> 5 * a
[1]  5 15 25 35

> # if we add a and b together, the sum would be a
> # vector whose members are the sum of the
> # corresponding members  from 'a' and 'b'.
> a + b
[1]  2  5  9 15
```

Vectors (cont.)

In other cases - like for basic statistical summary measures - the whole vector is the input of the function

```
> s1 <- c (6, 1, 5, 9, 4, 7, 8, 2, 5, 8)
> mean(s1) #mean
[1] 5.5
> median(s1) # median
[1] 5.5
> range(s1) # range (min and max)
[1] 1 9
> var(s1)   # variance
[1] 6.944444
> sd(s1)    #standard deviation
[1] 2.635231
> summary(s1) #provide summary stats about s1
Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
1.00    4.25    5.50    5.50    7.75    9.00
```

Logical values & operators

- Some operators return the logical values TRUE and FALSE:
  1. `<`: less than; `<=`: less or equal than;
  2. `>`: greater than; `>=`: greater or equal than;
  3. `==`: equal to; `!=`: different from;
  4. `is.element()`: set membership indicator;
  5. `is.na()` indicates the elements of the vectors that represent missing data
- Logical operators:
  1. `&`: logical intersection (AND);
  2. `|`: logical union (OR);
  3. `!`: logical negation
- `which()` returns the indices of elements satisfying a logical condition.

Logical values & operators: R code

```
> a<-c(6,2,5,3,8,2)
> b<-c(1,4,3,6,7,2)
> a<b
[1] FALSE  TRUE FALSE  TRUE FALSE FALSE
> a<=b
[1] FALSE  TRUE FALSE  TRUE FALSE  TRUE
> a!=b   # means "different than"
[1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
> is.element(2,a)
[1] TRUE
> is.element(9,a)
[1] FALSE
> > which (a<5)
[1] 2 4 6
```

Missing values

In R, missing values are represented by the symbol 'NA' (not available)

```
> a<-c(6,2,5,3,8,2)
> is.na(a)
> # returns TRUE if an element of a is missing
[1] FALSE FALSE FALSE FALSE FALSE FALSE
> a[3]=NA
> #assign a missing value at the third element of a
> is.na(a)
[1] FALSE FALSE  TRUE FALSE FALSE FALSE
```

May be useful recode a particular values as NA, missing values are represented by the symbol 'NA' (not available)

```
> # Re-coding the occurrence of a particular value
>  x = c(3.14, 98, 0, 99, 7, NA, 0, 99)
>  x[x==99] = NA # Re-code all 99 in x as NA
> # Re-coding the occurrence of NA
>  x[is.na(x)] = -1 # Recode all NA in x as -1.
```

Factors & lists

- **Factors**:
  1. are objects encoding categorical variables;
  2. are often used to group other (usually quantitative) variables;
  3. are defined by the command `factor()`;
  4. have a reference level which can be changed;
- **Lists**:
  1. are set of objects with different nature/dimension;
  2. are defined by the command `list()`;
  3. are useful to summarise an analysis or as output of complex functions.

Factors & lists: R code

```
> #define a numerical vector status
> status<-c(rep(0,5), rep(1,8), rep(2,3))
> status_factor<-factor(status, labels = c("divorced",
+                                 "married", "single"))
> table(status_factor)# gives the frequency table
status_factor
divorced  married   single
       5        8        3
> n = c(2, 3, 5); s = c("aa", "bb", "cc", "dd", "ee")
> b = c(TRUE, FALSE, TRUE, FALSE, FALSE)
> mylist = list(n, s, b, 3)
> str(mylist)
List of 4
$ : num [1:3] 2 3 5
$ : chr [1:5] "aa" "bb" "cc" "dd" ...
$ : logi [1:5] TRUE FALSE TRUE FALSE FALSE
$ : num 3
```

Matrices & Arrays

- A matrix is a collection of data elements arranged in a two-dimensional rectangular layout:
  `matrix(data,nrow,ncol,byrow = FALSE)` defines a `nrow`×`ncol` matrix; by default matrix is filled by columns (we can fill the matrix by row using the argument `byrow` = TRUE)
- Arrays are the R data objects which can store data in more than two dimensions:
  `array(data,dim=c(n1,...,np))` defines a p-dimensional array;
- *elementwise* operations like for vectors can be performed;
- Most common **linear algebra operators** readily available:
  1. `%*%`: matrix product (conformable matrix dimensions needed);
  2. `det()`: matrix determinant;
  2. `t()`: matrix/vector transposition;
  2. `solve()`: solution of a linear system (can be used to invert matrices);
  2. `diag()`: extracts the diagonal of a matrix or builds a diagonal matrix.

Define a matrix and indexing: R code

```
> A = matrix( c(1, 2, 3, 8, 8, 7), nrow=2, ncol=3)
> A  # print the matrix
     [,1] [,2] [,3]
[1,]    1    3    8
[2,]    2    8    7
> #INDEXING
> A[2, 3]  # element at 2nd row, 3rd column
[1] 7
> A[2, ]   # the 2nd row
[1] 2 8 7
> A[ ,3]   # the 3rd column
[1] 8 7
> A[ ,c(1,3)]  # the 1st and 3rd columns
[,1] [,2]
[1,]    1    8
[2,]    2    7
```

Define a matrix and indexing: R code

```
> A = matrix( c(1, 2, 3, 8, 8, 7), nrow=2, ncol=3)
> A  # print the matrix
[,1] [,2] [,3]
[1,]    1    3    8
[2,]    2    8    7
> #INDEXING
> A[2, 3]  # element at 2nd row, 3rd column
[1] 7
> A[2, ]   # the 2nd row
[1] 2 8 7
> A[ ,3]   # the 3rd column
[1] 8 7
> A[ ,c(1,3)]  # the 1st and 3rd columns
[,1] [,2]
[1,]    1    8
[2,]    2    7
```

Combine R Objects by Rows or Columns

Another way of creating a matrix is by using functions `cbind()` and `rbind()` as in column bind and row bind.
`cbind()` and `rbind()` take a sequence of vector, matrix or data frames arguments and combine by columns or rows, respectively

```
> cbind(c(1,2,3),c(4,5,6))
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> rbind(c(1,2,3),c(4,5,6))
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Dataframes

- Data frame is a two dimensional data structure in R. It is a special case of a list which has each component of equal length.
- Data frame is of particular importance in data analysis. It represents the matrix of data where **each row is an observation** and **each column is a variable** (the variables may be of different type) Dataframes are defined by command: `data.frame(data,nrow,ncol)`;
- Variable `X` of dataframe `df` is obtained by `df$X`;
- `data()`: returns the list of all R datasets;
- `names(df)` or `colnames(df)`: returns the names of the columns of `df`;
- `attach(df)`: allows to refer to variables in `df` without using `df$X`;
- `detach(df)`: stops the effect of `attach(df)`.

Dataframes: R code

```
> height = c(200, 165, 150)
> patent= factor(c(1,1,2),labels=c("no", "yes"))
> gender = c("male", "female", "female")
> # DEFINE THE DATAFRAME
> df = data.frame(height, patent, gender)
> # ADDING COLUMNS AND ROWS TO A DATAFRAME
> df2<- data.frame(birth_place=c("Pisa",
+                  "Florence", "Pisa"))
> df_tot<-cbind(df,df2)
> df_tot
  height patent gender birth_place
1    200     no   male        Pisa
2    165     no female    Florence
3    150    yes female        Pisa
```

Dataframes: R code (cont.)

```
> # Add a row to my dataframe using rbind()
> # Note: Column names and the number of columns
> # of the two dataframes needs to be same.
> df3<-data.frame(height = c(180, 182, 170),
+                 patent= factor(c(2,1,2),
+                 labels=c("no", "yes")),
+                 gender = c("female", "female",
+                 "female"),
+                 birth_place=c("Pisa","Pisa",
+                 "Florence"))
>
> df_tot2<-rbind(df_tot,df3)
> head(df_tot2,3)
  height patent gender birth_place
1    200     no   male        Pisa
2    165     no female    Florence
3    150    yes female        Pisa
```

Dataframes: R code (cont.)

```
> mean(iris$Petal.Length)
[1] 3.758
> attach(iris)
> mean(Petal.Length)
[1] 3.758
> # ordering dataframe by a given variable
> iris_new<-iris[order(iris$Petal.Length),]
> #selecting a subset of observations
> new_iris=subset(iris,
+                 Petal.Length>quantile(Petal.Length,
+                                       probs=0.25))
> #MERGE 2 DATAFRAMES
> df1<-data.frame(ID.1=1:10, y=21:30)
> df2<-data.frame(ID.2=1:10, z=61:70)
> df.merge<-merge(df1, df2, by.x="ID.1", by.y="ID.2")
> # by.x and by.y specifications of the columns used
> # for merging
```

Importing and exporting data

- The main commands to read external data are:
    1. `read.table()` for .txt files;
    2. `read.csv()` and `read.csv2()` for .csv files;
- Most common argument of these functions are (be careful because the default values of some of these argument in the two functions are different!)
    1. `file`: name with extension (and eventually pattern) of the data file;
    2. `header`: whether the first row contains the column names;
    3. `sep`: the column separator (space, comma, tabulation, semi-colon);
    4. `dec`: the decimal separator (dot or comma).
- Data can be exported with the commands: `write.table()`, `write.csv()`, `write.csv2()`

Importing and exporting data: R code

```
> #SET WD
> #setwd("your WD")
> # load .txt file
> mydata <- read.table("occupation.txt", header=TRUE,
+                       sep="" , dec=".")
> # if I do not specify the working directory I have
> # to specify the path of the data
> # load .csv file
> mycsv<-read.csv("example1.csv", sep=";" ,dec=",",
+                 header=TRUE)
```

R packages (1)

- All R functions and datasets are stored in packages (also called libraries). Only when a package is loaded are its contents available.
- A package is a collection of previously programmed functions
- Packages not included in the default R release must be downloaded from the CRAN with `install.packages(pkg1)`. R and RStudio have user-friendly interface to load/install packages.
- When a package is installed on your pc you have to load it in the current R session with or `library(pkg1)` (or `require(pkg1)`): in this way all the functions and datasets in the package are available for usage.
- `library()` lists the available packages in the current R session.

R packages (2)

- Information on a package and its functions can be gathered by `help(package=pkg1)` or by `??pkg1`.
- On the webpage of a package you can find the Reference manual in PDF format which may provide additional useful information.
- Moreover, many packages have vignettes. A vignette is a long-form guide to the package. It is like a book chapter or an academic paper: it describe the problem that the package is designed to solve, and then show the reader how to solve it.

Basic programming with R: Loops in R

- Loops are used in programming to repeat a specific block of code.
- On the whole, we can divide loops in two categories
  1. loops executed for a given number of times, as controlled by a counter or an index, incremented at each iteration cycle. These are part of the `for` loop family.
  2. loops based on the verification of a logical condition. The condition is tested at the start or the end of the loop construct. These variants belong to the `while` or `repeat` family of loops, respectively.

For loops

`for` loops can be used to loop through the values of an object. The command is of the form

```
        for (<index> in <vector>) {
        <statements>
        }
```

The expressions between curly brackets are executed separately for each value in the vector

```
> x = c(1,5,7,10)
> for (i in (1:length(x)))
+ {
+   x[i]=x[i] + i
+ }
> x
[1]  2  7 10 14
```

Apply-family functions

For loops are very slow. You can improve code readability over more efficient solutions such as the *apply functions
There are several related functions in R which allow you to apply a function to a series of objects (vectors, matrices, dataframes or lists).
They include: `apply`, `lapply`, `sapply`, `tapply`, `aggregate`.
For example I want to compute the mean of each column of a matrix

```
> x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
> apply(x, 2, mean)
x1 x2
3  3
```

I want the mean of Petal Length by Species (iris dataset).

```
> tapply(iris$Petal.Length,iris$Species,mean)
setosa versicolor  virginica
1.462      4.260      5.552
```

The if else statement

Sometimes, you want to execute a function only if a certain condition is met. In this case, the if structure can be used, and complemented by the else control.

```
if(boolean_expression) {
statement(s) will execute if the boolean exp is true.
} else {
statement(s) will execute if the boolean exp is false.
}
```

Alternatively you can use the `ifelse()` function:

```
ifelse(boolean_expression,
       value if condition is true,
       value if condition is false)
```

The if else statement (cont.)

```
> x = c(4:1, 2:5)
> for (i in 1:length(x))
+ {
+ if (x[i]>2)
+ {
+   print("x␣is␣larger␣than␣2")
+ } else
+ {
+   print("x␣is␣less␣than␣or␣equal␣2")
+ }
+ }
[1] "x␣is␣larger␣than␣2"
[1] "x␣is␣larger␣than␣2"
[1] "x␣is␣less␣than␣or␣equal␣2"
[1] "x␣is␣less␣than␣or␣equal␣2"
[1]  .....
```