

R-Bootcamp (day 1)

Dr. Matteo Tanadini & Co.

7-10 September 2020

Outline

- 1 Admin and what is **R**
- 2 **R** Basics
- 3 **R** Basics: Fundamental objects
- 4 **R** Basics: Functions
- 5 Importing data
- 6 Some **R**-functions to inspect data
- 7 Course assignments

Section 1

Admin and what is **R**

- Sarah Grimm
- Bernd Fellinghauer (Data Fittery, Oftrigen)
- Manuel Koller (Albourne Partners, London)
- Claude Renaux (ETHZ, Zurich)
- David Schwarz (Inventx, Zurich)
- Lorenzo Tanadini (BFH, Zollikofen)
- Matteo Tanadini (Zurich Data Scientists, Zurich)

Day structure

- 09:00 - 12:00 course (with 30 min break)
- lunch break¹
- 13:00 - 16:00 course (with 30 min break)
- 16:00 - 17:00 assignment / exercises

¹Breaks and lunch are "movable".

Teaching approach

- many medium-small "elements" (i.e. topics)
- from 10 min to 1 hour
- slides - R-code - exercises (15-15-15)
- do ask questions!

What is **R**?

- THE statistical software
- frequently used in Data Science (besides Python)
- collaborative project (academia)
- open source
- free
- large community that contributes to it (add-on packages)
- extremely rich and fast evolving

Steps of a data analysis

Question: What are the main steps of a data analysis?

Steps of a data analysis

Question: What are the main steps of a data analysis?

- 1 import data
- 2 prepare data (modify, merge, ...)
- 3 graphical analysis
- 4 fit models
- 5 check and interpret results (output and graphs)

Section 2

R Basics

First steps

Let's open **R** and run a simple computation:

- 1 write a statement into **R**
- 2 hit the "Return" key

```
> 7 + 5
```

```
[1] 12
```

- prompt >
- results [1]

Simple arithmetics

- + - * / ^
- calculation priorities
- incomplete statements can be continued on the next line(s)
- interrupting a statement ("Esc" key)

Assignments (1)

The results of a statement can be stored in an object:

```
a.1 <- 3 * (17 - 10)
```

Typing the name of an object displays it in the **R**-console:

```
a.1
```

```
[1] 21
```

Typical **R**-analyses are composed of very many steps, where the results of computations are stored into objects.

```
a.2 <- a.1^2
```

👉 Go to Democode 👈

Assignments (2)

When creating objects note that:

- **R** is case sensitive (i.e. `a.1` \neq `A.1`)
- objects can be overwritten (cannot be undone, is permanent)
- object names must...
 - ▶ start with a letter
 - ▶ can contain dots or underscores (e.g. `a.1`, `a_1`)
 - ▶ cannot contain special characters (e.g. `#`, `*`, `&`, ...)
 - ▶ more comes later

R-scripts

- statistical analyses are composed by many steps, where intermediate results are stored, inspected and commented.
- the sequence of instructions used to carry out a statistical analysis can be collected in a **script**.
- to run **R**-commands, statements must be copied into the **R**-console.
- comments can be added to **R**-scripts by using the special character **#**.
- any statement following **#** is ignored by **R**.

```
a.1 + 1
```

```
[1] 22
```

```
## This is a comment, and thus ignore by R
```

👉 Go to Series 1 exercise 1 👉

Editors (1)

Editors enable us to:

- write **R**-code into a script and send it to the console.
- make the code more readable via colouring.
- to automatically complete our code ("Tab" key).
- interact with many other software (more comes later, e.g. Rmarkdown).
- ...

Editors (2)

There are many editors freely available

- Rstudio
- Emacs Speaks Statistics (ESS)
- tinn-R
- ...

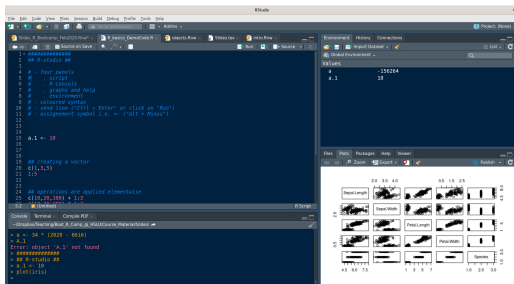
Currently, the most popular choice is Rstudio

- user friendly
- very capable and powerful
- developed by a private company
- free (other non-free products also available)
- no relationship with **R** (often misunderstood)

Rstudio (2)

4 panels:

- script
- **R**-console
- environment
- help / graphs / packages



Section 3

R Basics: Fundamental objects

Vectors (1)

One of the most important objects in **R** are **vectors**. A vector is a one-dimensional sequence of elements of a given type (e.g. numbers).

- numeric vectors
- character (or string) vectors
- logical vectors

Vectors can be created with the `c()` function ("`c`" stands for combine).

Vectors (2)

- numeric vectors that are sequences can be created with ":"
(e.g. 2:5)
- elements of a string vector are defined with quotation marks
(e.g. "Anna")
- elements of a string vector can contain empty spaces
(e.g. "Anna Meier")
- logical vectors can be created with logical operators (see next slide)

Logical operators

- larger than: `>` (e.g. `20 > 13` returns `TRUE`)
- smaller than: `<`
- equal: `==`
- not equal: `!=`
- larger equal: `>=`
- smaller equal: `<=`

Vectors (3)

Operations are applied element-wise:

```
c(10,10,10) + c(1,2,3)
```

```
[1] 11 12 13
```

When the length of the vectors involved does not match [recycling](#) is used.

```
c(10,20,30) + 1:6
```

```
[1] 11 22 33 14 25 36
```


Vectors (4)

Accessing elements of a vector can be done via indexing, which uses square brackets.

```
v.age <- c(15, 17, 34, 6, 101)
v.age[2]

[1] 17
```

Several elements can be accessed simultaneously.

```
v.age[c(2,3,5)]

[1] 17 34 101
```

Vectors (5)

Elements of a vector can be accessed with a logical vector.

```
v.age
```

```
[1] 15 17 34 6 101
```

```
v.age[c(FALSE,TRUE,TRUE,TRUE,FALSE)]
```

```
[1] 17 34 6
```

Vectors (6)

An existing vector can be used to access elements of another vector

```
v.age[c(2,3,5)]
```

```
[1] 17 34 101
```

```
## same as
```

```
v.ind <- c(2,3,5)
```

```
v.age[v.ind]
```

```
[1] 17 34 101
```

```
## same as
```

```
v.ind.logical <- c(FALSE,TRUE,TRUE,FALSE,TRUE)
```

```
v.age[v.ind.logical]
```

```
[1] 17 34 101
```

Vectors (7)

Negative indexes can be used to drop elements of a vector.

```
v.age  
[1] 15 17 34 6 101
```

```
v.age[2:5]
```



```
[1] 17 34 6 101
```

```
## same as
```

```
v.age[-1]
```

```
[1] 17 34 6 101
```

 [Go to Democode](#) 

 [Go to Series 1 exercise 2](#) 

Matrices (1)

A `matrix` is a two-dimensional object that contains elements of a given type (e.g. strings).

- numeric matrices
- character (or string) matrices
- logical matrices

Matrices can be created with the `matrix()` function. Alternatively, the functions `rbind()` or `cbind()` can also be used.

Matrices (2)

Operations also apply element-wise to matrices

```
Mat_1 <- matrix(data = 0:5, ncol = 3)
```

```
Mat_1
```

	[,1]	[,2]	[,3]
[1,]	0	2	4
[2,]	1	3	5

```
Mat_2 <- matrix(data = 10:15, ncol = 3)
```

```
Mat_2
```

	[,1]	[,2]	[,3]
[1,]	10	12	14
[2,]	11	13	15

```
Mat_1 + Mat_2
```

	[,1]	[,2]	[,3]
[1,]	10	14	18
[2,]	12	16	20

Matrices (3)

Accessing elements of a matrix is also done via indexing. Two dimensions must be specified.

Matrix.name[row(s), column(s)]

```
## element in first row and second column
```

```
Mat_1
```

```
      [,1] [,2] [,3]  
[1,]    0    2    4  
[2,]    1    3    5
```

```
Mat_1[1,2]
```

```
[1] 2
```

Matrices (4)

By omitting one dimension we get the entire row, respectively the entire column.

```
## elements in second row
```

```
Mat_1
```

```
      [,1] [,2] [,3]  
[1,]    0    2    4  
[2,]    1    3    5
```

```
Mat_1[2, ]
```

```
[1] 1 3 5
```

```
##
```

```
## elements in second and third column
```

```
Mat_1[ , 2:3]
```

```
      [,1] [,2]  
[1,]    2    4  
[2,]    3    5
```

Note that even though only one dimension is specified, the "comma" must be retained.

Take Home Messages: Vectors and Matrices

- both vectors and matrices are used to store data
- vectors are one-dimensional objects
- matrices are 2-dimensional objects²
- all the elements of a given vector must be of the same type
- all the elements of a given matrix must be of the same type
- the class of a vector/matrix can be numeric, character, logical, ... (other types come later)

👉 Go to Democode 👉

👉 Go to Series 1 exercise 3 👉



²A generalisation to N-dimensions exist, see `?array()`.

Data frames (1)

- **R** is a software to analyse data
- data often comes in 2-dimensional "tables" (e.g. excel sheets)

Different data types possible, in matrix not possible

Sepal.Length	Petal.Width	Species
5.1	0.2	setosa
4.9	0.2	setosa
7.0	1.4	versicolor
6.4	1.5	versicolor
6.3	2.5	virginica
5.8	1.9	virginica
7.1	2.1	virginica

within one column only one type of data

Question: Can we use a matrix to store this dataset?

Data frames (2)

- data frames can contain columns of different type³
- data frames are the most frequently used object to store data in **R**
- data frames can be created via the `data.frame()` function OR
- data frames can be created by importing data into **R** from external files (e.g. an excel file)

³Note, however, that all the elements of a given column **MUST** belong to the same type.

Data frames (3)

Accessing elements of a data frame:

- can be done via **square brackets** (as for vectors and matrices)
- can be done via the "\$" symbol (note "partial matching")
- **negative indexes** can be used to **drop elements**

Data frames (4)

Modifying existing data frames

- columns can be added via the "\$" symbol or the squared brackets
- rows and columns can be added with the `rbind()` and `cbind()` functions (less common)
- data frames can be combined via the `merge()` function⁴

⁴Merging data frames is a slightly more advanced topic not covered here.

Take Home Messages: Data Frames

- data frames are **2-dimensional objects**
- all elements of a given column in a data frame belong to the same class (e.g. numeric, character,...)
- however, different columns of a data frame can belong to **different classes**
- data frames can be created via the `data.frame()` function
- Most of the time data frames are created by **importing data** into **R** (more comes soon)

👉 Go to Democode 👉

👉 Go to Series 1 exercise 4 👉



Lists (1)

Data frames are very flexible as they can store data of different types in one object. However, they not are able to deal with vectors of differing length.

Lists are more flexible objects that **can store objects of different classes and different dimensions.**

```
l.1 <- list(A = "a",  
            num.vec = 10:5,  
            Mat_1)
```

```
l.1
```

```
$A  
[1] "a"
```

```
$num.vec  
[1] 10 9 8 7 6 5
```

```
[[3]]  
      [,1] [,2] [,3]  
[1,]    0    2    4  
[2,]    1    3    5
```

Lists (2)

- all elements of a list be accessed using the double squared brackets
- named elements of a list be accessed via the "\$" symbol

```
## accessing list elements
```

```
l.1[[2]]
```

```
[1] 10  9  8  7  6  5
```

```
## same as
```

```
l.1[["num.vec"]]
```

```
[1] 10  9  8  7  6  5
```

```
## same as
```

```
l.1$num.vec
```

```
[1] 10  9  8  7  6  5
```


Lists (3)

- a list can contain another list
- many objects created by fitting model are lists (e.g. LMs)

```
l.2 <- list(daysNames = c("monday", "tuesday"),  
            Other_info = list(9:5, B = c(TRUE, FALSE, FALSE)))
```

```
##
```

```
lm.iris <- lm(Sepal.Length ~ Petal.Length, data = iris)  
str(lm.iris, max.level = 0)
```

```
List of 12
```

```
- attr(*, "class")= chr "lm"
```

```
is.list(lm.iris)
```

```
[1] TRUE
```

- list elements are also called slots (e.g. `lm.iris` has 12 slots)

Take Home Messages: Basic **R** objects

- vectors, matrices, data frames and lists are fundamental **R** objects
- data sets are often stored into data frames
- data frames are composed by vectors
- lists are "complex" objects (e.g. models)

Section 4

R Basics: Functions

Functions (1)

- **R** comes with hundreds of predefined functions
- a function call can result in:
 - ▶ an output being created (e.g. `matrix()`)
 - ▶ a graph being produced (e.g. `plot()`)
 - ▶ a file being created (e.g. `write.csv()`)
 - ▶ an object being deleted (e.g. `rm()`)
 - ▶ ...

Functions (2)

- functions are called with
"Function.Name(argument1 = value1, argument2 = value2, ...)"
- functions have arguments (which are named)

```
matrix(data = 12:1, nrow = 3, ncol = 4)
```

Functions (3)

- functions have arguments
 - ▶ compulsory arguments
 - ▶ optional arguments
- compulsory arguments must be defined to make the function work

```
mean()
```

```
Error in mean.default(): argument "x" is missing, with no default
```

```
mean(x = 1:12)
```

```
[1] 6.5
```

`x` is a compulsory argument of the `mean()` function. Indeed, some data must be provided to be able to compute a mean.

Some functions do not have compulsory arguments, but only optional ones (e.g. `ls()`)

Functions (4)

- each **R** function has an [help page](#) (e.g. type `?mean`)
- help pages tell us
 - ▶ what the function does
 - ▶ what its arguments are
 - ▶ ...
 - ▶ and shows us a few examples (at the very bottom)
- Rstudio allows us to search for a given topic within an help page (see "Find in Topic")

Functions (4)

- arguments can have **default values**
- default values are specified in the help page (e.g. `?matrix`)

Question: look at the help page for the `matrix()` function and try to predict what the output of the following command will be.

```
matrix()
```


Functions (4)

- arguments can have **default values**
- default values are specified in the help page (e.g. `?matrix`)

Question: look at the help page for the `matrix()` function and try to predict what the output of the following command will be.

```
matrix()
```

```
      [,1]  
[1,]    NA
```

Functions (5)

Arguments position and name

- as long as the **correct ordering** is maintained, argument **names can be dropped**
- for example in `matrix()`
 - 1 "data"
 - 2 "nrow"
 - 3 "ncol"
 - 4 "byrow"
 - 5 "dimnames"

```
matrix(data = 1:12, nrow = 2, ncol = 6)
## same as
matrix(1:12, 2, 6)
```

Note: "ncol" does not really need to be specified as data has length 12 and there are two rows. Therefore, the matrix will have 6 columns.

Functions (6)

Arguments position and name

- as long as the correct ordering is maintained, argument names can be dropped
- but don't forget unused arguments

```
matrix(data = 1:12, nrow = 2, byrow = TRUE) ## ncol not specified  
## same as  
matrix(1:12, 2, , TRUE) ## 3rd argument is left empty
```

Functions (7)

Arguments position and name

- as long as arguments names are provided, their ordering can be changed

```
matrix(data = 1:12, nrow = 2, byrow = TRUE)
## same as
matrix(nrow = 2, byrow = TRUE, data = 1:12)
```

Functions (8)

Arguments position and name

- remember that **R** is case sensitive
 - ▶ `c()` is not the same function as `C()`
 - ▶ the argument "Data" does not exist in `matrix()` ("data" does)
- **partial matching**: argument names must not be written entirely as long as they are unambiguous

```
seq(from = 1, to = 10, length.out = 5)
## same as
seq(from = 1, to = 10, leng = 5)
## possible as no other argument name starts with "leng"
```

Functions (9)

Arguments position and name: [Good practices](#)

- the first argument name is often dropped (as it is obvious)
- do use names for all other arguments
- avoid "partial matching"
- leave an empty space between argument names and values (readability)
- for long function calls, you can write over several lines (readability)

```
matrix(1:16, ncol = 4, nrow = 4,  
       byrow = TRUE)
```

Take Home Messages: Functions

- **R** comes with very many prebuilt functions
- **R** functions perform very many tasks
- thousands of additional packages with additional functions are freely available (more on packages comes later)
- functions have useful help pages (most of the time)
- functions have arguments (optional and compulsory)
- good practices on how to call a function exists and should be used

👉 Go to Democode 👉

👉 Go to Series 1 exercise 5 AND 6 👉



Section 5

Importing data

Importing data (1)

- virtually all analyses imply that some data is imported
- **R** allows you to read almost any kind of data
- for example: .csv, .xls, .txt, ... files
- Two main options to import data
 - ▶ via **R** commands (e.g. `read.table()`)
 - ▶ via the Rstudio interface⁵

⁵In the background Rstudio also uses functions such as `read.table()`. It all boils down to the same.

Importing data (2)

- in base **R** there are many `read.xxx()` functions
- **Question:** How would you proceed to identify them?
- many more functions to import data are present in add-on packages

Importing data (3)

read.table() example:

```
d.blueEggs <- read.table(file = "ExampleDataSets_ForSlides/BlaueEier.txt",  
                        header = TRUE)  
head(d.blueEggs)
```

	Jahr	Teilnehmer
1	2004	42
2	2005	65
3	2006	98
4	2007	147
5	2008	99
6	2009	182

```
str(d.blueEggs)
```

```
'data.frame': 16 obs. of 2 variables:
```

```
$ Jahr      : int  2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 ...  
$ Teilnehmer: int  42 65 98 147 99 182 223 327 234 252 ...
```

Importing data (4)

`read.table()` example:

- "file": the path to a file or a valid url
 - ▶ one can provide an [absolute path](#)
 - ▶ or a path that [relative](#) to the current [working directory](#)
 - ▶ or a [url](#) address
- "header": in most files, the very first row contains the names of the columns

Working directory

R is working (or pointing) to a specific directory on your machine.

```
getwd()

[1] "/home/tana/Dropbox/Teaching/Boot_R_Camp_@_HSLU/Course_Material/Slides/Day_1"

## This will be different on your machine
```

- if you start Rstudio (and hence **R**) by opening an existing **R**-script. The working directory will be the folder containing the **R**-script
- if you start Rstudio (resp. **R**) directly, then the working directory will be as specified in your defaults settings (e.g. in the "Documents" folder).
- the working directory can be changed with the `setwd()` function (or via the Rstudio options).

Importing data (5)

`read.table()` example:

- "skip": number of lines to be skipped beginning to read data
- "nrows": maximum number of rows to read in
- "fileEncoding": encoding of the file (e.g. "UTF-8")
- ...
- "sep": is the field separator character (almost outdated)
- "dec": the character used in the file for decimal points (almost outdated)

Most of the function to import data have the same or similar arguments (argument names may differ).

Importing data (6)

Via Rstudio example:

Choose "File" → "Import Dataset" → "From ..."

Rstudio:

- automatically chooses the "best" import function to use (AI)
- uses the file name as object name
- tries to guess whether there are column names or not
- prints the corresponding R code in the console !!!
- opens the dataset in a new tab (via View())

Take Home messages: Importing data

- datasets can be imported into **R** from:
 - ▶ an existing file
 - ▶ an internet connection (url)
 - ▶ a SQL server (not discussed here)
- there are very many functions to import datasets⁶
- we can read data "directly" (via **R**-commands) **OR**
- via an editor (e.g. Rstudio)
- the lazy, but often simplest, solution is to use Rstudio once and then copy and paste the code

👉 Go to Democode 👉

👉 Go to Series 1 exercise 7 👉



⁶This is a direct consequence of the fact that there are very many different files types that contain data.

Section 6

Some **R**-functions to inspect data

Inspecting data

After having **imported** a dataset it is **essential to inspect the object** to make sure that the import process went well.

Common issues encountered when importing data are:

- data was not imported properly
- column names were taken as being observations
- the kind of data was not properly recognised
- ...

Inspecting data

Extremely useful functions to inspect data:

- `str()`
- `head()`
- `summary`
- `table()`

We must always check data after importing it!

👉 Go to Democode 👉

Section 7

Course assignments

Course assignments

In order to collect the credits for this course there are 3 conditions to satisfy

- presence on all 4 days (physical or online)
- create a list functions (small work)
- produce the data analysis as described in
"Assignment_Rbootcamp.pdf" (larger and most important work)