Rbasics

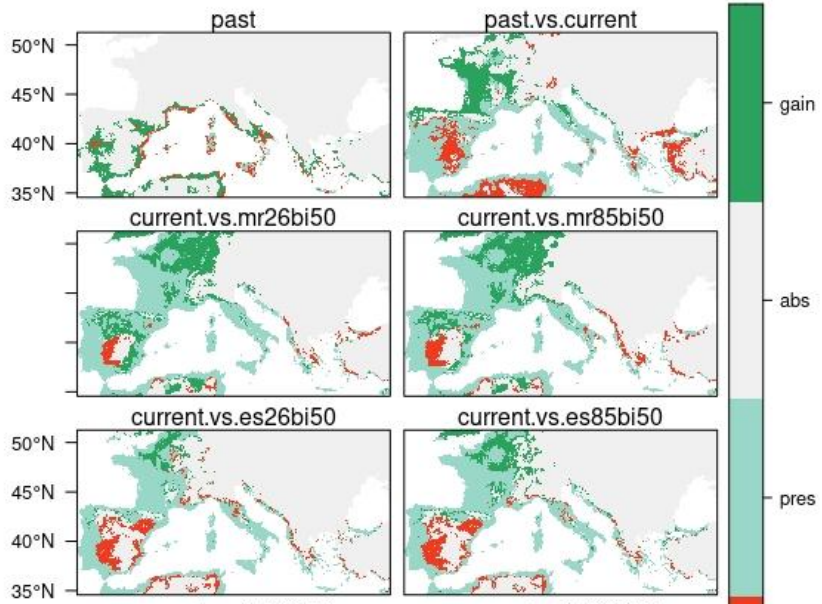
PhD toolbox - 40th PhD cycle

Dr. Matteo Chialva && Dr. Martino Adamo

teachers for a month

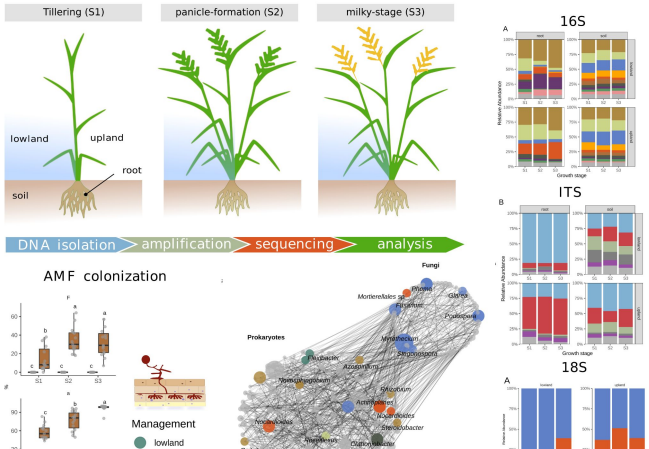
Dr. Martino ADAMO

He is researcher since 2023, I work on plants diversity conservation and orchid mycorrhizas using bio-molecular tools and boring statistical model to study traits influence on species spatial distribution.



Dr. Matteo CHIALVA

He is researcher since 2023 and his research focuses on plant-microbes interactions in model crop species by using multi-omics approaches from transcriptomics to metagenomics. By coupling these tools with systems biology and biostatistics he is interested in linking soil microbiota diversity and functioning to plant responses and ecosystem services.



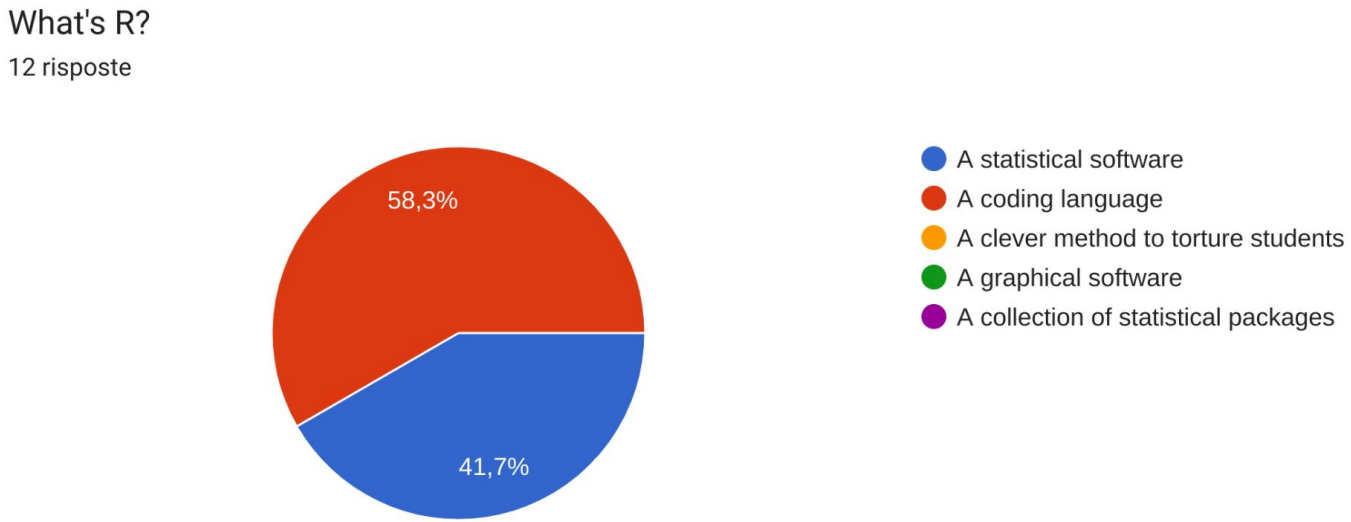
students forever

PhD is the first step of the academic career … thus, if you want to continue … you are going to study every single day of your working life!

We asked you to explain your study field in three keywords and this is results:

Describe your reseach field in 3 keywords
12 risposte
Impact invasive species
Operation, Logistics & Production
Amendments, microbial communities, soils physical characterization
Primates; Behaviour; Sociality
Microbe, Plant, Interaction
Genetic diversity, NGS technologies, Bioinformatics
Cell biology
Fungi Biotechnology pollutant
acoustic behaviour cetaceans
Micropaleontology, Paleocenography, Paleoenvironment
Ecology - Stream - Algae
Molecular, sequencing, cancer

What’s R?



R *per se* is a programming environment for statistics and graphics, not a simple standalone software.
Much more similar to a coding language rather than a statistical software

Why R?

1. R is **free** and **open-source**

2. With R you can do almost everything.

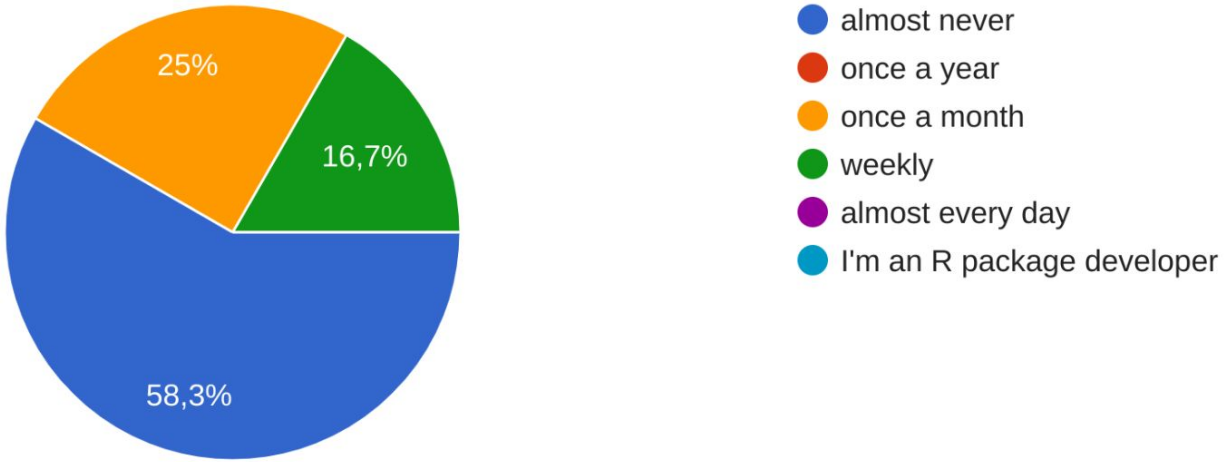
3. R can address many of the challenges of performing **reproducible research**

4. Learning R will make you a more attractive candidate

"When" R?

How frequently do you use R?

12 risposte

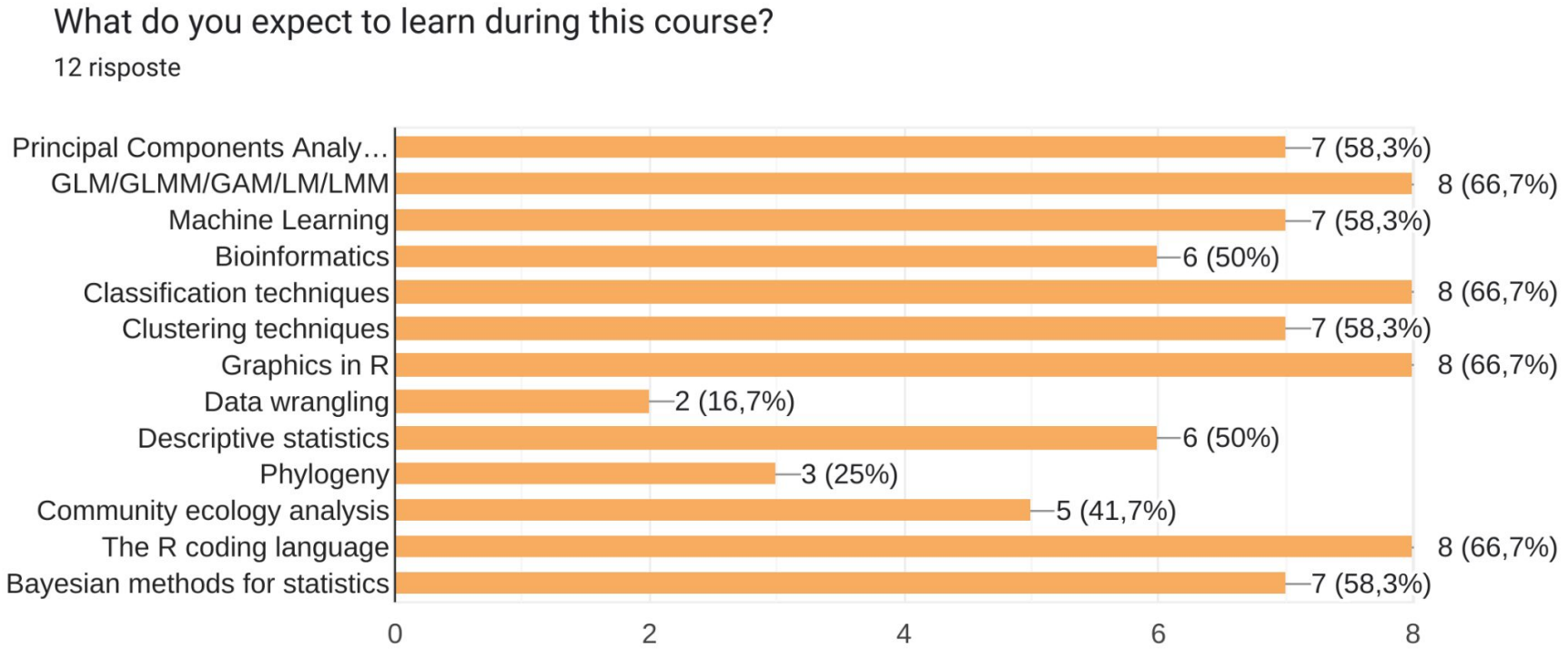


"When" R?

Why you should use it **every day**:

- open source and cross platform
- active community of developers: *i.e.* access to the state-of-the-art methods
- extensive documentation everywhere (books, online, forums etc..)
- great graphics capabilities: you can manage all the type of plots you can imagine (and mainly those that you could not!)

What are you going to learn?



What are you going to learn?

With R you can do analyses in all the scientific disciplines.

BUT here you'll learn R and its main functions to deal with tables and graphics.

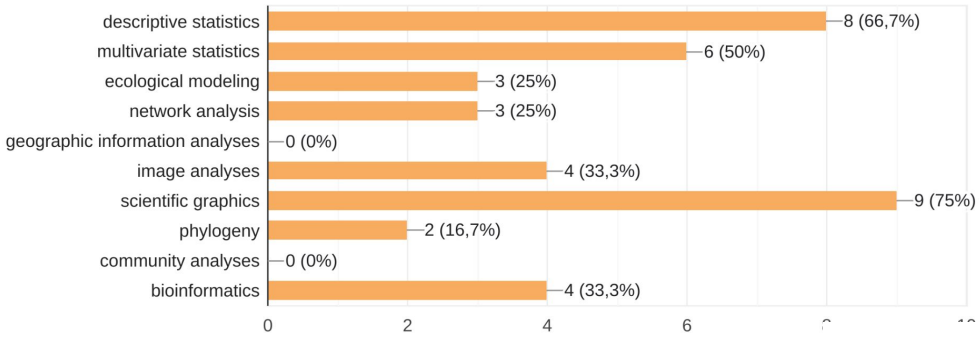
!! You'll not learn statistics !!

"*Our aim is to provide you universal tools to deal with your data in a reproducible way*"

What are you going to learn?

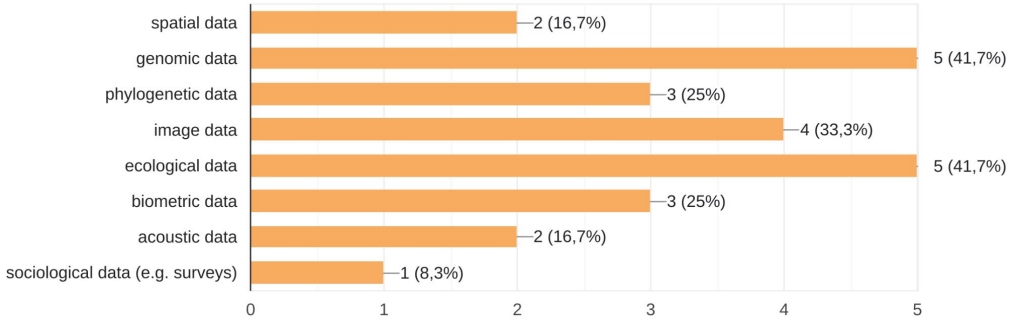
Which kind of data analyses do you usually implement or you plan to deal with?

12 risposte



Which kind of data do you usually analyze or are you planning to deal with?

12 risposte



Reproducible research

1. Define a question
2. Gather information and resources (observe)
3. Form an explanatory hypothesis
4. Test the hypothesis by performing an experiment and collecting data in a reproducible manner
5. Analyze the data
6. Interpret the data and draw conclusions that serve as a starting point for a new hypothesis
7. Publish results (and possibly the raw data and the code you used to analyze them)
8. Retest (frequently done by other scientists)

What is reproducibility?

"*The possibility to reproduce someone else results. It's part of the scientific method (or it should be...)*"

How can you ensure reproducibility? One of the best way is to use **CODE**

- it is universal
- it can be easily shared and documented
- it avoid manual data manipulation steps (prone to introduce errors; time consuming; impossible to reproduce)

If you cannot avoid manual data manipulation you must document what has been done and why.

Let’s start with practice!

installing R and RStudio

Almost everyone run R through RStudio GUI (graphic user interface) which is a software which helps you to write in R.

1. Install R - https://cran.r-project.org/
2. Install RStudio (free version) - https://posit.co/downloads/

Note: in both cases installation procedure depends on your operating system.

updating R and RStudio

Periodically you should have to update both R and/or RStudio.

Update R - different options available

The most effective way is to completely remove the previous R version and install the new one.

Note: you have not to fully reinstall all the packages: move the old packages in the new directory.

Update RStudio

Download and install the new version (removal of the previous version is highly suggested!)

using R through RStudio: the main features

- coding
- packages management
- output visualization
- environment monitoring
- execution/running
- paths exploration

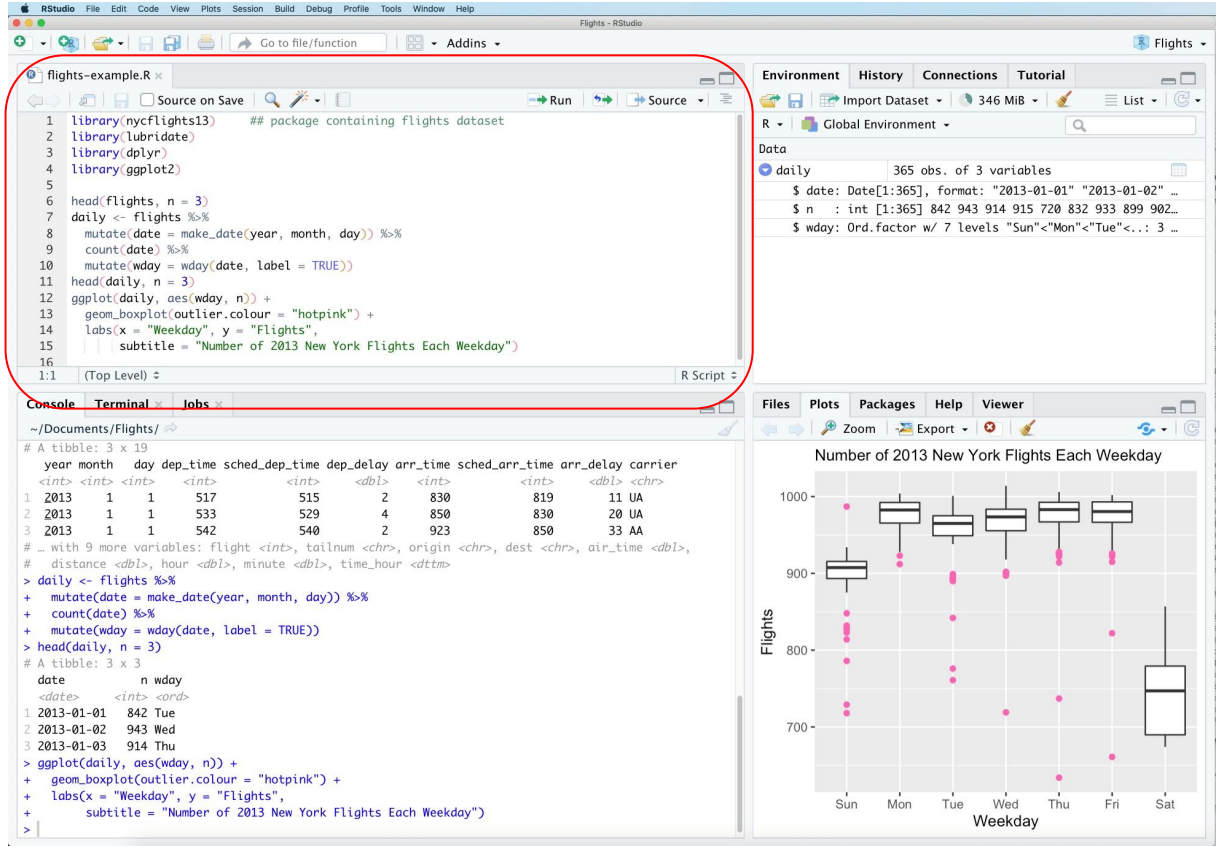
RStudio is an integrated development environment for R

using R through RStudio: the main features

- **coding**

  write here your code script

- packages mgmt
- output visualization
- environment monitoring
- execution/running
- paths exploration



using R through RStudio: the main features

- coding
- packages mgmt
- **output visualization**

  this is actually R!

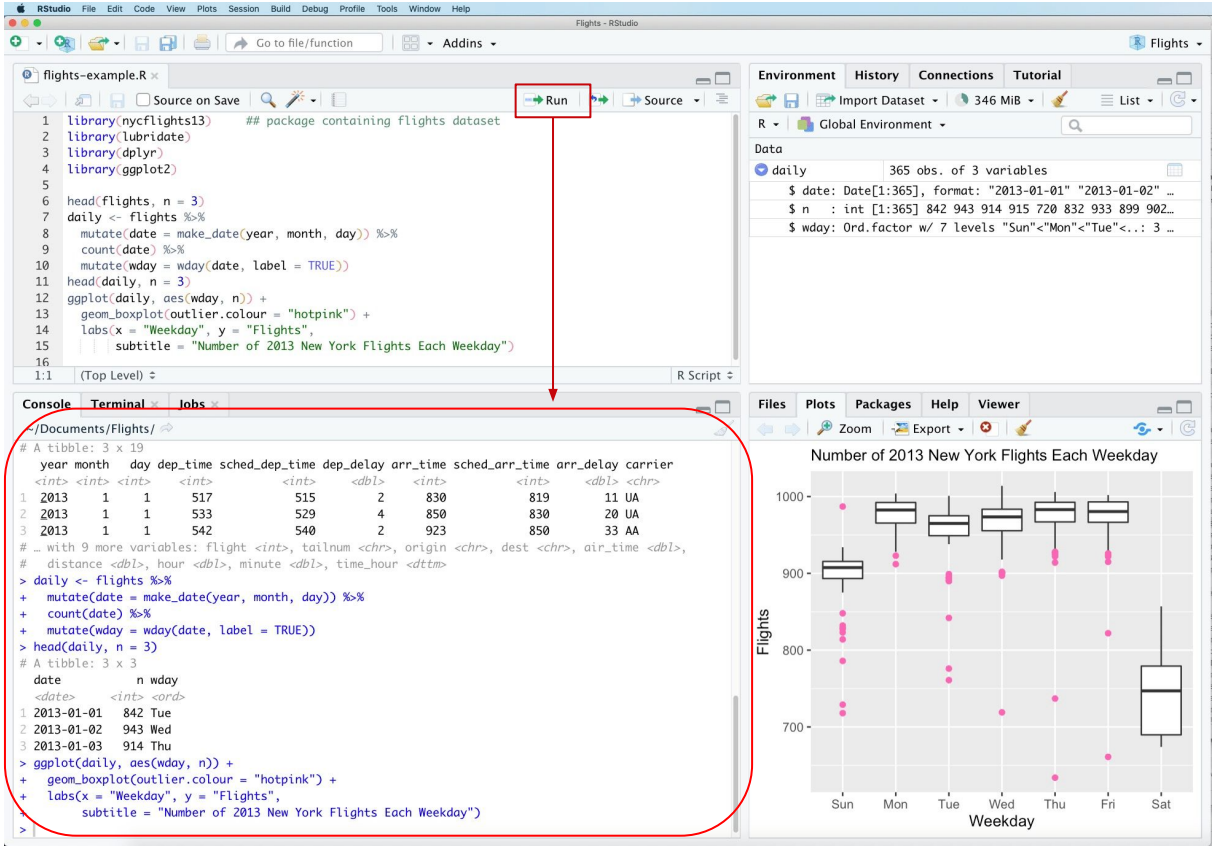
  results visualization

- environment monitoring
- **execution/running**

  this is actually R!

  code running

- paths exploration

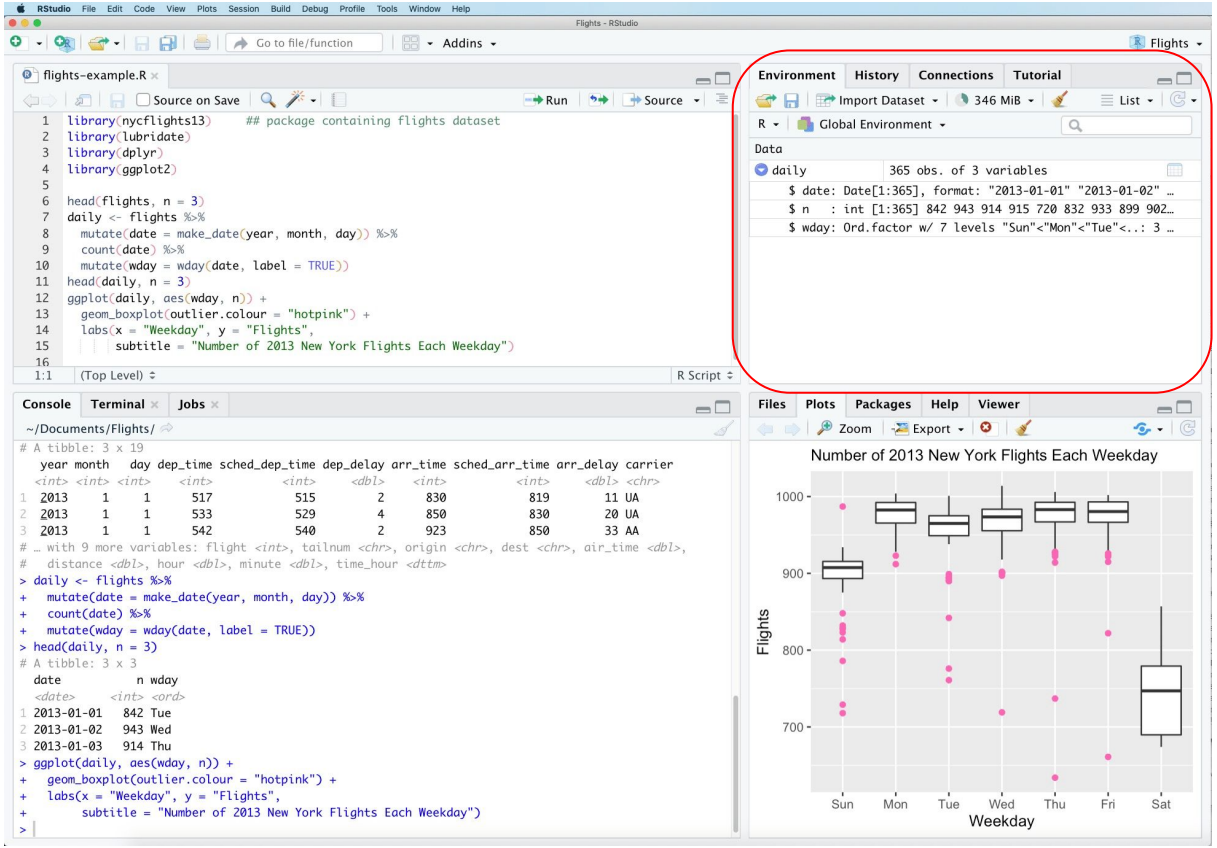


using R through RStudio: the main features

- coding
- packages mgmt
- output visualization
- **environment monitoring**

  the best idea of Rstudio, you can see all the **OBJECTS** you loaded/created during the session

- execution/running
- paths exploration



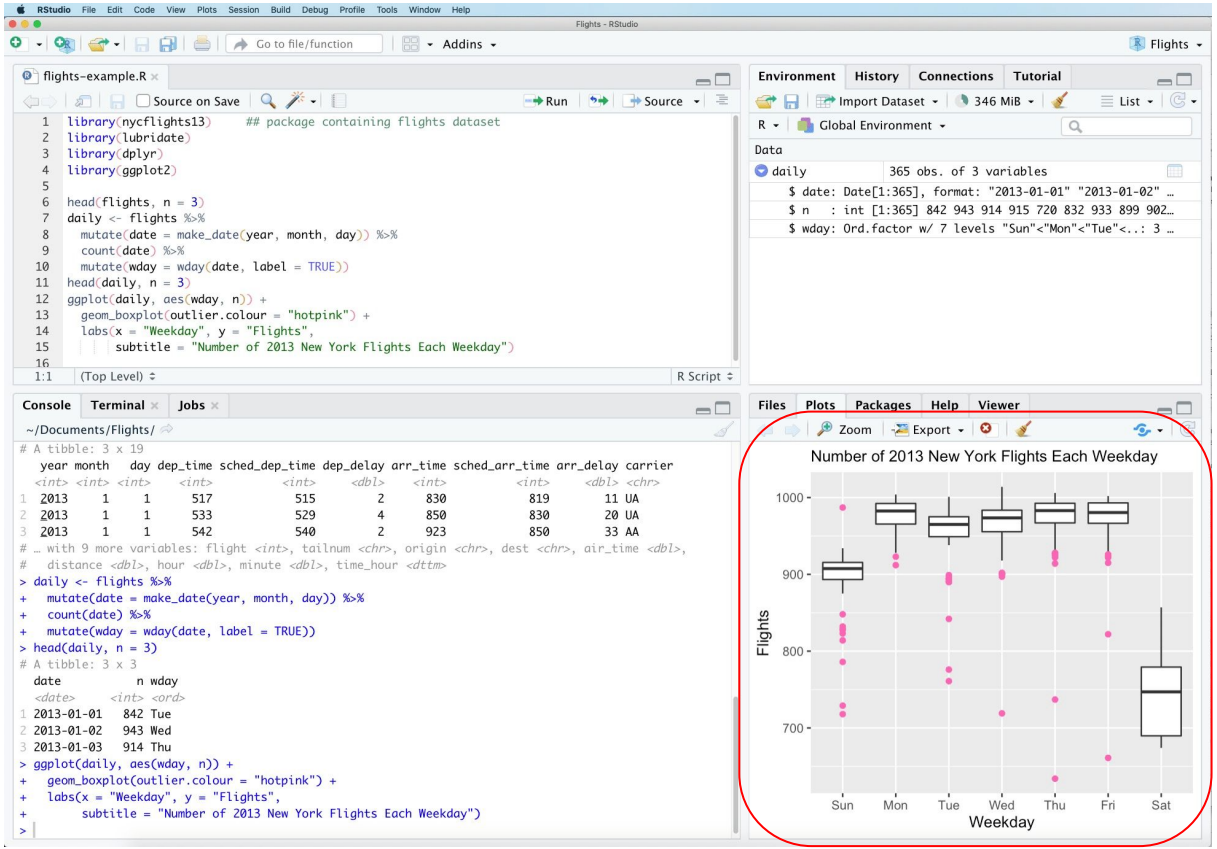
using R through RStudio: the main features

- coding
- **packages management**

  lookup help pages and your packages
- **output visualization**

  here you see graphical outputs
- environment monitoring
- execution/running
- **paths exploration**

  explore your folders/paths



Few notes on programming

- **R is** the most **simple** and intuitive coding language available (functions have simple names)
- You don’t need to be an **experience** programmer to use R language
- as every language, R has its own **grammar**
- Let **RStudio help** you in writing in R (as Word helps you writing in another language)
- Don’t panic, but … you’ll need to **memorize** a lot of functions
- most of R functions are **simple and intuitive**, the best way is to practice!
- coding is **case sensitive** (e.g. object and Object are not the same thing!)
- "**#**" symbol in general excludes lines that the language runs. It is used to comment-out lines with alternative/inactive code or to annotate your script.
- **text** strings are in **quotes**: e.g. "Object" is the string text name of Object
- objects names cannot contain spaces or mathematical operators
- possibly try to keep objects and functions name different!
- objects are created, replaced, filled with the symbol **"<-"**
- object <- "I’m an object"
- "<-" and "=" are alternatives

Let’s start with coding!

data type

numeric (num) = numbers

```
num [1:91] 1 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 ..
```

factor (factor) = categories, ordinal numbers, boolean

```
Factor w/ 2 levels "due","uno": 2 1
```

character (char) = text

```
chr [1:8] "A" "A" "B" "B" "A" "A" "B" "B"
```

integer (int) = integer numbers

```
int [1:10, 1:5] 41 18 25 12 13 23 42 10 21 2 …
```

logical (logi) = TRUE/FALSE values (boolean data type = factor special case)

```
logi [1:5] TRUE FALSE FALSE FALSE TRUE
```

Boolean are all two-step variables:

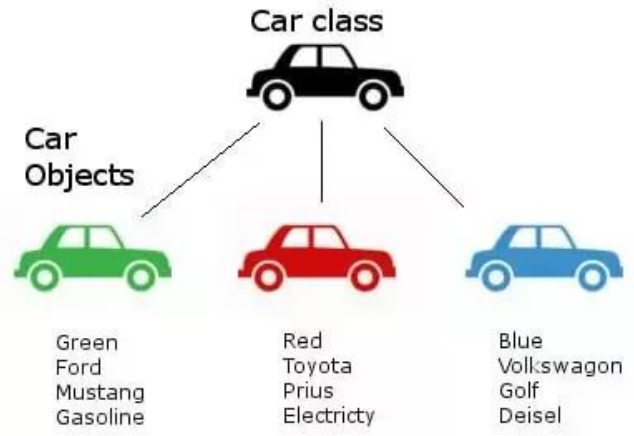
True-False
Yes-No
1-0
dog-cat

object classes

R is a **object-oriented** programming (OOP) language

object types are:

1. atomic vectors
2. matrices
3. dataframes
4. lists
5. functions
6. special features



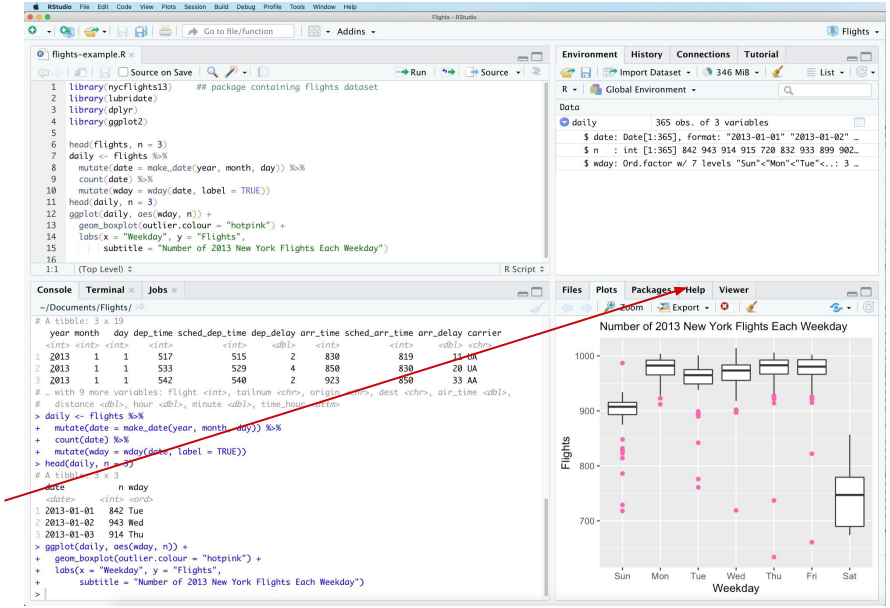
R object class :: functions

`>install.packages()` is a R function.

R function are **ALWAYS** followed by brackets

R functions are grouped in packages.

each function is more or less explained in a **help page**:

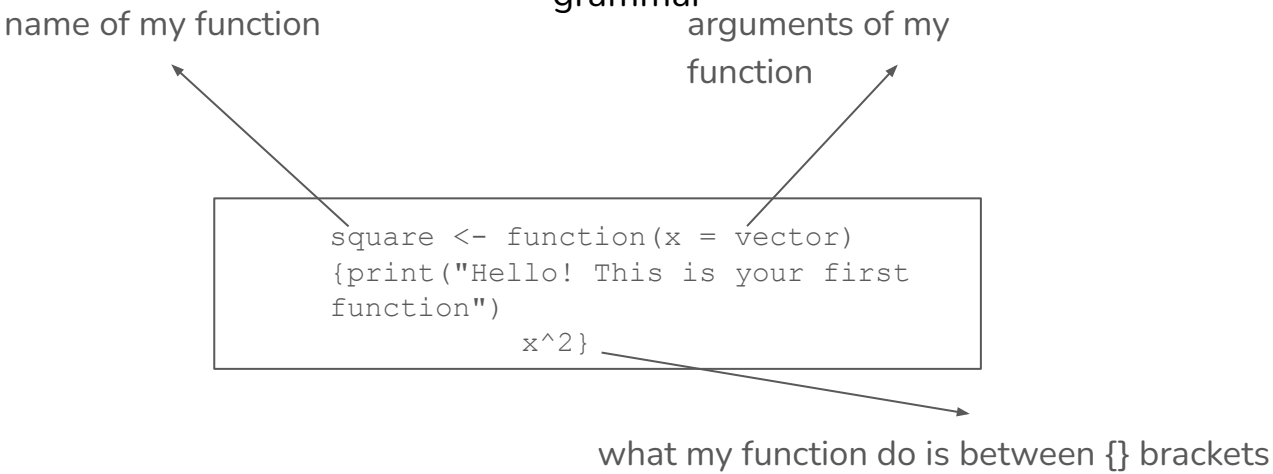


Usage of a simple function:

```
>factor(x = character(), levels, labels = levels,
     exclude = NA, ordered = is.ordered(x), nmax = NA)
```

R object class :: functions

Everybody in R can write functions. Just a naive example to understand the grammar

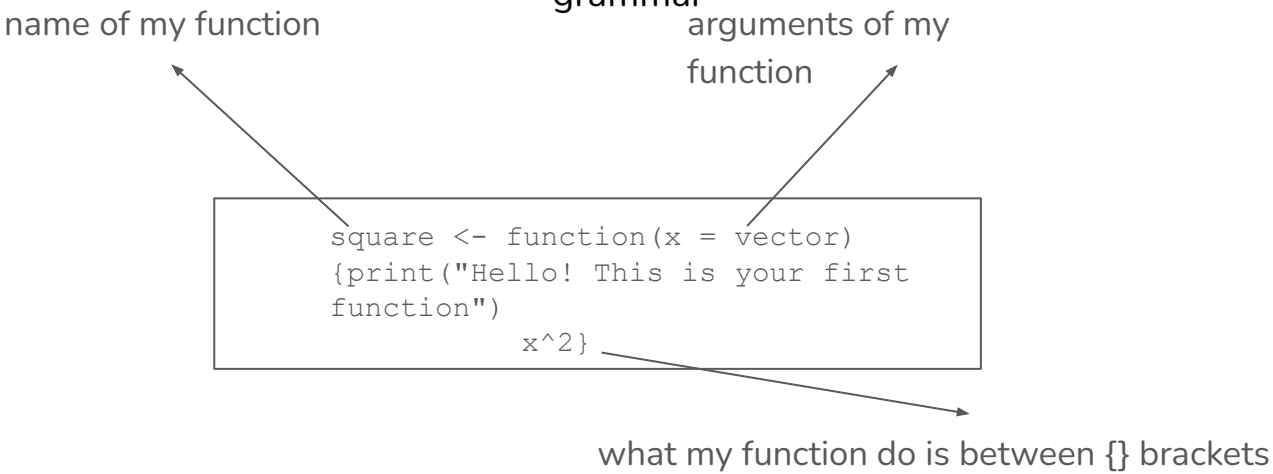


print() is a function to visualize something in the console

x^2 squares the x value/object

R object class :: functions

Everybody in R can write functions. Just a naive example to understand the grammar




```
> square(2)
[1] "Hello! This is your first function"
[1] 4
```

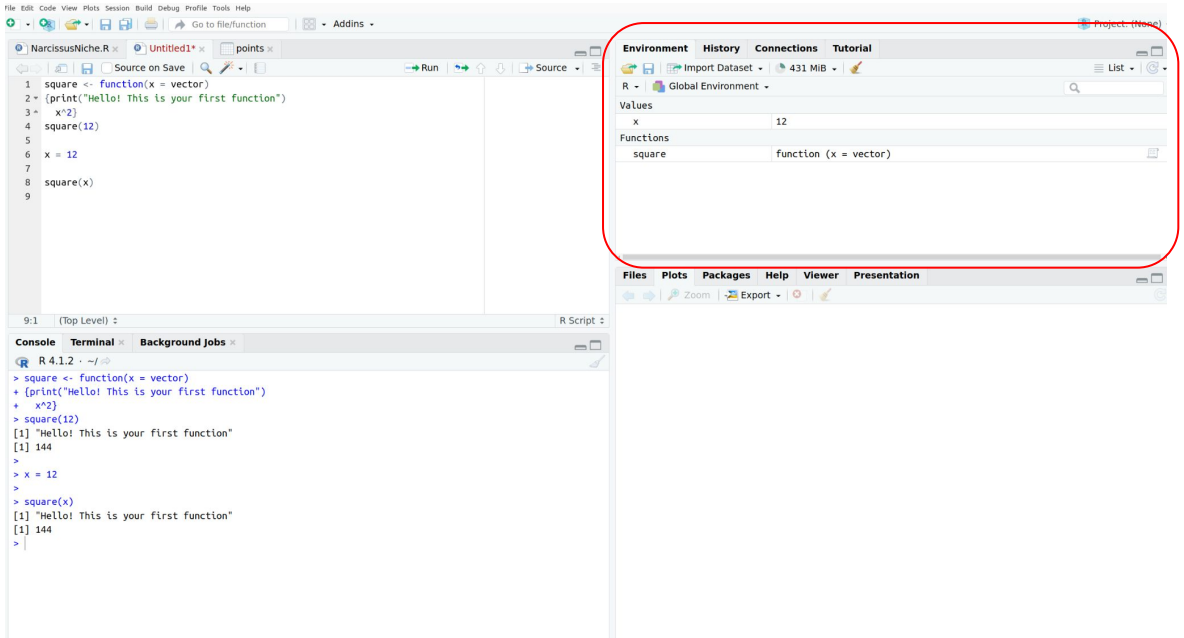
OR

```
> object_1 <- 2
> square(object_1)
[1] "Hello! This is your first function"
[1] 4
```

as “x” inside the function you can simply write a number, but you can also insert an object already exisiting.

R object class :: functions

After the previous step you will have new object (a function in this case) in the Rstudio object list so you can use it!



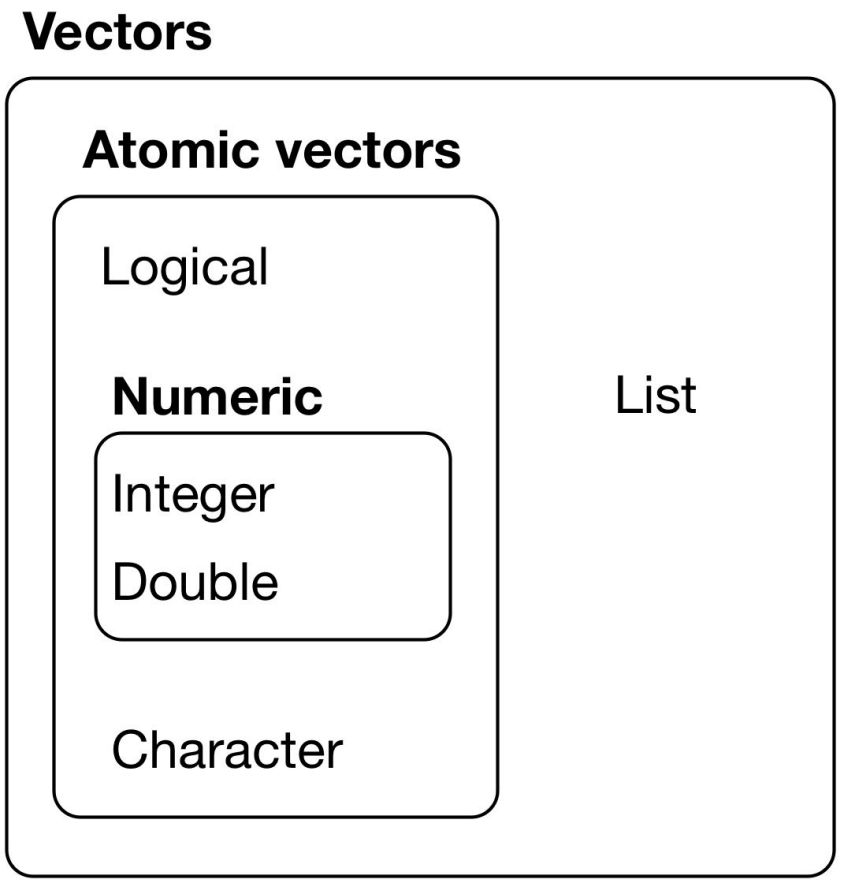
R object class :: functions

Most of the time you don’t need to write your own functions but you have to look for the functions you need among already available R packages.

Some shortcuts:

```
?factor                 # displays the function help page
args(factor)            # prints the arguments the function can take
```

R object class :: vectors



To be honest we are going to tell you something about “**atomic vectors**”

BUT

it is more practical to call them just “**vectors**”

R object class :: vectors

vector = a sequence of elements (number, characters, ...) assigned using the **c()** function which combines its arguments

simple vector

```
> my_vector <- c("first_element", "second_element", number, …)
```

named vector

```
> my_vector <- c("name_1"="first_element", "name_2"="second_element",
"name_3"="second_element", …)
```

display vector elements name

```
> names(my_vector)
```

R object class :: vectors

Vector can contain only one data type (homogeneous), otherwise R tries to convert the content to avoid losing information.

```
> c(FALSE, 2)
[1] 0 2
```

This is called implicit **coercion**.

The coercion rule goes **logical** -> **integer** -> **numeric** -> **character**.

You can also coerce vectors explicitly using the as.<class_name>. Example:

```
> x<-c(FALSE, 2)
> as.logical(x)           as.character(x)           as.numeric(x)
[1] FALSE TRUE          [1] "0"   "2"             [1] 0     2
```

R object class :: vectors :: exercise

Exercise1: write a numeric vector in a R object named "my_first_vector"

Exercise2: copy the characters vector in a R object named "colors"

```
# Write a simple vector and save in the object “colors”
> colors<-c("red", "blue", "yellow", "orange", "green", "brown")
```

R object class :: vectors

Once you have an object, you can check its features with some simple functions

```
# Check vector length with length() function
> length(colors)
[1] 6

# type of data in vector
> class(colors)
[1] "character"

# str() gives an overview of the  structure of an object and its elements
> str(colors)
 chr [1:6] "red" "blue" "yellow" "orange" "green" "brown"
```

R object class :: vectors

Or modify them

```
# unique values in vectors (when there are repeated values)

> unique(colors)

[1] "red"    "blue"   "yellow" "orange" "green"  "brown"


# add elements to a vector

> colors <- c(colors, "purple")  # add to the end of the vector

[1] "red"    "blue"   "yellow" "orange" "green"  "brown"  "purple"


> colors <- c("white", colors)  # add to the beginning of the vector

[1] "white"  "red"    "blue"   "yellow" "orange" "green"  "brown"
```

R object class :: vectors

Some shortcuts to create vectors

```
>seq(1, 10, by=1) # sequence of numbers from x to y
[1]  1  2  3  4  5  6  7  8  9 10

>vector <- rep(c("A", "B"), 2, each=2) # repeat elements in vectors
[1] "A" "A" "B" "B" "A" "A" "B" "B"

>vector_1 <- sample(seq(1,30), 5) # generate vector with sampled random elements
[1] 22  8  7 18 28
```

R object class :: vectors :: exercise

Produce the following three different vectors

1. the sequence of number (from 3475 to 3500 at 0.5 steps) and check the vector length

2. repeated sequence of factors (repeat the words cat, tree and dog two times each and generate a vector of length 18

3. vector of two random numbers between 0 and 1 (considering two numbers after the decimal point)

R object class :: vectors

Subsetting vectors: to extract values from vectors positional indices should be provided in square brackets

```
> animals<-c("cat", "dog", "rabbit", "duck", "monkey", "fish")
> animals[2]
[1]  "dog"
```

R object class :: vectors

multiple elements can be selected

```
> animals[c(3,2)]
[1] "rabbit" "dog"
```

conditional subsetting (basic logical operators applies):

```
> numbers<-c(24, 5, 47, 6, 98, 2, 10, 144)
> numbers[numbers>60]
[1]  98 144
```

multiple operators can be used

```
> numbers[numbers<=5 | numbers>100 | numbers ==47]
[1]   5  47   2 144
```

logical operators:

| = OR

&& = AND

R object class :: vectors

Sum two or more vectors:

```
> c(1,1,1,1)+c(1,1,1,1)
[1]  2 2 2 2
```

if vectors are not the same length, the shorter will be recycled

```
> c(1,2)+c(1,1,1,1)
[1]  2 3 2 3
```

Arithmetic operations on vectors applies element-wise

```
> c(1,1,1,1)*5
[1]  5 5 5 5
```

Vectors concatenation

```
> c(c(1,2), c(1,1,1,1))
[1] 1 2 1 1 1 1
```

R object class :: vectors :: exercise

1. write a vector in an object called "num" made of 10000 numbers comprised in the interval between 2 and 20000
2. count how many values are >3000 or <50
3. sum elements number 385, 1001 and 7521

R object class :: matrices

```
> matrix(data = sample(seq(1,50)), nrow = 10, ncol = 5, byrow = F)
      [,1] [,2] [,3] [,4] [,5]
 [1,]    2   21   37   30   43
 [2,]   12   42    9    3   46
 [3,]    7   19   29   26   40
 [4,]   17   32   39   23   25
 [5,]   10   27   50   33   15
 [6,]    5   13    1   44    8
 [7,]   47   24    6   22   34
 [8,]   28   18   14   45   36
 [9,]   11   48   16    4   49
[10,]   41   31   38   20   35
```

Matrices are grouped NUMERIC vectors; you can create matrices starting from several different vectors.

R object class :: matrices :: exercise

Exercise1:

create 3 different vectors of length 45:

- vector 1 should be created using seq()
- vector 2 should be created using rep() repeating 45, 45 times
- vector 3 should be created using sample() extracting 45 random numbers between 1 and 100

Exercise2:

- build your own matrix named "matrix" merging the previous vectors with the rbind() function. rbind mean rows bind and it is useful to create a matrix where vectors are the rows.

- ? what happen if you use the cbind function ?

R object class :: dataframe

A dataframe is a group of any type of vectors (character, number, integer, factor). Dataframes are the most common object in R

Manually write a dataframe using **data.frame()** function

```
> df<-data.frame("colname_1"=rep(c("A", "B"), 2, each=2),              # vector 1
     "colname_2"=sample(seq(1,30), 8),                                 # vector 2
     "colname_3"=seq(8,11.5, by=0.5))                                  # vector 3

> df
  colname_1 colname_2 colname_3
1         A         9       8.0
2         A        27       8.5
3         B        13       9.0
4         B        30       9.5
5         A        18      10.0
6         A        29      10.5
7         B        21      11.0
8         B        10      11.5
```

R object class :: list

a list is a group of any type of objects (vector, matrix, data.frame). Function is list(), each object in the list can be identified by a name

```
> my_list<-list("vector"=vector, "matrix"=matrix, "dataframe"=df)

> my_list
$vector
[1] "A" "A" "B" "B" "A" "A" "B" "B"

$matrix
     A B
[1,] 1 2
[2,] 1 2
[3,] 1 2

$dataframe
  colname_1 colname_2 colname_3
1         A        13       8.0
2         A         5       8.5
3         B        20       9.0
4         B        21       9.5
5         A         9      10.0
6         A        28      10.5
7         B         8      11.0
8         B        12      11.5
```

R object class

each object can be duplicated (or overwritten!)

```
>df2 <- df
```

```
>my_list2 <- my_list
```

each object can be removed by **rm()**

```
>rm(my_list2)
```

R object class

As well as for vectors you can:

Find the type of object

```
> class(df)
```

```
[1] "data.frame"
```

inspect object (find the data structure of a R object)

```
> str(df)
```

```
'data.frame':    8 obs. of  3 variables:
 $ colname_1: chr  "A" "A" "B" "B" ...
 $ colname_2: int  13 22 18 23 10 29 8 7
 $ colname_3: num  8 8.5 9 9.5 10 10.5 11 11.5
```

R object class

As well as for vectors you can:

Inspect object dimensions

```
> dim(df)         # applies to data.frame and matrices (rows x columns)
[1] 8 3
> length(vector)  # applies to vectors only
[1] 8
```

Peep the first (or last) rows of a dataframe/matrix

```
>head(df, n = howmanyrows)  # n = 5 by default
>tail(df, n = howmanyrows)
```

R object class :: special objects

We are not here to teach specific packages, but keep in mind that several package use the "special objects" they are often lists built with specific parameters!

This is the case of graphics objects (ggplot...), but also very common in genetics

examples:

- APE (phylogeny)
- POPPR (population genetics)
- PHYLOSEQ (metabarcoding)
- DESeq (functional expression)

... and many other...

R object class :: rows and columns name

Each columns or rows in a given object (matrix, dataframe) can have a name.

display column names: colnames()

```
> colnames(object)
```

display row names: rownames()

```
> rownames(object)
```

The same function can be used to change or assign new row/column names:

```
> colnames(object)<-c("colname1", "colname2")
```

R object class :: rows and columns transposition

Columns and rows can be flipped by transposition using the function **t()**

```
> matrix
     [,1] [,2]
[1,]    1    2
[2,]    1    2
[3,]    1    2


> t(matrix)
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
```

R object class :: objects subsetting

Often you need to select only some column/s or rows of a given object to perform operations:

`object[rows, columns]` **`object[columns]`**

select **row** 6 of the object df

```
> df[6,]
  colname_1 colname_2
6         A        12
```

select **column** 2 of the object df

```
> df[,2]
[1]  9  3 25  4 26 12 20 11
```

R object class :: objects subsetting

How to **select a range** from row 6 to row 8 of the object df

```
> df[6:8,]
  colname_1 colname_2
6         A        12
7         B        20
8         B        11
```

... same for columns

The same way can be used to remove columns/rows using "-".

```
> df[-2,] # remove the second row of the df

  colname_1 colname_2
1         A         9
3         B        25
4         B         4
5         A        26
6         A        12
7         B        20
8         B        11
```

R object class :: objects subsetting

non-contiguous columns/rows interval must be provided as a vector!

```
> df[,c(1,3)]
  colname_1 colname_3
1         A       8.0
2         A       8.5
3         B       9.0
4         B       9.5
5         A      10.0
6         A      10.5
7         B      11.0
8         B      11.5
```

R object class :: rows and columns indexing

Columns in dataframes objects can be selected by their names or created using "$" symbol

```
> df$colname_1
[1] "A" "A" "B" "B" "A" "A" "B" "B"
```

Remove columns in dataframes with $ and the NULL operator

```
> df2$colname_2 <- NULL
> df2
  colname_1 colname_3
1         A       8.0
2         A       8.5
3         B       9.0
4         B       9.5
5         A      10.0
6         A      10.5
7         B      11.0
8         B      11.5
```

R object class :: objects subsetting

Indexes can be used with logical operators to subset values

```
> df[df$colname_1=="B",]
  colname_1 colname_2 colname_3
3         B        18       9.0
4         B        23       9.5
7         B         8      11.0
8         B         7      11.5

> df[df$colname_1=="B" && df$colname_2<10,]

  colname_1 colname_2 colname_3
7         B         8      11.0
8         B         7      11.5
```

R object class :: objects subsetting

select element in **lists**

```
> my_list[[1]]  # positional selection (select the first element)
[1] "A" "A" "B" "B" "A" "A" "B" "B"

> my_list[["df"]] # selection by name
  colname_1 colname_2 colname_3
1         A        13       8.0
2         A         5       8.5
3         B        20       9.0
4         B        21       9.5
5         A         9      10.0
6         A        28      10.5
7         B         8      11.0
8         B        12      11.5
```

R object class :: objects subsetting

select column in a dataframe included in a list

```
> my_list[["df"]]$colname_1
[1] "A" "A" "B" "B" "A" "A" "B" "B"
```

select rows (or columns) in element included in a list

```
> my_list[["df"]][1:5,]
  colname_1 colname_2 colname_3
1         A        13       8.0
2         A         5       8.5
3         B        20       9.0
4         B        21       9.5
5         A         9      10.0
```

R object class :: objects subsetting

select column in a dataframe included in a list

```
> my_list[["df"]]$colname_1
[1] "A" "A" "B" "B" "A" "A" "B" "B"
```

select rows (or columns) in element included in a list

```
> my_list[["df"]][1:5,]
  colname_1 colname_2 colname_3
1         A        13       8.0
2         A         5       8.5
3         B        20       9.0
4         B        21       9.5
5         A         9      10.0
```

R object class :: objects subsetting

Filtering and subsetting are the empowered version of row/columns selection that we explained before, but they use specific functions in specific packages such as:

subset()

dplyr::filter()

they are based on logics. The grammar for logics in R is simple:

== for equal (not "="

!= for different

> < major an minor (followed by = to include the limit value)

| for OR (this is the "pipe" character usually is before the 1 in keyboards)

& or && for AND

! for NOT

R object class :: objects subsetting

`subset()` and `dplyr::filter()` are similar and useful function to select certain values in a dataframe. They are created as shortcuts to make easier the subsetting operations explained in the slide before.

```
>subset(df, colname_1 == "A")
  colname_1 colname_2 colname_3
1         A        13       8.0
2         A         5       8.5
5         A         9      10.0
6         A        28      10.5
```

subsetting by factor

```
>subset(df, colname_2 < 13)
  colname_1 colname_2 colname_3
2         A         5       8.5
5         A         9      10.0
7         B         8      11.0
8         B        12      11.5
```

subsetting by value

R object class :: objects subsetting

```
>subset(df, colname_1 %in% c("B", "C"))    # reads like: "is contained in.."
colname_1 colname_2 colname_3
1         B        18       9.0
2         B        23       9.5
3         B         8      11.0
4         B         7      11.5

>subset(df, !colname_1 %in% c("B", "C"))   # reads like: "is NOT contained
in.."
colname_1 colname_2 colname_3
1         A        13       8.0
2         A        22       8.5
3         A        10      10.0
4         A        29      10.5
```

R object class :: objects subsetting :: exercise

Create a dataframe from the following vectors:

```
> n <- 1:10
> sex <- rep(c('male', 'female'), 5)
> age <- c(23, 22, 21, 22, 24, 30, 23, 29, 19, 29)
> weight <- c(72, 90, 120, 80, 75, 65, 91, 58, 78, 50)
> height <- c(171, 185, 210, 170, 189, 150, 168, 165, 188, 143)
```

Question 1: Which is the mean age? [use mean() function]

Question 2: Which is the minimum male weight? [use function min()]

Question 3: add the column “ratio” calculated as weight/height

Question 4: subset only female between 15 and 29 yo taller than 165. How many observations do you get?

the working directory

"wd" is where you're working and where R looks for files to import or to save outputs.

Default wd is into the R folder in your home.

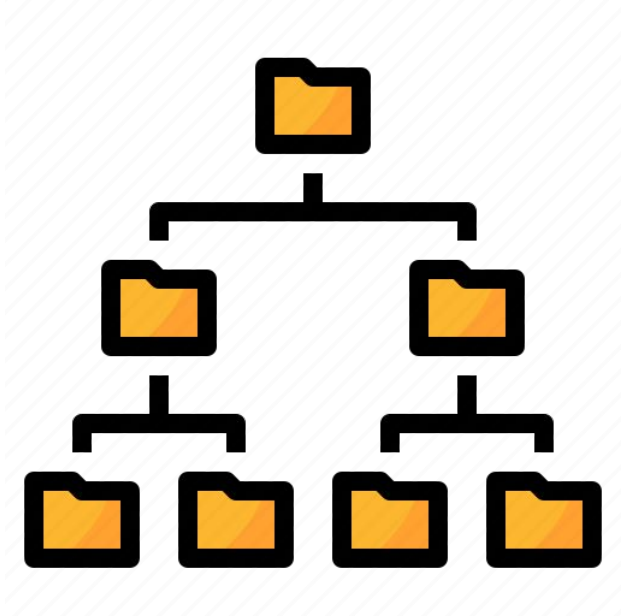
We suggest to set as wd the folder where your input files are stored.

```
setwd("your_path")
getwd()
```

Example path formats in different OS.

/ubuntu/unix/macos/your_dir/your_filename

"C:\Users\your_folder\your_filename"



Save your session

***.R** files # your main script (often very light text file data)

***.RData** files # your environment (all the objects you saved, often very big files (>1Gb)

- You can easily save *.R and *.RData files through the RStudio interface
- you can also manage your environments through the R command line (save you some time)

*.RDS files # Serialization Interface for Single Objects: are files that stores a single R object.

```
# save your environment
> save.image(file = "your_path/your_envirnoment_name.RData")
```

Restore your session

```
# load a *.Data file in your current session
> load(file = "your_path/your_envirnoment_name.RData")

# make searchable a stored environment without loading it
> attach(file = "your_path/your_envirnoment_name.RData")
Note: don’t forget to detach() your environment!

# load into your session functions or object from another script file
> source("my_script.R")
>
source("https://raw.githubusercontent.com/mchialva/myNGS_tools/master/TaxR.R"
```

Packages (libraries)

Packages are extensions of the coding language that add functionalities to R:

- packages contain functions, example data and manuals (called "vignettes").
- packages can use functions from other packages (dependency)
- you'll likely need to install dozen of packages to perform your analyses
- packages need to be loaded at the beginning of each session to be used in you script.

  > library(package_name)

Suggestions: list all the packages you are using in your script in structured way (in proper sections of the script).

Functions with same names but different usage/behaviour can exist in different packages.

packages installation

- **CRAN**: it is a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R.

  `>install.package("pkg.name")` OR via RStudio interface

- **Bioconductor**: develop, support, and disseminate free open source software that facilitates rigorous and reproducible bioinfo analysis

  `>BiocManager::install(c("package1", "package2"))`

- **From external repositories, e.g. from GitHub**: it is an online software development platform. It's used for storing, tracking, and collaborating on software projects

  `>remotes::install(c("package1", "package2"))`

- using **devtools**: it makes package development easier by providing R functions that simplify and expedite common tasks

  `>devtools::install_github("repository_name")`

packages installation :: exercise

1. install a few CRAN packages - we'll use most of them later on …

install a package from CRAN:

'devtools'

'BiocManager'

'reshape'

'tidyverse'


```
>install.packages(c("devtools", "BiocManager", …))
```

packages installation :: exercise

2. install a package from GitHub (https://github.com/jfq3/QsRutils)

Github packages are installed through the *devtools::install_github()* function

```
>library(devtools)
>install_github("jfq3/QsRutils")
```

OR

```
>devtools::install_github("jfq3/QsRutils")
```

packages installation :: exercise

3. install a package from Bioconductor (Biostrings)

Bioconductor packages are installed through the BiocManager::install() function

```
>BiocManager::install("Biostrings")
```

packages installation :: exercise

NB: If for some reason installation fails, R reports you some errors

After installation let's check:

1. if they are in the list of the installed packages
2. if they can be loaded. Use the function `library()`