

# Data Management with R (session 1)

M2 Statistics and Econometrics

Thibault Laurent

Toulouse School of Economics, CNRS

Last update: 2023-09-13

# Table of contents

1. General informations
2. Basic data manipulation
3. Importing data
4. Data cleaning




# 1. General informations

# Packages needed in this session



```
install.packages(c(
  "foreign", "jsonlite", "readr", "readxl", "sas7bdat",
  "XML", # import data
  "reticulate", # use Python
  "data.table", "ff", # big data
  "Matrix", # sparse matrix
  "classInt", "glue", "stringr", "wordcloud", # character treatment
  "gplots", # plotting data
  "tidyverse", "DSR", # Data Scientists toolkits
  "Amelia", "DMwR", "missForest", "naniar", # missing values treatment
  "sp", # spatial data object
  "zoo") # Time series analysis
)
devtools::install_github("hadley/emo")
devtools::install_github("edwindj/ffbase", subdir = "pkg")
py_install("pandas")
```

# What is R?

- **R** is a software dedicated to statistical and scientific computing using its own language. Actually, it is maintained by the *R Core Team*. It is multiplatform (Linux, Mac OS, Windows), free (included in GNU project) and can be downloaded from [CRAN website](#).
-  **How to install R?**
- It can be coupled with **C**, **C++**, **Fortran** (many base functions are coded in one of this language) and **Python**.
- It allows to realize both data management and statistical analysis (data visualization, machine learning, text mining, time-series analysis, spatial econometric, etc.).
- It includes around 30 base packages and a huge number of other packages which can be downloaded from CRAN or GitHub. To get the number of available packages on CRAN, we can do:

```
nrow(available.packages())
```



```
# [1] 19859
```

# R for economist?

- Unlike **Eviews**, **Gauss**, **SAS**, **SPSS**, **Stata**, etc. **R** is free.
- It can be used for:
  - data management,
  - data visualization,
  - data analysis,
  - programming new methods.
- Many packages are dedicated to econometrics: <https://cran.r-project.org/web/views/Econometrics.html> (many of them are related to time series analysis).
- It includes also many tools for optimization (see for instance [this document](#)).
- Free alternatives to **R**: **Python** (common for data management and machine learning on Big Data), **Julia** (common for speed programming).

# Why using RStudio?



- RStudio (<https://www.rstudio.com/>) allows to use a code editor multiplatform (alternatives to RStudio: Spyder, Tinn-**R**). It has direct access to:
  - the **R** console,
  - the figures,
  - the list of installed packages,
  - the list objects
-  **Presentation of RStudio**
- RStudio desktop is free. The Professional versions permit facilities to ODBC data connectors and server management. It is easily possible to :
  - access to Markdown for creating reports/presentations and making reproducible results (alternative: jupyter notebook).  **Presentation of R Markdown**
  - create interactive web applications with Shiny.
  - execute **Python** or **C++** code from the module.

# Basics operations

The `>` symbol in the console means that **R** is waiting some instructions to be executed. It could be some mathematical operations like addition, subtraction, multiplication, division, etc. For example:

```
(10 + 12 + 8.5) / 3
```

**Remark:** when **R** returns a scalar (which is actually a vector of size 1) the `[1]` at the beginning of a line, corresponds to the position of the first value of the printed line. If a line is starting with the `+` symbol, it means that an instruction has been executed but it is not complete. For example:

```
> 12 *  
+
```

The function `c()` allows to collect different values in a vector. To get the help of a function, use `help()` or `?` symbol. For example:

```
c(10, 12, 8.5)  
?c  
help(c)
```



# What is an object?

We can do object oriented programming in **R**. In fact, everything in **R** is an object. An object is a data structure having some attributes and methods which act on its attributes.

To create an object in **R**, we use the following syntax (note that the `<-` operator can be replaced by `=`):

```
name_object <- a_data_structure
```

For example we store in object **my\_vec** a vector of numeric values:

```
my_vec <- c(10, 12, 8.5)
```

**my\_vec** is now stored in the RAM. We can now apply many **R** functions which take as argument a vector of numeric. For example:

```
mean(my_vec)
```

```
## [1] 10.16667
```

# Some tips when creating a new object

Some names can not be used for creating a new object.

- there are some reserved words in **R** which can not be used for creating a new object. To know them, just print

```
?Reserved
```

- the name of an object can not start with a numeric value:

```
1a <- 5
```

**Remark:** the previous instruction causes an error message. An error breaks the execution of the current instructions.

Moreover, usually we do not use the name of an existing function for creating an object. To know if a function already exists, just print the name in the console. If there is no message error, it means that the function exists

```
c  
mean
```

# Some tips when assigning a new object

Usually, we assign a value to an object in 1 line of code. However, it is possible to:

- create two objects in 1 line of code. We use ; operator between the two assignments

```
a <- 5; b <- a - 1
```

- create one object in 2 lines of code; usually, we use the line break after a punctuation symbol:

```
d <- c(5, 15, 15, 14, 13, 12, 12, 12,  
      10, 10, 5, 40, 40, 25, 20, 10)
```

To print an **R** object, we can directly print the name of the object in the console or use the *print()* function:

```
print(d)  
d
```

# Save the objects created during a session

To print the names of the objects that we have created, we use function `ls()` or `objects()`. To remove one particular object (we do this when we create temporary variable), we can use function `rm()`:

```
ls()  
rm(a)  
objects()
```

Before leaving a session, we can save the objects created with `save()` function:

```
save(list = ls(), file = "session_1.RData")
```

The format used to save object with **R** is ".RData". It can contain any **R** object (vectors, matrices, **data.frame**, etc). Here, "session\_1.RData" has been created in the working directory. To print the working directory use function `getwd()`:

```
getwd()
```

# Load .RData files

To load a ".RData" file with **R**, we simply use the function *load()*:

```
load(file = "session_1.RData")
```

If the file "session\_1.RData" is not located in the working directory, there are two options to load it:

1. change the working directory with *setwd()* function. For example:

```
setwd("C:/Documents/my_path/to_the_file")
```

2. indicate the full name of the path:

```
load("C:\\Documents\\my_path\\to_the_file\\session_1.RData")
```

**Remark:** In **R** the "\" character is a special character. That is why we need to use two backslashes operator for being interpreted as a backslash:

```
cat("Break line: \n To print backslash : \\ \n")
```



# Training: exercice 1.1

To answer the exercise, try to use a **R** markdown document.

- Create the object **my\_vec** which contains a vector of numeric values: 28, 29, 35, 75, 40, 52, 23, 25, 10, 50.
- Compute the mean, min, max and standard deviation of **my\_vec** by using the functions *mean()*, *min()*, *max()*
- Compute the variance of **my\_vec** by using only the function *sum()*, *mean()*, *length()* (which gives the size of a vector). We remind that the variance of  $x_1, \dots, x_n$  is  $\frac{1}{n} \sum (x_i - \bar{x})^2$  where  $\bar{x}$  is the mean.
- Compare it with the result obtained by *var()* function
- Create object **my\_vec\_st** which subtracts the mean and divide by the standard deviation:
- Print the working directory (WD) and save objects **my\_vec** and **my\_vec\_st** in a file "exo1.RData"



# Training: exercise 1.2

- What is the difference between *library()* and *require()* ?
- Why these two syntaxes are working ?

```
require("stringr")  
require(stringr)
```

- Use the operator `::` to use the function *str\_to\_title()* included in the package **stringr** without calling *library()* or *require()*.



## 2. Basic data manipulation

### a. Vectors



# Vectors: Create, access and modify

Vector is a basic data structure in R. It contains elements of the same type like **double**, **integer**, **logical** or **character**. It can be created with the `c()` function:

```
a.numeric <- c(0.36, 64, 0.56, 0.44)
a.integer <- c(1L, 0L, 3L, 1L)
a.logical <- c(T, F, T, F)
```

It can also be initialized and then filled 1 by 1 or group by:

```
a.character <- character(4)
a.character[1] <- "France"
a.character[c(2, 4)] <- c("Belgium", "England")
a.character[-c(1:2, 4)] <- "Croatia"
```

Vectors are object with optional attributes like **names**:

```
names(a.numeric) <- a.character
a.numeric["France"]
```

# Vectors: data types coercion

If a character string is present in a vector, everything else in the vector will be converted to character strings. The other coercing rule is: if a vector only has logicals and numbers, then logicals will be converted to numbers; **TRUE** values become 1, and **FALSE** values become 0.


```
c(TRUE, 1, FALSE, pi)
```

Strings dominates numeric and logical. Numeric dominates logical:

```
c(TRUE, 1, FALSE, pi, "a_string")
```

🤔 In other languages, we would obtain an error message. **R** is automatically executing some stuffs that we can not see; it takes time and it explains why **R** is considered as slow compared to compiled languages.

# Vectors: Comparison

The operators of comparison are:  $\mathbf{x == y}$ ,  $\mathbf{x != y}$ ,  $\mathbf{x > y}$ ,  $\mathbf{x >= y}$ ,  $\mathbf{x < y}$ ,  $\mathbf{x <= y}$ . The size of  $x$  and  $y$  can be different and when it is the case, there is not necessary a  message.

```
a.numeric >= rep(0.5, 3)
a.numeric >= rep(0.5, 4)
```

Note that each operator is actually a function:

```
a.numeric >= 0.5
`>=`(a.numeric, 0.5)
```

It also works for strings (the rules of lexicography depends on the local setting):

```
a.character == "France"
a.character > "d"
```

# Vectors: Matching

To identify the positions of the element of  $x$  in  $table$ , use function  $match(x, table)$ :

```
clients_jour <- c("Dorian", "Inès")
base_clients <- c("Jordan", "Scottie", "Inès", "Dorian")
match(clients_jour, base_clients)
```

The operator  $x \%in\% y$  (very useful) looks if an element of  $x$  belongs to  $y$ . It is using function  $match()$ :

```
clients_jour \%in\% base_clients
```

Function  $which()$  returns the indices of the **TRUE** elements of a logical vector:

```
which(a.logical)
```

# Vectors: Multiple comparison

The operators `&`, `|` indicate logical AND, OR. They are vectorized.

```
a.numeric >= 0.5 & a.logical  
a.integer == 1L | !a.logical
```

 The operators `&&`, `||` are short cut (do not evaluate a term if not necessary) and are not vectorized (evaluate only the first element of a vector). For more informations see this [stackoverflow post](#)

```
a.numeric >= 0.5 && a.logical  
a.integer == 1L || !a.logical
```

The function `all()` verifies that all elements of a vector of logical are **TRUE**.  
`any()` verifies that there is at least one **TRUE**

```
all(a.numeric >= 0.0 & a.numeric <= 1.0)  
any(a.logical < 0L)
```


# Vectors: Operation

To modify a vector, insert a vector of logical or the number of indices in the `[]`

```
a.numeric[a.numeric > 1] <- a.numeric[a.numeric > 1] / 100  
a.integer[which(a.integer == 1L)] <- 2L
```

The different "mathematical" operators are: +, -, \*, /, ^, %%, %/%.

```
all.equal(100 * a.numeric, c(100, 100, 100, 100) * a.numeric)  
5 / 2 + 1.5 * a.numeric - 2.5 * a.numeric ^ 2
```

 The length of vectors can be different. In some cases, the types can also be different:

```
(1:9) ^ c(1, 2, 3)  
(a.numeric * a.logical) ^ a.integer  
a.character + a.numeric  
paste(a.character, a.integer, sep = " : ")
```

# Vectors: Mathematical vectorized functions

The algorithm to find the minimum of a vector is:

```
x <- rnorm(1000000)
system.time({
  n <- length(x)
  our_min <- x[1]
  for (i in 2:n) {
    if (x[i] < our_min)
      our_min <- x[i]
  }
  cat("our min is ", our_min, "\n")
})
```

🤔 Many base **R** functions are already vectorized (see [this post](#) for more informations). Moreover, these functions call **C**, **C++**, or **FORTRAN** program to carry out operations which explains why the computational time is better

```
system.time(min(x))
```

# Vectors: Mathematical functions for numeric type

We create a vector of **numeric** with one missing value (*NA*, for *Non Availabale* different that *NaN* for *Not a Number*)

```
age <- c(25, 28, 30, NA, 21, 26, 29, 31, NA, 22, 27)
```

```
min(age, na.rm = T)
max(na.omit(age))
age[is.na(age)] <- mean(age, na.rm = T)
range(age)
sum(age)
median(age)
quantile(age, probs = 0.9)
1 / length(age) * sum((age - mean(age)) ^ 2)
sd(age) == sqrt(var(age))
cumsum(age)
sqrt(age)
exp(log(age))
cos(age)
```



# Vectors: Useful functions

*rep()* replicates elements of vectors and lists :

```
rep(2, times = 5)
rep(c(1.2, 3.5), each = 2)
```

*rev()* reverses the elements of a vector:

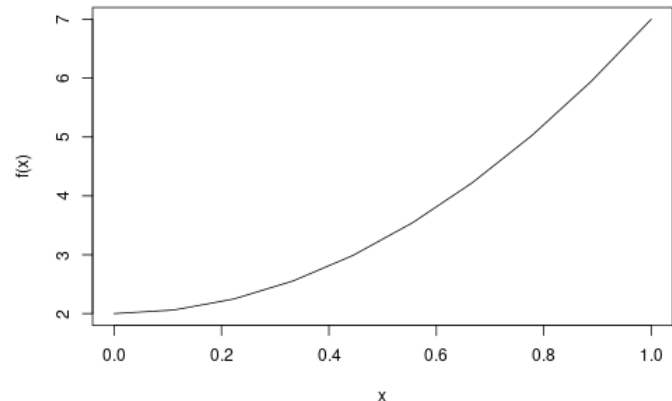
```
rev(age[order(age)])
```

*seq\_len(n)* is equivalent to *1:n* and *seq\_along(x)* is equivalent to *1:length(x)*

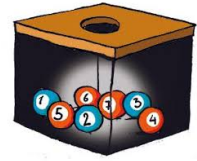
```
seq_len(5)
1:5
seq_along(rnorm(5))
```

To generate regular sequences:

```
x <- seq(from = 0, to = 1,
         by = 0.1)
x <- seq(from = 0, to = 1,
         length.out = 10)
f <- function(x) 2 + 5 * x ^ 2
plot(x, f(x), type = 'l')
```



# Vectors: Useful functions (2)



We can make random permutation with `sample()`:

```
sample(age)
```

To make random sampling, use option **size**.

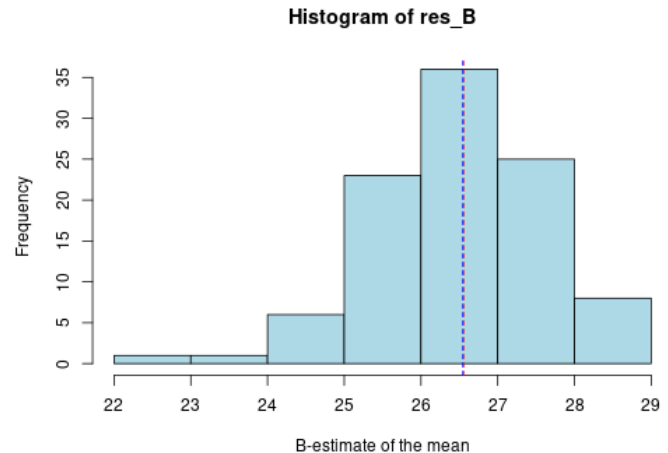
```
sample(1:45, size = 5)
```

Option **replace** leads to sample with replacement (useful for bootstrap algorithm).

```
my_boot <- function(x, B) {  
  res <- numeric(B)  
  for(i in 1:B)  
    res[i] <- mean(sample(x, replace = T, size = 5))  
  res  
}
```

Application:

```
res_B <- my_boot(age, 100)  
hist(res_B, xlab = "B-estimate of the mean",  
      col = "lightblue")  
abline(v = mean(age, na.rm = T))  
abline(v = mean(res_B), col = "red")
```



# Vectors: union, intersection

We define two sets of elements:

```
A <- 1:10  
B <- c(3:6, 12, 15, 18)
```

- Union: can be seen as the unique values of the vector containing  $A$  and  $B$ :

```
unique(c(A, B))  
union(A, B)
```

- Intersection: can be seen as the values of  $A$  included in  $B$

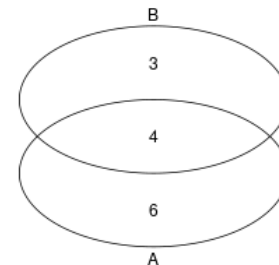
```
A[A %in% B]  
intersect(A, B)
```

- Differences:

```
setdiff(A, B)  
setdiff(B, A)
```

- Venn Diagram

```
gplots::venn(list(A = A, B = B))
```





# Training

## Exercise 2.1

- Describe what's gonna happen for each line of code:

```
c(21, 180, "F", "DU", "FR", TRUE)
TRUE | this_object_does_not_exist
TRUE || this_object_does_not_exist
c(1, 1, 1, 1) ^ c(0, 1) + c(0, 1, 2)
```

- **R** includes a lot of base functions which can be seen [here](#). Choose 5 of them, describe and illustrate them.
- Plot the function  $f(x) = \frac{1}{\sqrt{2\pi}} \exp(-\frac{1}{2}x^2)$  for  $x \in [-4, 4]$
- By using the function `sample()` draw a random sample of size 100 of a Bernoulli distribution with  $p = 0.5$
- What are the differences between `sort()`, `order()` and `rank()`?



## 2. Basic data manipulation

### b. Text mining

# Vector of strings: manipulation (1)


For a good introduction to text data, visit [this web page](#) or [this course](#).

A **string** is a collection of characters. It is stored in a vector

```
a_1 <- c("String 1", "String 2")
a_2 <- c("String 1\n", "String\t2")
```

The function `nchar()` computes the number of characters of each element

```
nchar(a_1)
nchar(a_2)
```

 "`\n`" or "`\t`" are considered as special characters and "`"`" is considered as character. The special characters can be seen [here](#). `cat()` allows to evaluate the special characters.

```
cat(a_2)
```

# Vector of strings: manipulation (2)

- *substr()* allows to extract characters with respect to the position.

```
substr(c("code_A1", "code_A2"), start = 6, stop = 8)
```

- *paste()* concatenates strings of different vectors. A character vector can be collapsed to a string by using argument **collapse**.

```
paste(c("Opel", "Peugeot"), rep(2005:2009, each = 2), sep = "_")  
paste(c("one", "character", "from", "a vector"), collapse = " ")
```

- *toupper()* and *tolower()* convert all the character letters in upper or lower.

```
toupper(a_1)  
tolower(a_2)
```

- *abbreviate()* abbreviate strings .

```
abbreviate(c("Bosnie-Herzégovine", "Burkina Faso", "Côte d'Ivoire")
```

# Vector of strings: pattern matching (1)

```
aa <- c("There are 7 words in this sentence.",  
       "Just 4 words here.")
```

- *grep()* allows to identify the elements of a vector *x* which contain a *pattern* of characters.

```
grep(pattern = "words", x = aa)
```

- *agrep()* allows some differences in the pattern:

```
agrep(pattern = "wards", x = aa)
```

- *regexpr()* indicates at which position of the string, the pattern has been found:

```
gregexpr(pattern = "words", text = aa, ignore.case = TRUE)
```




# Vector of strings: pattern matching (2)

- *strsplit()* splits the elements of a character vector when a pattern has been identified

```
res <- strsplit(aa, " ")
sapply(res, length)
```

- *gsub()* search for matches to argument *pattern* within each element of a vector *x* and do *replacement*:

```
gsub(pattern = "words", replacement = "mots", x = aa)
```

 Instead of using a pattern with a specific string, we can use a regular expression. For example, to identify the elements of a vector which contains one of this character : "0", "1", "2", "3", "4", "5", "6", "7", "8", "9":

```
gsub(pattern = "[[:digit:]]", replacement = "X", x = aa)
```

# Regular expression (1)

A regular expression is a pattern that describes a set of strings (for more informations, see *help(regex)* or [this web page](#)):

```
textes <- c("b.bi", "bibé", "tatane", "bAbA", "tbtc",  
           "tut", "byb=", "baba", "b\nb1", "t5t3")
```

- In a regular expression, "." replaces any characters

```
print(grep("b.b.", textes))  
print(grep("b\\.b.", textes))
```

- [aeiouy] indicates that "a", "e", "i", "o", "u" or "y" are allowed. ^ is the negation. - gives a sequence and [:digit:] allows any unicode digits

```
print(grep("t[aeiouy]t[aeiouy]", textes))  
print(grep("t[^aeiouy]t[^aeiouy]", textes))  
print(grep("t[a-z]t[a-z]", textes))  
print(grep("t[[:digit:]]t[[:digit:]]", textes))
```

# Characters: Clean tweets

Text mining is very useful for the analysis of tweets.

```
tweet <- c("TopStartupsUSA: RT @FernandoX: 7 C's of Marketing in the  
"#Analytics #MachineLearning #DataScience #MalWare #IIoT",  
"YvesMulkers: RT @wil_bielert: RT @neptanum: Standard Model Physi  
"#BigData #Analytics #DataScience #AI #MachineLearning #IoT #IIoT
```

Analysis of tweets:

```
correct <- gsub("(RT|via)((?:\\b\\W*@\\w+)+)", "", tweet)  
correct <- gsub("@\\w+", "", correct)  
correct <- gsub("[[:punct:]]", "", correct)  
correct <- gsub("[[:digit:]]", "", correct)  
correct <- gsub("http\\w+", "", correct)  
correct <- gsub("[\\t ]{2,}", " ", correct)  
correct <- gsub("^\\s+|\\s+$", "", correct)  
correct <- iconv(correct, "UTF-8", "ASCII", sub="")
```

Package **rtweet** allows to import data from twitter (see vignette [here](#))

# Characters: Word cloud

What can we do with words?

```
word <- unlist(strsplit(correct, " "))
tab_word <- table(word)
wordcloud::wordcloud(names(tab_word), tab_word)
```



See also [this web page](#) for more informations about "Text Mining"

# Characters: Dedicated packages (1)



There exist some packages which try to simplify the **R** base code. Some of them belong to the **Tidyverse project**.

```
library(tidyverse)
```

For example, to count the number of times character "a" is appearing in the elements of a vector, we present here the code by using **R** base code and by using **stringr** package:

```
res1 <- gregexpr(pattern = "a", text = word, ignore.case = T)
sapply(res1, function(x) ifelse(x[1] > 0, length(x), 0))
stringr::str_count(word, "a")
```

The package contains many other functions (see the vignette [here](#)). For example *str\_pad()* allows to fill the elements of vector until the number of character is constant:

```
vec_to_change <- c("1", "10", "105", "9999", "0008")
stringr::str_pad(vec_to_change, 4, pad = "0")
```

# Characters: Dedicated packages (2)



To evaluate these objects into string, we will use *paste()* and *cat()* functions

```
name <- "Fred"
anniversary <- as.Date("1991-10-12")
age <- as.numeric(floor((Sys.Date() - anniversary)/365))
cat(paste0("My name is ", name,
          ", my age next year is ", age + 1,
          ", my anniversary is ", format(anniversary, "%A, %d %B, %Y")))
```

whereas with **glue** package, we will do

```
require("glue")
new_object <- glue('My name is {name},',
  ' my age next year is {age + 1},',
  ' my anniversary is {format(anniversary, "%A, %d %B, %Y")}.')
new_object
```



# Training

## Exercise 2.2

Let consider the vector of strings **my\_word**:

```
my_word <- c("we went 2 times to warwick",  
            "moi 1 fois 1 w-e")
```

- give the character position of "we" in **my\_word**
- give the character position of "w" or "e" in **my\_word**
- give the character position of "we" in **my\_word**, knowing that there is a empty space before
- give the character position of any numbers in **my\_word**
- count the number of times any numbers is appearing in **my\_word**



## 2. Basic data manipulation

c. Matrices, lists and `data.frame`



# Matrix: Create

Matrix is a table in 2D which contains elements of the same type. It can be created with the `matrix()` function which consists in transforming a vector into a matrix. By default the matrix is filled by column (use **byrow** argument otherwise).

```
a.vec <- c(25, 26, 30, 31, 26, 27, 29, 30, 22, 23)
a.matrix <- matrix(a.vec, nrow = 5, ncol = 2, byrow = T,
                  dimnames = list(letters[1:5], c("y.2005", "y.2006")))
```

It has a **dim** attribute of length 2 (number of rows and columns) and an optional **dimnames** attribute, a list of length 2 with the names of rows and columns.

```
attributes(a.matrix)
```

It can also be done by merging vectors by row (`rbind()`) or by column (`cbind()`):

```
a.matrix.1 <- rbind(a.character, a.logical)
colnames(a.matrix.1) <- paste0("team_", 1:4)
rownames(a.matrix.1) <- c("country", "winner")
a.matrix.2 <- cbind(a.integer, a.numeric)
```

# Matrix: Access and modify

Elements can be accessed as `var[row, column]`. Here `row` and `column` are vectors of **integer** (the indices), **character** (the names) or logical.

```
a.matrix[c(2, 4), 2]
a.matrix[c("b", "d"), "y.2006"]
a.matrix[a.matrix[, 1] >= 30, ]
a.matrix.1[ , dimnames(a.matrix.1)[[2]] %in% c("team_1", "team_4")]
```

We can combine assignment operator with the above learned methods for accessing elements of a matrix to modify it.

```
a.matrix[2, 2] <- round(a.matrix[2, 2], 0)
```

We can also use `cbind()` and `rbind()` for adding columns or rows

```
a.matrix <- cbind(a.matrix, y.2007 = a.matrix[, 2] + 1)
a.matrix <- rbind(a.matrix, f = c(31, 32, 33))
```

# Matrix: Basic operations

We use the same operators `+`, `-`, `*`, `/`, `^`, `%%`, `%/%` than for vectors. There are some rules with respect to the dimension : a matrix of size  $(n, p)$  can be associated with a scalar and a vector of size lower than  $np$ .

```
1 + 2 * a.matrix - 0.5 * a.matrix ^ 2  
a.matrix + c(1, 2, 3)
```

which is equivalent to:

```
a.matrix + matrix(c(1, 2, 3), nrow(a.matrix),  
                  ncol(a.matrix))
```

A matrix can be associated with another matrix iff the dimensions coincide:

```
a.matrix[2:5, 1:2] + a.matrix.2 ^ 2  
a.matrix[1:2, ] + a.matrix.2 ^ 2
```

# Matrix: function *apply()*

*apply()* is a common way for doing a *for loop* instruction. For example, instead of doing:

```
my_res <- numeric(ncol(a.matrix))
for (k in 1:nrow(a.matrix)) {
  my_res <- my_res + a.matrix[k, ]
}
my_res
```

We can do:

```
apply(a.matrix, MARGIN = 2, FUN = sum)
```

The second argument of the function indicates the dimension on which applying the function **FUN**. The last argument can be an existing or personal function.

```
apply(a.matrix, 1, function(x) c(min(x), max(x)))
```

# Matrix calculation (1)

```
y <- c(178, 180, 165, 158, 183)
x <- cbind(x0 = 1, x1 = c(0, 0, 1, 1, 0),
          x2 = c(43, 43, 40, 39, 45))
```

- `t()` give the transpose of a matrix. Note that a transpose of a vector belongs to the **matrix** class of object.

```
t(x)
t(y)
```

- `%*%` allows to multiply two matrices if they are conformable

```
t(x) %*% y
t(x) %*% x
```

- `crossprod(x, y)` is optimized to compute  $x^T y$

```
crossprod(x, y)
crossprod(x)
```

# Matrix calculation (2)

- Note that a vector in  $\mathbf{R}$  is considered as row-vector and column-vector ( $\mathbf{R}$  is actually doing the verification)

```
y %*% y  
t(y) %*% y
```

- `solve(A, b)` allows to resolve the linear problem  $Ax = b$ . For example:

```
A <- crossprod(x)  
b <- crossprod(x, y)  
solve(A, b)
```

It allows to find the inverse of a square matrix by resolving  $Ax = I$ :

```
solve(A) %*% b
```

- To inverse a symmetric definite positive matrix, use instead the Cholesky or  $QR$  factorization (more stable)

```
chol2inv(chol(A)) %*% crossprod(x, y)  
qr.solve(A, b)
```

# Matrix calculation (3)

To deal with sparse matrix, we recommend to use **Matrix** package (or alternatively **spam**).

```
library("Matrix")
mat <- matrix(rbinom(10000, 1, 0.05), 100, 100)
object.size(mat)
Mat <- as(mat, "Matrix")
object.size(Mat)
```

The same functions (*crossprod()*, *solve()*, etc.) and operators (%\*%) can be applied and will be faster.



# Training: exercise 2.3

We consider the two vectors **weight** and **group**

```
ctl <- c(4.17, 5.58, 5.18, 6.11, 4.50, 4.61, 5.17, 4.53, 5.33, 5.14)
trt <- c(4.81, 4.17, 4.41, 3.59, 5.87, 3.83, 6.03, 4.89, 4.32, 4.69)
group <- gl(2, 10, 20, labels = c("Ctl", "Trt"))
weight <- c(ctl, trt)
```

- create a matrix **X** of dim  $20 \times 2$  which contains in the first column 1 if `group="Ctl"`, 0 otherwise, and contains in the second column 1 if `group="Trt"` and 0 otherwise.
- What does the following command do?

```
split(weight, group)
```

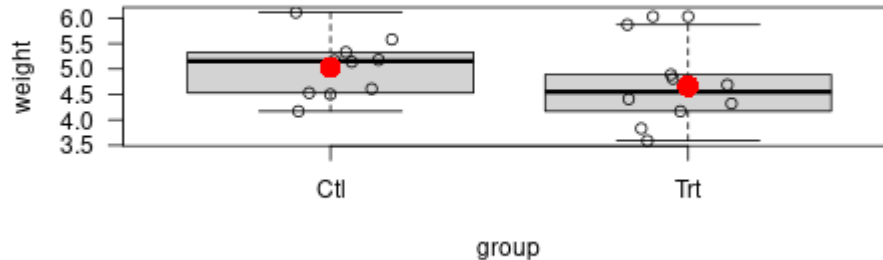
- Compute the mean of **weight** in the two groups "Ctl" and "Trt"





# Training: exercise 2.3

- On the following figure, do you think there is a difference of weight between the two groups ?



- create matrix  $A = (X'X)$  and  $b = (X'y)$  where  $y$  is the **weight**
- solve the equation  $A\beta = b$ . We call **hat\_beta** the solution of the equation
- compute the adjusted values  $\hat{y} = X\hat{\beta}$ . We call **hat\_y** this vector.
- compute the vector of residuals  $\hat{e} = y - \hat{y}$ . We call **hat\_e** this vector



## Training: exercise 2.3

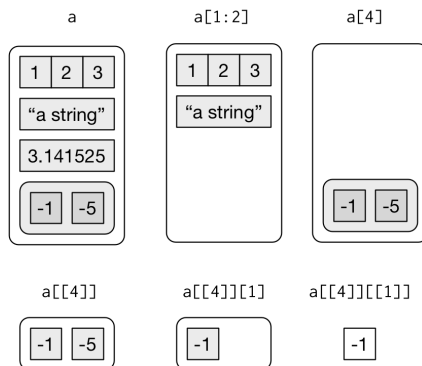
- Compute the sum of squares of the residuals. We call **sse** this value:
- compute the residual standard error which is equal to  $\sqrt{SSE/(n - 2)}$
- compute **SST** which is equal to  $\sum (y - \bar{y})^2$
- compute **SSR** which is equal to  $\sum (\hat{y} - \bar{y})^2$
- verify that  $SST = SSR + SSE$
- compute the  $R^2$  which is equal to  $SSR/SST$

# List: Create and access

A **list** contains several elements of different types and length. It can be created with the `list()` function:

```
a <- list(a_vector = 1:3,  
         a_character = "a string",  
         a_scalar = pi,  
         b_list = list(-1, -5))
```

Access the elements of a list (extract of the H. Wickham *R Advanced book*)



? What are the results of

```
a$a_vector  
a[[1]]  
a[1]  
a[1:2]  
a[[4]][1]  
a[[4]][[1]]
```

# List: Manipulation (1)

A list can be modified with `[]` or `$`:

```
a[[1]] <- a[[1]] + 1
a$a_scalar <- pi * 5 ^ 2
a$a_matrix <- matrix(c(1, 0, 1, 3, 2, 1),
                     ncol = 3, byrow = T)
```

It can contain a function:

```
a$a_function <- function(x) x ^ 2
a$a_function(2)
```

The informations relative to a list can be obtained with `length()`, `names()` or `str()`.

```
length(a)
names(a)
str(a)
```

# List: Manipulation (2)

To apply a function to each element of a list, use function *lapply()*.

```
lapply(X = a, FUN = length)
```

Function *sapply()* is doing the same thing but tries (when it is possible) to return the results in a more elegant way (a vector or a matrix for example)

```
sapply(X = a, FUN = length)
```

Argument **FUN** can contain an existing or a self-programmed function

```
lapply(X = a, FUN = function(x)  
  ifelse(is.character(x), paste("number of characters: ", nchar(x))  
        paste("length of object: ", length(x))))
```

# Dataframe: Definition and attributes

A **data.frame** contains several elements of different types with same length. It shares properties of **matrix** and **list**. It can be created by hand with the *data.frame()* function.

```
age <- c(20, 21, 20, 25, 29, 22)
taille <- c(165, 155, 150, 170, 175, 180)
sexe <- c("F", "F", "F", "M", "M", "M")
don <- data.frame(age, taille, sexe, stringsAsFactors = T)
```

We can get the attributes of the **data.frame** with functions *dim()*, *nrow()*, *ncol()*, *colnames()*, *row.names()*.

```
dim(don)
nrow(don)
ncol(don)
colnames(don)
row.names(don)
don <- don[sample(nrow(don)), ]
row.names(don)
```

# Dataframe: Manipulation

To extract part of a **data.frame**, it is like a **list**. Be careful to distinguish the row names with the number of rows.

```
don[c("1", "3", "5"), ]  
don[c(1, 3, 5), ]  
don[, c(2, 3)]  
don[, c("age", "sexe")]  
don[c("3", "5"), c(3, 5)]  
don[don$sexe == "F", ]  
don[which(don$sexe == "F"), ]
```

Function *subset()* allows to make a selection by doing conditions. User can call directly the variable names without calling the **data.frame**

```
subset(don, sexe == "F")
```

# Dataframe: row names and useful functions

In the **tidyverse** universe, it is recommended to store the ID's of the observations in one specific variable:

```
don$id <- c("sonia", "maud", "iris", "mathieu", "amin", "gregory")  
row.names(don) <- NULL
```

Here are some useful functions for **data.frame**:

```
head(don, 3)  
tail(don, 4)  
str(don)  
summary(don)  
res_lm <- lm(taille ~ sexe - 1, data = don)  
summary(res_lm)
```



# Dataframe: what is a factor?

In **R**, factors are used to work with categorical variables, variables that have a fixed and known set of possible values

```
class(don$sexe)
levels(don$sexe)
don$sexe[1] <- "others"
```

To add a new level, it must be first specified

```
don$sexe <- factor(don$sexe, levels = c("F", "M", "others"))
don$sexe[1] <- "others"
```

A factor can order the levels which can be useful when plotting the distribution:

```
don$note <- factor(c("AB", "B", "B", "P", "TB", "TB"),
                  levels = c("P", "AB", "B", "TB"), ordered = T)
barplot(table(don$note))
```

# Dataframe: Modify

Use `[]`, `$`, or `cbind()` to add a column with respect to the match of dimensions (there is a verification strict)

```
don$diplome <- c("DU", "M2", "M2", "DU", "DU", "M2")
don[, "diplome"] <- c("DU", "M2", "M2", "DU", "DU", "M2")
don <- cbind(don, pays = c("FR", "FR", "SG", "CA", "HA", "BF"),
            stringsAsFactors = F)
```

Use `rbind()` to add a row (with respect to the match of dimensions):

```
don <- rbind(don, list(21, 180, "F", "isa", "DU", "FR"))
```

Concatenate two dataframes (with respect to the match of names):

```
don2 <- data.frame(age = c(20, 21), taille = c(180, 175),
                  sexe = c("M", "F"), id = c("pierre", "sonia"),
                  diplome = c("DU", "DU"),
                  pays = c("FR", "ESP"))
don <- rbind(don, don2)
```

# Dataframe: Merge

```
note_1 <- data.frame(note_al = c(18, 10, 8, 15), id = c("101", "152", "103", "104"))
note_2 <- data.frame(note_st = c(5, 15, 20, 10), id = c("102", "120", "103", "104"))
```

To merge two data frame by using **R** base functions, we can do:

```
id_inter <- intersect(note_1$id, note_2$id)
data.frame(id = id_inter,
           note_al = note_1[match(id_inter, note_1$id), !names(note_1) %in% "id"],
           note_st = note_2[match(id_inter, note_2$id), !names(note_2) %in% "id"])
```

Function *merge()* allows to do other versions of database join operations:

```
merge(note_1, note_2, by.x = "id", by.y = "id")
merge(don, don3, by.x = "id", by.y = "nom", all.x = T)
merge(don, don3, by.x = "id", by.y = "nom", all.y = T)
merge(don, don3, by.x = "id", by.y = "nom", all = T)
merge(don, don3)
```

# Dataframe: split/aggregate

```
don4 <- data.frame(note = c(18, 10, 8, 15, 20, 5, 17, 12, 8),  
                   semestre = c("s1", "s1", "s1", "s1", "s2", "s2",  
                                "s2", "s2", "s2"),  
                   matiere = c("al", "st", "st", "st", "st", "st",  
                               "al", "al", "al"))
```

Function *split()* allows to split a **vector** or a **data.frame** with respect to a (list of) categorical variable. The result is a list which allows to apply function *sapply()* on it. Function *tapply()* does the two steps split/sapply in the same function and it also does a merge step to get a *data.frame*.

```
my_split <- split(x = don4$note, f = don4[2:3])  
sapply(my_split, mean)  
tapply(X = don4$note, INDEX = don4[2:3], FUN = mean)
```

Function *aggregate()* can do the same thing for more than one variable. It is possible to use the **formula** syntax:

```
aggregate(note ~ semestre + matiere, data = don4,  
          FUN = mean, na.rm = T)
```



## Training: Exercise 2.4

- Execute the solution of exercise 2.3 and create a **list** object called **res\_lm** which contains the residuals, the SSE value and the  $R^2$ .
- Create a **data.frame** called **pred\_y** which contains the fitted values, the residuals and the  $Y$  variable.



## 2. Basic data manipulation

d. Examples of other useful classes of object

# Date with R

Reference of this section: see the [R Task View](#)

- There exists a **Date** class of object to work with daily data.

```
(format.Date <- Sys.Date())  
class(format.Date)  
dates <- c("01/01/17", "02/03/17", "03/05/17")  
as.Date(dates, "%d/%m/%y")  
dates <- c("1 janvier 2017", "2 mars 2017", "3 mai 2017")  
as.Date(dates, "%d %B %Y")
```

- The **POSIXct/POSIXt** class of object contains the date + the time:

```
dates <- c("02/27/92", "02/27/92", "01/14/92",  
          "02/28/92", "02/01/92")  
times <- c("23:03:20", "22:29:56", "01:03:30",  
          "18:21:03", "16:56:26")  
x <- paste(dates, times)  
strptime(x, "%m/%d/%y %H:%M:%S")
```

# POSIXct/POSIXt

There exists many functions which deal with this class of object:

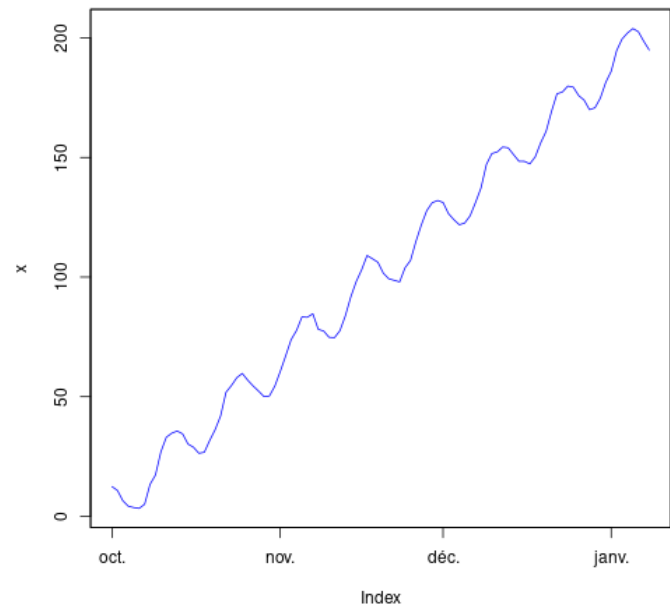
```
(format.POSIXlt <- Sys.time())  
class(format.POSIXlt)  
weekdays(format.POSIXlt)  
months(format.POSIXlt)  
quarters(format.POSIXlt)
```

The **zoo** package allows to associate a vector of values with a **Date** or **POSIXct/POSIXt**. a Create time object:

```
date_x <- seq.Date(as.Date("201  
require("zoo")  
y_t <- 1 + 2 * (1:100) + 10 * c  
x <- zoo(y_t, date_x)
```

To plot the series, the function *plot.zoo()* allows to represent a serie efficiently:

```
plot(x, col = c("blue", "red"))
```





# Analysis of a time serie

- Simulate Time series

```
set.seed(493)
x1 <- arima.sim(model = list(ar = c(.9, -.2)), n = 100)
```

- To study the autocorrelations of a process, use *acf()* :

```
acf(x1)
```

- To plot a lag-plot, use *lag.plot()*

```
lag.plot(x1)
```

# Spatial data

For a full presentation of spatial data with **R**, see my course [here](#).

```
link <- "http://www.thibault.laurent.free.fr/cours/R_intro/Ressources"
```

In this example, we create a spatial object **sp**, we define the Coordinate Referential System (CRS) and plot the data on the map

```
require("sp")
seisme_df <- read.csv2(paste0(link, "seisme.csv"))
seisme <- seisme_df
coordinates(seisme) <- ~ Longitude + Latitude
proj4string(seisme) <- CRS("+proj=longlat +ellps=WGS84 +datum=WGS84 +units=m +no_defs")
```

To plot the map:

```
plot(seisme, cex = sqrt(seisme$mag))
```





## 3. Importing data

### a. Usual file formats

# Checklist

When using the **data.frame** class of object to import data, try to check the following conditions:

- With spreadsheets, first row is usually reserved for the header, while the first column is used to identify units;
- Avoid names, values or fields with blank spaces, otherwise each word will be interpreted as a separate variable, resulting in errors that are related to the number of elements per line;
- Short names are preferred over longer names;
- Try to avoid using names that contain symbols such as ?, \$, %, ^, &, \*, (, ), -, #, ?, , , <, >, /, |
- Delete any comments that you have made in your Excel file to avoid extra columns or NA's to be added to your file; and
- Make sure that any missing values in your data set are indicated with NA.

When using **tibble** class of object, some of these constraints are less important

# Importing data from text data

```
link <- "http://www.thibault.laurent.free.fr/cours/R_intro/Ressourc
```

- Function *readLines()* can be used to read only few rows, to have an overview of the data file structure:

```
readLines(con = paste0(link, "dontxt_correct.txt"), n = 2)
```

- Function *read.table()* is used to import all kind of text files such that ".txt" or ".csv". It contains many arguments to fill:

```
don.txt <- read.table(file = paste0(link, "dontxt_correct.txt"),  
  header = TRUE,  
  sep = ",", dec = ".", na.strings = "NA",  
  nrow = -1,  
  skip = 0,  
  stringsAsFactors = default.stringsAsFactors(),  
  fileEncoding = "",  
  encoding = "unknown")  
head(don.txt)
```

# Importing data from classical formats

- Functions `read.csv()` and `read.csv2()` are wrappers of `read.table` to import "csv" files:

```
don.csv <- read.csv2(file = paste0(link, "communes-de-toulouse-metro")  
head(don.csv)
```

- Function `fromJSON()` from **jsonlite** package allows to import "json" format:

```
elec <- "https://www.data.gouv.fr/fr/datasets/r/cae2bd1b-e682-4866-4  
don.json <- jsonlite::fromJSON(elec)
```

- function `load()` allows to import objects saved in the **R** format (".RData" file):

```
save(don, don3, file = "data_exo.RData")  
load("data_exo.RData")
```

# Importing data from statistical softwares

- Package **readxl** is part of the **Tidyverse project**. Function `read_xls()` allows to import data in the Excel format.

```
f <- "https://www.insee.fr/fr/statistiques/fichier/3292622/dep31.xls"
download.file(f, destfile = paste0(getwd(), "/dep31.xls"))
don.xls <- readxl::read_xls("dep31.xls", skip = 7)
```

The result is an object of class **tibble** (see [this chapter book](#) for more informations). It is very similar to a **data.frame** object but with less constraints (for example, only few rows of code are printing, the name of the variables can be anything, etc.).

- Package **sas7bdat** allows to import data saved with SAS (".sas7bdat" file):

```
don.sas <- sas7bdat::read.sas7bdat(paste0(link, "baseball.sas7bdat"))
```

- **foreign** package allows to import data saved in the Stata format (".dta" file):

```
automiss <- foreign::read.dta(paste0(link, "automiss.dta"))
```

# Importing data from the Web

- XML packages allows to import data stored in ".xml" file

```
library("XML")
don.xml <- xmlParse(paste0(link, "input.xml"))
rootnode <- xmlRoot(don.xml)
print(rootnode[1])
rootsize <- xmlSize(rootnode)
xmldataframe <- xmlToDataFrame(paste0(link, "input.xml"))
print(xmldataframe)
```

- Package **rtweet** allows to import data from Twitter (need a twitter account). See [the vignette](#) for more informations.





# 3. Importing data

## b. Big data files

# Create a big file of data and export it

We consider a first type of data: big but not big enough for not being imported with **R**. We create such a data set with 500,000 rows and 3 columns:

```
n <- 500000 # to modify
data_to_import <- data.frame(chiffre = 1:n,
  lettre = paste0("caract", 1:n),
  date = sample(seq.Date(as.Date("2017-10-01"), by = "day",
    len = 100), n, replace = T))
```

Function *object.size()* gives the memory used to store an **R** object. Here, our data consumes 40Mo of RAM (most of the recent machines have at least 4Go of RAM).

```
object.size(data_to_import)
```

Finally, we export the object as a "txt" file. It occupies 16Mo disk memory:

```
write.table(data_to_import, "fichier.txt", row.names = F)
file.info("fichier.txt")
```

# Import a big file (1)

Function `system.time()` returns the computational time.

- Solution 1: When using **stringsAsFactors = T**, the categorical variables are coded into a **factor**. It is time consuming, because **R** has to determine first the different possible levels before storing it.

```
system.time(import1 <- read.table("fichier.txt", header = T,  
                                stringsAsFactors = T))
```

- Solution 2: After **useR!2019** conference in Toulouse, the **R** core decided to save the strings as **character**. The reason used is not the computational time, but the non reproducible issue due to the characters sorting (for more informations, see [this post](#)).

```
system.time(import2 <- read.table("fichier.txt", header = T))
```

# Import a big file (2)

- Solution 3: identify first the types of the variables

```
bigfile_sample <- read.table("fichier.txt", header = T, nrows = 20)  
(bigfile_colclass <- sapply(bigfile_sample, class))
```

Then, import the data by specifying the types. It avoids to read all the data before defining the types.

```
system.time(bigfile_raw <- read.table("fichier.txt", header = T,  
                                     colClasses = bigfile_colclass))
```

# Use dedicated packages

- The **Tidyverse project** includes the package **readr** which allows to optimize the importation of the data.

```
system.time(  
  tibble.don <- readr::read_table2("fichier.txt")  
object.size(tibble.don)  
class(tibble.don)
```

- Package **data.table** is an alternative which use another class of object (**data.table**) with some specific rules of manipulation (see this [web page](#) for more informations).

```
system.time(  
  objet.data.table <- data.table::fread("fichier.txt")  
class(objet.data.table)
```

# What is big data?

When does data become very big? With **R**, data is stored in RAM. Data becomes too big when the machine has not enough RAM for storing it.

What can I do?

- Make the data smaller (sampling)
- Get a bigger computer
- Split up the dataset for analysis (Map/Reduce)
- Access the data differently (**ff** package or interact with Database Management System)

References:

- [http://www.columbia.edu/~sjm2186/EPIC\\_R/EPIC\\_R\\_BigData.pdf](http://www.columbia.edu/~sjm2186/EPIC_R/EPIC_R_BigData.pdf)
- [https://rpubs.com/msundar/large\\_data\\_analysis](https://rpubs.com/msundar/large_data_analysis)

# Example of big data

Create a big data (around 50 millions of rows):

```
readr::write_csv(data_to_import, "big_file.csv")
require(dplyr)
p <- progress_estimated(100)
for(k in 1:100){
  p$pause(0.1)$tick()$print()
  readr::write_csv(data_to_import, "big_file.csv", append = T)
}
```

With this example, the **data.frame** is around 1.2Go (it corresponds to 1.5Go on disk).

```
my_big_data <- readr::read_csv("big_file.csv")
object.size(my_big_data)
file.info("big_file.csv")
```

Here, many operations (extraction, statistics, etc.) can be done even if it becomes costly in computational time.

However, let suppose that this dataset could not to be loaded in **R**.

# Map/Reduce algorithm

*Map/Reduce* algorithm can be used for avoiding to import all the data set at the same time. It consists in splitting the data and doing a job in each data (Map step). The reduce step consists in assembling the results. 🤔 Parallel computing can be used to get higher performance

```
n_split <- 11
ind <- 1
n_max <- 5000001
my_max <- numeric(n_split)
my_mean <- numeric(n_split)
my_n <- numeric(n_split)
for (k in 1:n_split) {
  split_don <- read_csv("big_file.csv", skip = ind, n_max = n_max,
                       col_names = c("chiffre", "lettre", "date"),
                       col_types = "ncc")

  my_max[k] <- max(split_don$chiffre)
  my_mean[k] <- sum(split_don$chiffre)
  my_n[k] <- nrow(split_don)
  ind <- ind + n_max
}
sum(my_mean) / sum(my_n)
max(my_max)
```



# Access the data differently

- package **ff** uses hard disk to store the native binary flat files rather than its memory (see [this tutorial](#) for more informations). Many algorithms for base statistics methods (min, max, sum, mean, etc) can then be applied with **ffbase**.

```
require("ff")
bigDF <- read.csv.ffdf(file="big_file.csv", header = TRUE,
                      first.rows = 500000, next.rows = 5000000)
basename(filename(bigDF$date))

object.size(bigDF)
bigDF[25000000:25000003, ]
tail(bigDF)
library("ffbase")
mean.ff(bigDF$chiffre)
```

- Alternative: package **disk.frame**(see [Githun](#)).
- Interact with Database Management System (DBMS) by using R packages **RODBC**, **RMySQL**, **RPostgreSQL**, **RSQLite** or **mongolite/couchDB**.



# Training

## Exercise 3.1

- Import one data set from these different web pages by using the method of your choice:
  - [link 1](#)
  - [link 2](#) (import if possible a ".xls" file)
  - [link 3](#)



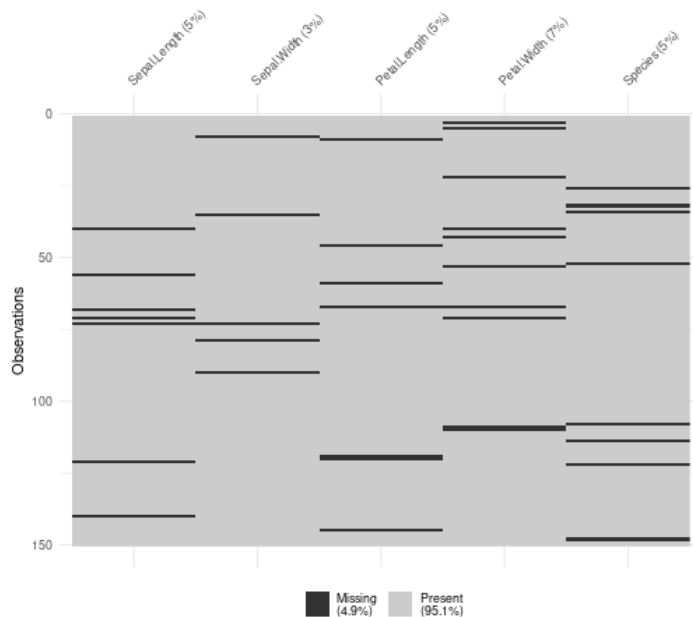
# 4. Data cleaning

## a. Missing values

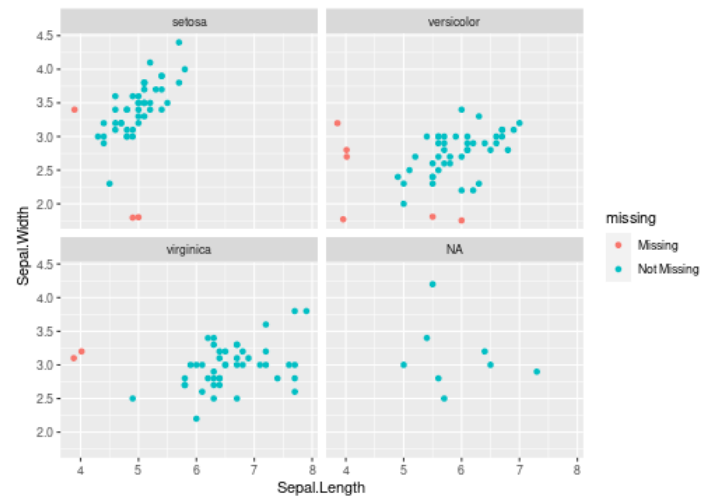
# Visualizing missing values

R packages **visdat** and **naniar** include many different tools for visualizing missing values (see [vignette](#) for more information).

```
iris.mis <- missForest::prodNA(  
  visdat::vis_miss(iris.mis)
```



```
ggplot(iris.mis, aes(x = Sepal.  
  naniar::geom_miss_point() +  
  facet_wrap(~Species)
```



# How to deal with NA?

The reference of this section can be found [here](#).

- Doing nothing:

```
res_lm <- lm(Sepal.Width ~ Sepal.Length + Petal.Length +  
            Petal.Width + Species, data = iris.mis)
```

- Deletion

```
iris.mis <- subset(iris.mis, !is.na(Species))
```

- Mean/ Mode/ Median Imputation

```
ind_NA_Sepal.Length <- is.na(iris.mis$Sepal.Length)  
mean_spec <- aggregate(Sepal.Length ~ Species,  
                       data = iris.mis, FUN = mean, na.rm = T)  
for (k in levels(iris.mis$Species)) {  
  iris.mis[ind_NA_Sepal.Length & iris.mis$Species == k,  
           "Sepal.Length"] <- mean_spec[mean_spec$Species == k, 2]  
}
```

# Examples of treatment

- Prediction Model

```
iris.imp_mean <- data.frame(sapply(iris.mis[, 1:4],  
  function(x) ifelse(!is.na(x), x, mean(x, na.rm = T))),  
  Species = iris.mis$Species)  
pred <- predict.lm(res_lm, newdata = iris.imp_mean)  
iris.mis[is.na(iris.mis$Sepal.Width), "Sepal.Width"] <-  
  pred[is.na(iris.mis$Sepal.Width)]
```

- Dedicated packages: **MICE**, **Amelia**, **missForest**

```
require("missForest")  
iris.imp <- missForest(iris.mis)
```

- KNN Imputation

```
require("DMwR")  
iris.knn <- knnImputation(iris.mis, k = 2)
```



## 4. Data cleaning

### b. Tidyverse universe: data management

# dplyr

**dplyr** package belongs to the **tidyverse** universe. It allows to use another approach to manipulate data instead of using the **R** base syntax.

```
require("tidyverse")  
data("diamonds")
```

We're going to learn some of the most common **dplyr** functions:

- *select()*: subset columns
- *filter()*: subset rows on conditions
- *mutate()*: create new columns by using information from other columns
- *group\_by()* and *summarize()*: create summary statistics on grouped data
- *arrange()*: sort results
- *count()*: count discrete values



# dplyr: Filtering

- *filter()* allows to subset rows using column values.

**Example:** instead of using the following **R** base code to subset the data:

```
filt <- diamonds[diamonds$price > 15000 & (diamonds$color == "E" |
```

It can be done like this with the **tidyverse** syntax:

```
filt <- filter(diamonds, price > 15000 & (color == "E" | color == "F")  
filt <- filter(diamonds, price > 15000, (color == "E" | color == "F"))
```

**Remark:** the & operator can be replaced by the coma.

# dplyr: select

- Function *select()* allows to extract variables. Instead of doing:

```
select1 <- diamonds[, c("carat", "price", "color", "y")]
```

We can do:

```
select1 <- select(diamonds, carat, price, color, y)
```

- There are some shortcuts to make characters matching. For example, if we want to select the variables that contain character "y" in the names, it can be done like this with **R** base syntax:

```
select1 <- diamonds[, names(diamonds)[grep("y", names(diamonds))]]
```

- function *contains()* allows to do the same thing:

```
select1 <- select(diamonds, contains("y"))
```

# dplyr: rename, mutate

- Function *rename()* allows to change the names of columns. Instead of doing:

```
names(diamonds)[c(match("y", names(diamonds)),  
                  match("x", names(diamonds)))] <- c("width", "length")
```

It can be done like this:

```
renom1 <- rename(diamonds, width = y, length = x)
```

- Function *mutate()* allows to create new variables. Instead of doing:

```
diamonds$prix.kilo <- diamonds$price/diamonds$carat  
diamonds$prix.kilo.euro <- diamonds$prix.kilo * 0.9035
```

It can be done like this:

```
calcul1 <- mutate(diamonds, prix.kilo = price/carats,  
                  prix.kilo.euro = prix.kilo * 0.9035)
```

# Pipeline operator on vectors

The pipeline operator `%>%` consists in calling an **R** object on the left on which we want to apply a function on the right. For example to compute the mean of a vector:

```
x <- c(10, 8, 5, 12, 9, 12)
x %>% mean()
```

```
## [1] 9.333333
```

It is possible to execute successive pipeline operator:

```
x <- c(10, 8, 5, 12, 9, 12, NA, - 5)
x %>%
  replace(list = which(x < 0), NA) %>%
  na.omit() %>%
  mean()
```

```
## [1] 9.333333
```

# Pipeline operator on `data.frame`

It can be interesting to use pipeline operator on `data.frame` because it allows to make successive operations in one command and leads the different operations understandable. For example, to compute the mean of variables `price` and `carat` after filtering the data, one can do in **R** base code:

```
sapply(diamonds[diamonds$price > 15000 &
               (diamonds$color == "E" | diamonds$color == "F"),
        c("price", "carat")],
       FUN = mean)
```

It can be done like this with the pipeline syntax:

```
diamonds %>%
  filter(price > 15000, (color == "E" | color == "F")) %>%
  select(price, carat) %>%
  sapply(FUN = mean)
```

# dplyr: ordering

function `arrange()` allows to order data with respect to one variable. Instead of doing with R base code:

```
diamonds[order(diamonds$cut, diamonds$color, -diamonds$price),]
```

One can do by using the tidyverse universe:

```
arrange(diamonds, cut, color, desc(price)) %>%  
  head(n = 3)
```

```
## # A tibble: 3 × 10  
##   carat cut    color clarity depth table price      x      y      z  
##   <dbl> <ord> <ord> <ord>    <dbl> <dbl> <int> <dbl> <dbl> <dbl>  
## 1  2.02 Fair  D      SI1      65      55 16386  7.94  7.84  5.13  
## 2  2.01 Fair  D      SI2      66.9     57 16086  7.87  7.76  5.23  
## 3  3.4  Fair  D      I1      66.8     52 15964  9.42  9.34  6.27
```

# dplyr: count observation by group

To count the number of observations by group, instead of doing the following codes by using base **R** code (the result is a vector):

```
sapply(split(diamonds, diamonds$color), FUN = nrow)
```

```
##      D      E      F      G      H      I      J
## 6775 9797 9542 11292 8304 5422 2808
```

One can do (the result is a **data.frame**):

```
diamonds %>%
  count(color)
```

```
## # A tibble: 7 × 2
##   color      n
##   <ord> <int>
## 1 D      6775
## 2 E      9797
## 3 F      9542
## 4 G     11292
## 5 H      8304
## 6 I      5422
```

# dplyr: summarize

- Function *summarize()* allows to compute statistics like the mean, median, etc. for each combination of grouping variables. Instead of doing the following codes by using **R** base code:

```
calcul_a <- aggregate(formula = price ~ cut + color, data = diamonds,  
  c(mean = mean(x), n = length(x)))
```

One can do with the **tidyverse** syntax:

```
calcul2 <- summarise(group_by(diamonds, cut, color),  
  size = n())
```

**Remark:** Note that function *n()* gives the current group size



# dplyr: summarize

We can summarize all the commands executed by using the pipeline operator:

```
diamonds %>%  
  filter(price > 15000, (color == "E" | color == "F")) %>%  
  mutate(prix.kilo = price/carat,  
         prix.kilo.euro = prix.kilo * 0.9035) %>%  
  group_by(cut, color) %>%  
  summarize(eff = n(),  
           prix.moy = mean(prix.kilo))
```



## 4. Data cleaning

b. Tidyverse universe: tidy data with **tidyr**

# tidyr: tidy data (separate)

What is **tidy data**: data sets that are arranged such that each variable is a column and each observation (or case) is a row.

**tidyr** package transform data from *messy* to *tidy* (more informations [here](#))

```
wide_data <- data.frame(country = c("C_1", "C_2"),
                        Y = c("6/4.6"),
                        X1 = c("7.8-0.6"))
```

- Function `separate()` splits a variable into two variables with respect to a **string**

```
wide_data <- wide_data %>%
  separate(col = Y, into = c("Y_2009", "Y_2019"), sep = "/") %>%
  separate(col = X1, into = c("X_2009", "X_2019"), sep = "-")
```

# tidyr: wide to long (1)

The same variable  $Y$  is observed at different time in two columns. It should be included in one unique variable  $Y$ , the year should be another variable and the countries duplicated for each year.

Function `pivot_longer()` (ex `gather()`) vectorizes argument **cols** (here **Y\_2009**, **Y\_2019**, etc.) into argument **values\_to** (here **Y**) and creates a variable **names\_to** (here **year**) with the names of **cols**

From **wide**...

country	Y_2009	Y_2019
C_1	6	4.6
C_2	6	4.6

... to **long**

country	years	Y
C_1	Y_2009	6
C_1	Y_2019	4.6
C_2	Y_2009	6
C_2	Y_2019	4.6

# tidyr: wide to long (2)

Applications to our data set: we must gather  $Y$  then  $X$ , and finally merge the data. All can be done by using the pipeline operator

```
merge(wide_data %>%
  select(country, Y_2009, Y_2019) %>%
  pivot_longer(cols = c("Y_2009", "Y_2019"),
               names_to = "years",
               values_to = "Y") %>%
  mutate(years = substr(years, 3, 6)),
panel_data %>%
  select(country, X_2009, X_2019) %>%
  pivot_longer(cols = c("X_2009", "X_2019"),
               names_to = "years",
               values_to = "X") %>%
  mutate(years = substr(years, 3, 6)),
by = c("country", "years")
)
```

# tidyr: wide to long (3)

Previous codes can be simplified by using regular expressions (this is a special syntax for finding characters).

```
pivot_longer(wide_data,  
             cols = 2:5,  
             names_to = c(".value", "year"),  
             names_pattern = "(.)_(.*)")
```

```
## # A tibble: 4 × 4  
##   country year   Y     X  
##   <chr>   <chr> <chr> <chr>  
## 1 C_1     2009   6     7.8  
## 2 C_1     2019  4.6   0.6  
## 3 C_2     2009   6     7.8  
## 4 C_2     2019  4.6   0.6
```

# tidyr: long to wide

```
long_data <- data.frame(country = c("C_1", "C_2", "C_1", "C_2"),  
                        key = c("X", "X", "Y", "Y"),  
                        value = c(25, 45, 2500, 5500))
```

**value** includes the values of two variables **X** and **Y**. **key** contains the labels **X** and **Y**. We should have one column for **X** and one column for **Y**. Function *pivot\_wider()* (ex *spread()*) spreads the vector with argument **values\_from** (here **value**) with respect to the argument **names\_from** (here **key**).

```
pivot_wider(long_data, names_from = key, values_from = value)
```

From **long**...

country	key	value
C_1	X	25
C_2	X	45
C_1	Y	2500
C_2	Y	5500

... to **wide**

country	X	Y
C_1	25	2500
C_2	45	5500



# Training: Exercise 4.1

- In the data **admnrev** from package **wooldridge**, transform the data from the long to the wide form with respect to the variable year. We call **admnrev\_wide** this object
- Transform the object **admnrev\_wide** from wide to long object.