

Prérequis

A introduction to



Claire Vandiedonck,

Associate Professor

claire.vandiedonck@u-paris.fr

@Cvandiedonck



No conflict of interest to disclose

Un script reprenant l'ensemble du code présenté dans ce diaporama est fourni

Plan des prérequis

- Premiers pas avec R: diapos 1 à 24, 4 vidéos et leur quiz, SWIRL

vidéo1.1

1. ouvrir et quitter R
2. exécution de commandes
3. utilisation d'une fonction built-in de R

vidéo1.2

1. Assigner des objets R
2. Gérer les objets R dans la session R

vidéo1.3

1. Gérer une session R et son répertoire de travail
2. Sauvegarder une session R, les objets et l'historique

vidéo1.4

1. Les lignes de commandes: utiliser un éditeur de texte et lancer un script
2. L'environnement Rstudio

==> tutoriel swirl (diapo 23-24): 5 exercices interactifs

- Les types de variables et d'objets dans R: diapos 25 à 28
- Les vecteurs: diapos 29 à 36 + exercices 1 à 10
- Les matrices: diapos 37 à 47

Premiers pas avec



Claire Vandiedonck - Université de Paris

R – <https://www.r-project.org>



- Open-source
- statistical programming language available for Windows, Mac, Unix
- Widely used in academia, finance, pharma, social sciences...
- Core language, ‘base’ and > 3000 contributed packages
- Objectives:
 1. Data manipulation: import, transform, export
 2. Perform statistics
 3. Generate advanced graphics
- Interactive sessions, scripts, packages in the CRAN (“Comprehensive R Archive Network”)
- Possible interactions with other languages
- Project started in 1993; 12-12-2019: version R.3.6.2; version 4.0.3 (Bunny-Wunnies Freak Out) on 2020-10-10
- Some useful links or documentation:
 - ✓ R for beginners d’E. Paradis (exists in English and in French)
 - ✓ QuickR: <http://www.statmethods.net/index.html>
 - ✓ And mostly the help menu: **help.start()**

Vidéo 1.1. Premiers pas avec



1. ouvrir et quitter R
2. exécution de commandes
3. utilisation d'une fonction built-in de R

 lien vers la vidéo: <https://youtu.be/KebToqxaEts>

 quizz d'autoévaluation: <https://forms.gle/ppYtkBMzJzkXQbYn9>

First contact with R

To start

Click on  in Windows/Mac or type **R** in Unix

prompt **'>'** at the beginning of your command line

To quit: **> q()**

-> you may save your current R session by typing **q("yes")**
or not save your current R session by typing **q("no")**

Interaction with R:

You write a **command** and press « Enter »

R executes the command

R waits for another command

trick: use the ↑ and ↓ arrows to move to previous or next commands

Some very simple examples

Enter the commands (here in red)

R answers (here in blue)

Some simple operations:

```
> 2 + 2
```

```
[1] 4
```

```
# it returns the result
```

```
# an index is shown within '[' ]': if multiple results displayed on different lines, an  
# index at the beginning of each line is given for the first value of the line
```

```
> exp(-2)
```

```
[1] 0.1353353
```

```
> log(100, base = 10)
```

```
[1] 2
```

`exp()` or `log()` are built-in functions

What is a function?

Functions are a set of pre-programmed commands

A function is characterized by its:

- name
 - arguments put within brackets to execute the command
- you enter the required parameters within the brackets
- to know the parameters of the function and their default settings:
 '?' followed by the function name

```
>?exp # a help-window opens  
# equivalent to either one of the following two command lines  
> help(exp)  
> help.search("exp")
```


The help associated to a function

Several sections:

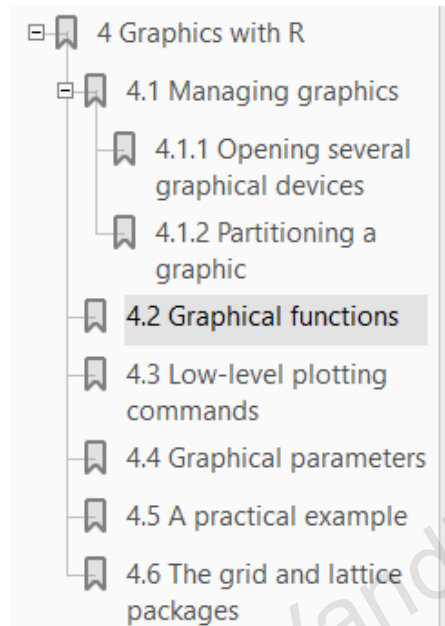
- *Description* -> what is the purpose of the function?
- *Usage* -> how is the function used?
- *Arguments* -> which parameters are used by the function? Defaults values may be specified
- *Details* -> technical description of the function
- *Value* -> what are the output parameters returned by the function?
- *See also* -> are there some similar functions in R?
- *Source/ References* -> not always present...
- *Example* -> concrete examples to use the function

=> the best way to learn how to use the function

Getting help

R for beginners E. Paradis

Chapter 4 for graphs quite exhaustive
in moodle in French and English



R gallery

<http://www.r-graph-gallery.com/all-graphs/>
for specific kinds of graphs

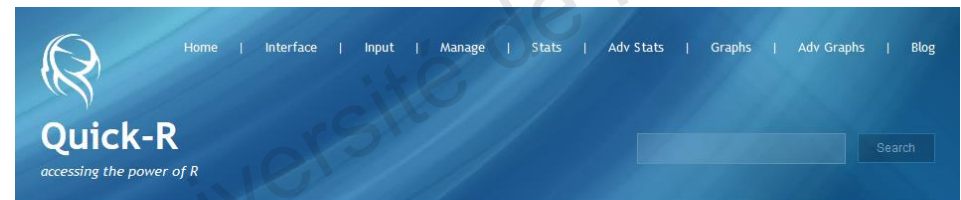
And some blogs for specific questions

https://www.stat.ubc.ca/~jenny/STAT545A/block14_colors.html#using-colors-in-r

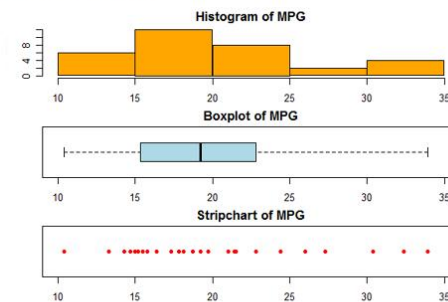
<https://danieljhocking.wordpress.com/2013/03/12/high-resolution-figures-in-r/>

QUICK R:

<http://www.statmethods.net/basic>
and advanced graphs
with main parameters



Basic Graphs



One of the main reasons data analysts turn to R is for its strong graphic capabilities.

Vidéo 1. 2. Premiers pas avec



1. Assigner des objets R
2. Gérer les objets R dans la session R

 lien vers la vidéo: <https://youtu.be/V4Fp0Nmfm3Q>

 quizz d'autoévaluation: <https://forms.gle/vyJ4H7c9bpDAFegr8>

Assigning data into R objects, using and reading them

Use '`<-`' or '`=`' to assign values to R objects

```
> x <- 2      # equivalent to x = 2, assigns 2 to the variable x
> y <- x + 3
> s <- " this is a string of characters"
```

Using/reading values

```
> x          # you call the object
[1] 2        # its value is returned....note that an index is written between [ ]

> y
[1] 5

> s
[1] "this is a string of characters"

> x + x      # computes the operation knowing the value of x
[1] 4

> x^y        # x to the power y
[1] 32

> x <- 4     # change the value of x
> y
[1] 5        # y not dynamically changed!!!!

> y <- x + 3  # need to reattribute y value to update for the new value x
> y
[1] 7
```

Managing objects in your R session

List all objects present in the memory with the function `ls()`

```
> ls()  
[1] "s" "x" "y"
```

Delete an object with the function `rm()`

```
> rm(y) # pour déléter y seulement  
> ls()  
[1] "s" "x"  
> rm(list=ls()) # pour tout déléter  
> ls()
```

Vidéo 1. 3. Premiers pas avec



1. Gérer une session R et son répertoire de travail
2. Sauvegarder une session R, les objets et l'historique

 lien vers la vidéo: <https://youtu.be/7S00g10me5A>

 quizz d'autoévaluation: <https://forms.gle/TL67MHavM3YDzEis8>

Managing your session and working directory

Which R version are you using?

Getting your working directory

Setting a working directory

```
> sessionInfo() # returns R version as well as the  
                  version of loaded packages
```

```
> getwd()
```

```
> setwd() # indicate the path inside the () and  
           flanked by simple ' or double quotes"
```

trick: if ignoring the path, slide a text file from the directory where you want to work, there will be an error message but you will see the path of your directory

Creating a folder in your working directory

Listing all files and folders in your working directory

```
> list.files ()      # returns all files and folders within the working directory
                    # add pattern= ".txt" if you want to list only the .txt files
                    # like ls in Unix
```

Creating a new folder

```
> dir.create ("myfolder")  # creates a folder named "myfolder" in my working directory
                          # like mkdir in Unix

> list.files ("myfolder")
[1] "myfolder"

> ls ()                  # but it is not in my R session !!!!!
character(0)
```


Connecting to a new file in your working directory (wd)

Opening a connection to a new file to write in

```
> zz <- file("mynewfile.txt", open="wt") # opens a new file to write in my working directory
> list.files ()
[1] "myfolder" "mynewfile.txt"
> close(zz)
```

↪ diverting R console outputs with sink()

```
> sink("myRoutputs.Rout") # diverts the console outputs to myRoutputs.Rout within the wd
> 1+1
> is.numeric(x)
> sink() #to close -> open myRoutputs.Rout in a text editor to see how it looks like
```

or using file() before:

```
> zz <- file("myRoutputs2.Rout", open="wt")
> sink(zz) # diverts the console outputs to myRoutputs2.Rout within the wd
> is.numeric(x)
> 1:10
> sink()
> close(zz) #to close -> open myRoutputs2.Rout in a text editor to see how it looks like
```

Saving your session, data and history

Saving one object from your session

```
> save(x, file="x.RData")
```

Saving all objects

```
> save.image(file="AllMyData.RData")
```

Saving your history

```
> savehistory(file="MyHistory.Rhistory") # save all your commands in a text file
```

After closing R and restarting it, load your data in a new session

```
> load("x.RData")           # load only x  
> load("AllMyData.RData")  # load all objects saved from the session
```

Vidéo 1. 4. Premiers pas avec



1. Les lignes de commandes: utiliser un éditeur de texte et lancer un script
2. L'environnement Rstudio

 lien vers la vidéo: https://youtu.be/V-zO-hoi_WM

 quizz d'autoévaluation: <https://forms.gle/WVfik1kJWMyFHap17>

Saving your command lines in a text editor

- On windows:

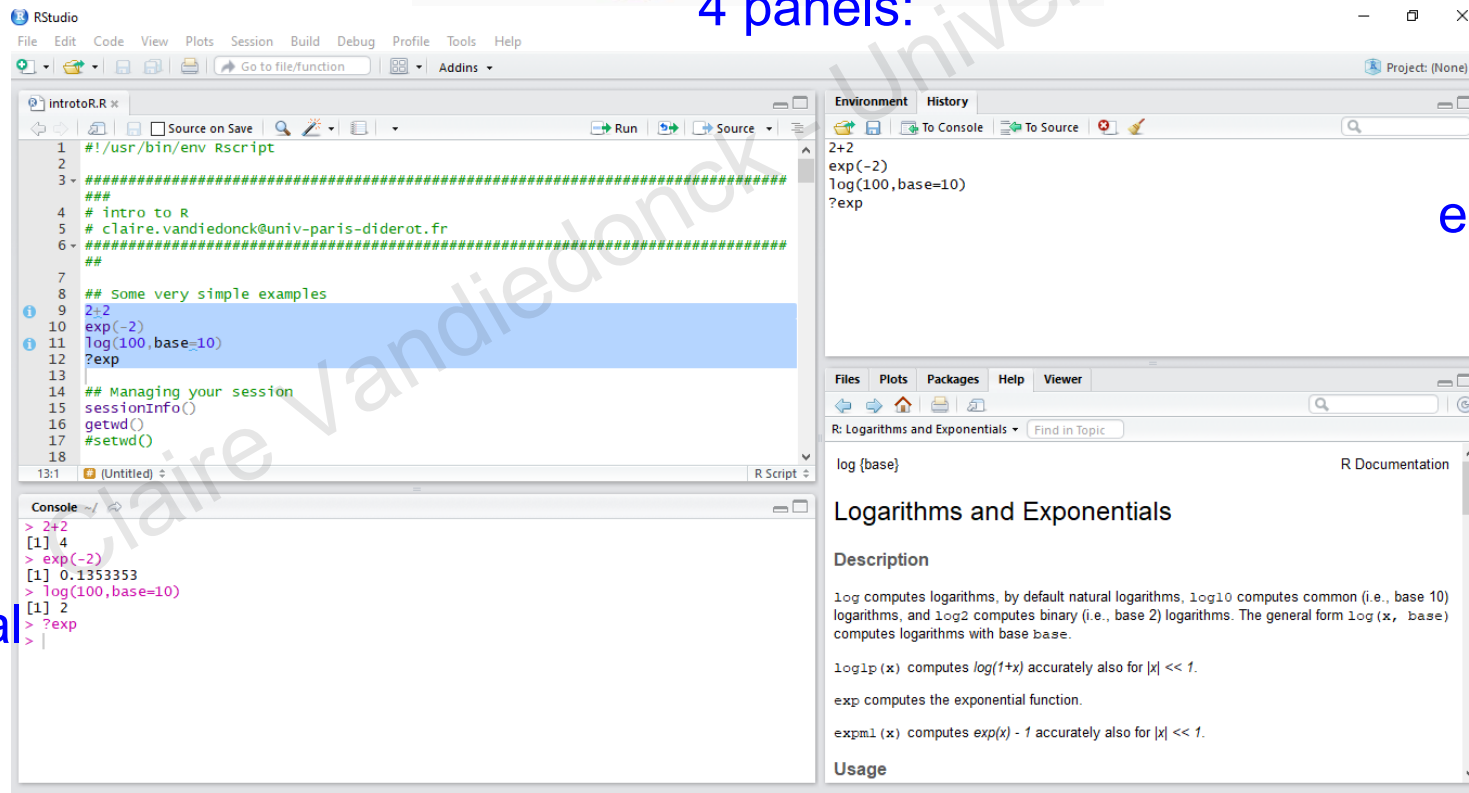
R studio
Notepad++ with NppToR
EMACS
vi
Tinn-R

- On Mac:

R studio
EMACS
Komodo edit

- On linux:

Rstudio
Gedit
Geany
vi
Emacs



script

environment

terminal

figures,
file browser,
help...

R style and language rules

MUST READ:

<https://google.github.io/styleguide/Rguide.xml>

- Avoid any accent, space, special characters
- Comments
 - # Any line starting with "#" will not be read by R
 - # you may report some results as comments
 - # you are highly advised to comment your command lines for your and other's usage
- To name your variables, do not use reserved letters or reserved words already used by R (names of functions or of data class):
eg. c, t, table, data, pi, TRUE, FALSE, T, F, letters, mean, var, ...
=> A good text editor highlight them

Running a set of command lines

Save your commands in a text file: « myscript.R » in your working directory

In R, run automatically all the command lines:

```
> source ("myscript.R")
```

Claire Vandiedonck - Université de Paris

Interactive tutorials



Let's use the « swirl » library

<https://swirlstats.com/students.html>

```
> install.packages("swirl") # this command has to be done only once
> library(swirl) # this command has to be performed at each R session in which you
# want to use swirl
> swirl() # to start swirl
```

Then follow the instructions and practice with tutorials n°1 and 3 to 6 of the "R Programming" course (choice n°1 = "R programming: the basics of R programming" and "1: R programming").

To quit swirl: choose 0 several times or enter `bye()`. At any time during a lesson, the instructions are:

```
| You can exit swirl and return to the R prompt (>) at any time by pressing the Esc key. If you are
| already at the prompt, type bye() to exit and save your progress. When you exit properly, you'll see
| a short message letting you know you've done so.
```

```
| When you are at the R prompt (>):
| -- Typing skip() allows you to skip the current question.
| -- Typing play() lets you experiment with R on your own; swirl will ignore what you do...
| -- UNTIL you type nxt() which will regain swirl's attention.
| -- Typing bye() causes swirl to exit. Your progress will be saved.
| -- Typing main() returns you to swirl's main menu.
| -- Typing info() displays these options again.
```

If it fails finding the "1: R programming" course

1. you may try to install the course with the following command:

```
> swirl::install_course("R_Programming")
```

2. The swirl website may also be temporary down and the "1: R programming" not accessible. If it happens, you can install the " R_Programming.swc " course manually from an archive repository:

https://web.archive.org/web/20200219034901/http://swirlstats.com/scn/R_Programming.swc size (127 Ko)

Then, in R or Rstudio, install the course with this command:

```
> swirl::install_course()
```

and with Rstudio, select the "R_Programming.swc" file. If in R without Rstudio, enter the file name with its path.

You are now ready to start swirl and the R programming course (cf. previous slide)

Les types de variables et d'objets



Claire Vandiedonck - Université de Paris

Classes of R Objects

Main variable types

numeric / integer

character

logical (FALSE / TRUE / NA)

complex

time, time series

factors

↪ `mode()` or `typeof()` returns the type of the object

↪ `class()` returns the class of the object

↪ `is.logical()` tells us if the object is a logical type.

There is also `is.character()`, `is.numeric()`, `is.integer()`,
`is.null()`, `is.na()`

↪ `as.character()`, `as.numeric()`...to coerce objects
from one type to another

```
> x <- c(3,7,1,2)
> x<2
[1] FALSE FALSE TRUE FALSE
> x==2
[1] FALSE FALSE FALSE TRUE

> mode(x) # idem as class(x)
[1] "numeric"
> mode(s) # idem as class(s)
[1] "character"

> as.numeric( x < 2 )
[1] 0 0 1 0
```

Special values : NA

Not Available = missing data

NaN

Not a Number = computation is not possible

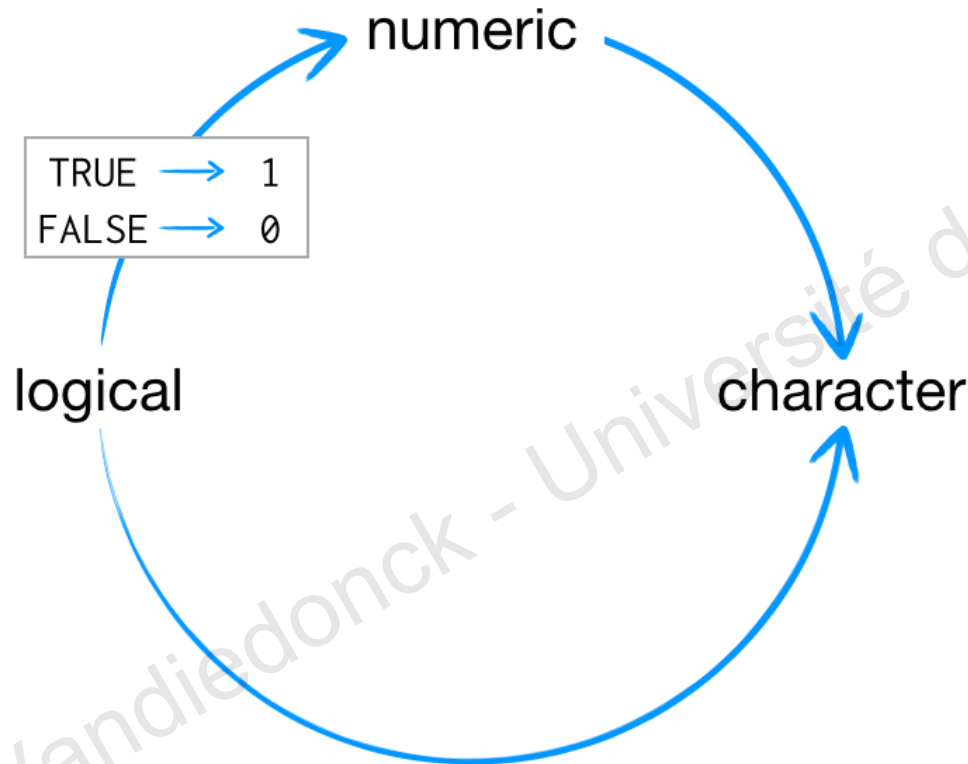
-Inf/Inf

positive and negative infinities

NULL

the value does not exist (rather than being unknown)

Coercion rules



- if character strings are present, everything will be coerced to a character string.
- otherwise logical values are coerced to numbers: TRUE is converted to 1, FALSE to 0
- values are converted to the simplest type required to represent all information
- object attributes are dropped
- the ordering is roughly:

logical < integer < numeric < complex < character < list

Classes of R Objects

Main data structures

object	Heterogeneous = several types may coexist
vector	no
matrix	no
dataframe	yes
list	yes

Les vecteurs



Claire Vandiedonck - Université de Paris

Vector

The most elementary R object

Some functions to create vectors: `c()`, `seq()`, `x:y`, `rep()`, `append()`...

```
> a <- c()      # creates an empty vector that can be further filled
```

```
> a
```

```
NULL
```

```
> weight <- c(60, 72, 57, 90, 95, 72) #c() stands for concatenate
```

```
> weight
```

```
[1] 60 72 57 90 95 72
```

```
> 4:10
```

```
[1] 4 5 6 7 8 9 10
```

```
> seq(4,10)    # returns all numeric values from 4 to 10 (with an increment of 1 =default)
```

```
[1] 4 5 6 7 8 9 10
```

```
> seq(2,10,2)  # returns all numeric values from 1 to 10 with an increment of 2
```

```
[1] 2 4 6 8 10
```

```
> rep(4, 2)    # repeats 4 twice
```

```
[1] 4 4
```

Vector

You may combine functions which are read from inside to the outside:

```
> rep(seq(4,10),2)
[1] 4 5 6 7 8 9 10 4 5 6 7 8 9 10
> c(rep(1,4), rep(2,4))
[1] 1 1 1 1 2 2 2 2
```

A vector is homogeneous

```
> c(5,s)
[1] "5" " this is a string of characters" #5 is not read as an integer
> mode(c(5,s)) # it is converted/coerced into a character with ""
[1] "character"
> class(c(5,s))
[1] "character"
```

To get its size and data type, use functions `length()` and `str()` (for structure)

```
> length(1:10) # the vector is of size 10
[1] 10
> length(weight) # the vector is of size 6
[1] 6
> str(weight) # the vector is of size 6 with numeric values
num [1:6] 60 72 57 90 95 72
```

Vector

You may apply arithmetic operators on numeric values in vectors:

- + to add
- to subtract
- * to multiply
- / to divide
- ^ to raise to the power (or **)
- %% to estimate the remainder of a division (modulo)

```
>size <- c(1.75, 1.8, 1.65, 1.9, 1.74, 1.91)
```

```
>size^2
```

```
[1] 3.0625 3.2400 2.7225 3.6100 3.0276 3.6481
```

```
>bmi <- weight/size^2 # creates a vector with the computed body mass index
```

```
>bmi
```

```
[1] 19.59184 22.22222 20.93664 24.93075 31.37799 19.73630
```


Vector

You may apply pre-computed functions, in particular some descriptive stat functions

```
> size
```

```
[1] 1.75 1.80 1.65 1.90 1.74 1.91
```

```
> sort(size)
```

sorts the data

```
[1] 1.65 1.74 1.75 1.80 1.90 1.91
```

```
> mean(size)
```

```
[1] 1.791667
```

```
> sd(size)
```

returns the standard deviation

```
[1] 0.1002829
```

```
> median(size)
```

```
[1] 1.775
```

```
> min(size)
```

```
[1] 1.65
```

```
> max(size)
```

```
[1] 1.91
```

```
> range(size)
```

```
[1] 1.65 1.91
```

```
> summary(size)
```

```
Min. 1st Qu. Median Mean 3rd Qu. Max.
```

```
1.650 1.742 1.775 1.792 1.875 1.910
```

Vector

Retrieving a value within a vector using its index (**R is 1-based**)

```
> size
```

```
[1] 1.75 1.80 1.65 1.90 1.74 1.91
```

```
> size[1] # returns the 1st value within vector "size"
```

```
[1] 1.75
```

```
> size[2] # returns the 2nd value within vector "size"
```

```
[1] 1.8
```

```
> size[6] # returns the 6th value within vector "size"
```

```
[1] 1.91
```

```
> size[c(2,6)] # returns the 2nd and the 6th value within vector "size"
```

```
[1] 1.80 1.91
```

```
> size[c(6,2)] # returns the 6th and the 2nd value within vector "size"
```

```
[1] 1.91 1.80
```

```
> min(size[c(6,2)]) # returns the min between the 6th and the 2nd value
```

```
[1] 1.80
```

Vector

Attributing names to values of a vector after or at the creation of the vector

```
> names(size)
```

```
[1] NULL           # there are currently no names
```

```
> names(size) <- c("Fabien", "Pierre", "Sandrine", "Claire", "Bruno", "Delphine")
```

```
# names the values of the vector "size"
```

```
> size
```

```
Fabien  Pierre Sandrine  Claire  Bruno Delphine  
  1.75   1.80   1.65   1.90   1.74   1.91
```

```
> str(size)
```

```
Named num [1:6] 1.75 1.8 1.65 1.9 1.74 1.91
```

```
- attr(*, "names")= chr [1:6] "Fabien" "Pierre" "Sandrine" "Claire" ...
```

```
> my_vector <- c("one"=1, "two"=2, "three"=3) # or giving the names at the vector creation
```

```
> my_vector
```

```
one two three
```

```
1 2 3
```

```
> str(my_vector)
```

```
Named num [1:3] 1 2 3
```

```
- attr(*, "names")= chr [1:3] "one" "two" "three"
```

Summary on vectors

Format	one-dimension
Datatype	homogeneous: only one type of character, numeric, logical, factor.. -> coercion if heterogeneous <ul style="list-style-type: none">- check with <code>class()</code> or <code>mode()</code>- checking type with <code>is.num()</code>, <code>is.character()</code>, ...- conversion with <code>as.num()</code>, <code>as.character()</code>, ...
Creation	<code>c()</code> , <code>:</code> , <code>seq()</code> , <code>rep()</code> , <code>sample()</code> , <code>rnorm()</code> , ...
Adding new items	<code>c()</code>
Size	<code>length()</code>
Slicing	<code>my_vector[i]</code>
Filling	<code>my_vector[i] <- "toto"</code>
Naming	<code>names()</code>

Les matrices



Claire Vandiedonck - Université de Paris

Matrix

Matrix: 2-dimension object (rows x columns)
contains only one kind of variables (eg. numeric) = homogeneous

↳ fonctions to create a matrix

`matrix()`

`rbind()` to append rows

`cbind()` to bind columns

```
> myData <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol = 3, byrow = TRUE)
```

```
> myData
```

```
  [,1] [,2] [,3] # returns a matrix of 2 rows and 3 columns
```

```
[1,]  1  2  3 # it is filled by rows with the data provided in vector c(1,2,3, 11,12,13)
```

```
[2,] 11 12 13
```

```
> myData <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol = 3, byrow = FALSE)
```

```
> myData
```

```
  [,1] [,2] [,3] # same but filling the matrix by columns = the default
```

```
[1,]  1  3 12
```

```
[2,]  2 11 13
```

```
> length(myData) # total number of elements (like a vector, elements read by columns)
```

```
[1] 6
```

```
> myData[4] # returns the 4th element read by columns
```

```
[1] 11
```

Matrix indexes

Indexes and dimensions:

```
> myData
[1,] [2,] [3,]
[1,]  1  3 12
[2,]  2 11 13

Dim(myData)
[1] 2 3
```

☞ Note the indexes separated by a comma:

[i,] indicates the i^{th} row

[,j] indicates the j^{th} column

Subsetting matrices into vectors:

```
> myData[1,2] # returns the value of the 1st row and 2nd column
[1] 3
> myData[2,1] # returns the value of the 2nd row and 1st column
[1] 2
> myData[,1] # returns the values of the vector corresponding to the 1st column
[1] 1 2
> myData[2,] # returns the values of the vector corresponding to the 2nd row
[1] 2 11 13
> myData[,2:3] # subsets the initial matrix returning a sub-matrix
  [1,] [2,]
[1,]  3 12
[2,] 11 13
# with all rows of the 2nd and 3rd columns from the initial matrix
# the generated matrix has 2 rows and 2 columns
> dim(myData[,2:3]) # the generated matrix has 2 rows and 2 columns
[1] 2 2
```

Matrix structure and dimension

Getting dimension and structure of a matrix:

```
> dim(myData)
[1] 2 3
> mode(myData) # returns the type of data
[1] "numeric"
> class(myData) # returns the kind of object
[1] "matrix"
> class(myData[,1]) # reminder for vectors, class returns the type of data and not vector itself
[1] "numeric"
> str(myData)
num [1:2, 1:3] 1 2 3 11 12 13 # data type and dimensions
```

Or look at the length of the rows and of the columns

```
> nrow(myData)
[1] 2
> ncol(myData)
[1] 3
> length(myData[1,]) # length of the first row = hence number of columns
[1] 3
> length(myData[,1]) # length of the first column = hence number of rows
[1] 2
> dim(myData)[1] # the first dimension = hence number of columns
[1] 2
```


Creating an empty matrix

```
> newmatrix <- matrix(NA, 2,3) # or matrix(, 2,3) with NA by default
```

```
> newmatrix
```

```
      [,1] [,2] [,3]  
[1,]  NA  NA  NA  
[2,]  NA  NA  NA
```

```
> dim(newmatrix)
```

```
[1] 2 3
```

```
> is.na(newmatrix)
```

```
      [,1] [,2] [,3]  
[1,] TRUE TRUE TRUE  
[2,] TRUE TRUE TRUE
```

```
> mode(newmatrix)
```

```
[1] "logical"
```

```
> class(newmatrix)
```

```
[1] matrix
```

```
> str(newmatrix)
```

```
logi [1:2, 1:3] NA NA NA NA NA NA
```

Filling a matrix

```
> newmatrix[2,3] <- "toto"           # filling the 2nd row and 3rd column value
> newmatrix
  [,1] [,2] [,3]
[1,] NA  NA  NA
[2,] NA  NA  "toto"
> newmatrix[,1] <- "tutu"           # filling the 1st column with same values
> newmatrix
  [,1] [,2] [,3]
[1,] "tutu" NA  NA
[2,] "tutu" NA  "toto"
> newmatrix[,2] <- c("titi", "tata") # filling the 2nd column with different values
> newmatrix
  [,1] [,2] [,3]
[1,] "tutu" "titi" NA
[2,] "tutu" "tata" "toto"
> is.na(newmatrix)                   #testing whether the values in the matrix are missing values
  [,1] [,2] [,3]
[1,] FALSE FALSE TRUE           #only the 3 values in first row was NA
[2,] FALSE FALSE FALSE
```

Creating a matrix with cbind/rbind

↪ with `cbind()`: binding vectors by columns

```
> myData2 <- cbind(weight, size, bmi) # creates another matrix binding vectors as columns
```

```
> row.names(myData2)
```

```
[1] "Fabien" "Pierre" "Sandrine" "Claire" "Bruno" "Delphine"
```

```
> myData2
```

	weight	size	bmi
Fabien	60	1.75	19.59184
Pierre	72	1.80	22.22222
Sandrine	57	1.65	20.93664
Claire	90	1.90	24.93075
Bruno	95	1.74	31.37799
Delphine	72	1.91	19.73630

↪ with `rbind()`: binding vectors by rows

```
> myData3 <- rbind(weight, size, bmi)
```

```
> myData3
```

	Fabien	Pierre	Sandrine	Claire	Bruno	Delphine
weight	60.00000	72.00000	57.00000	90.00000	95.00000	72.00000
size	1.75000	1.80000	1.65000	1.90000	1.74000	1.91000
bmi	19.59184	22.22222	20.93664	24.93075	31.37799	19.73630

```
> t(myData2) # transpose myData2 -> we obtain the same matrix as myData3!
```

Note: `t()` is a function, so do not call an R object `t`!

Use a color-case text editor for R to know reserved words

Row/column names of matrices

Names of rows and columns are stored in a vector :

```
> rownames(myData2)           # returns the vector of the names of the rows
[1] "Fabien" "Pierre" "Sandrine" "Claire" "Bruno" "Delphine"
> colnames(myData2)           # returns the vector of the names of the column
[1] "weight" "size" "bmi"
> rownames(myData)            # the vector is empty for this other object
NULL
> colnames(myData)            # the vector is empty for this other object
NULL
```

Giving names to rows and columns :

```
> colnames(myData) <- c("one", "two", "three") # gives names to columns
> rownames(myData) <- c("A", "B")             # gives names to rows
> myData
  one two three
A  1  3  12
B  2 11  13
```

Subsetting matrices using row/column names :

```
> myData["B",]                # gets row called "B"
  one two three
  2  11  13
> myData[, "two"]             # gets column called "two"
  A B
  3 11
```

Functions on matrices

As expected, operators work on numeric matrices:

```
> myData2*2
```

	weight	size	bmi
Fabien	120	3.50	39.18367
Pierre	144	3.60	44.44444
Sandrine	114	3.30	41.87328
Claire	180	3.80	49.86150
Bruno	190	3.48	62.75598
Delphine	144	3.82	39.47260

All columns of a matrix can be explored at once with some functions

```
> summary(myData2)
```

	weight	size	bmi
Min.	:57.00	Min.:1.650	Min.:19.59
1st Qu.:	63.00	1st Qu.:1.742	1st Qu.:20.04
Median	:72.00	Median :1.775	Median :21.58
Mean	:74.33	Mean :1.792	Mean :23.13
3rd Qu.:	85.50	3rd Qu.:1.875	3rd Qu.:24.25
Max.	:95.00	Max.:1.910	Max.:31.38

But most functions need to specify the vector corresponding to the column of interest

```
> mean(myData2)
```

```
[1] 33.08587
```

```
> mean(myData2[,1])
```

```
[1] 74.33333
```

```
# mean of all data
```

```
# mean of the vector corresponding to
```

```
# the first column only
```

Summary on matrices

Format

two-dimensions

Datatype

`class()` to check it is a matrix

homogeneous: only one type of character, numeric, logical, factor
-> coercion if heterogeneous -> check with `mode()`

Creation

`matrix()` , `cbind()` , `rbind()`

Adding new items `cbind()` , `rbind()`

Size

`length()` -> nb of items

Dim

`dim()`, `str()`

Slicing

`my_vector[i,j]`

Filling

`my_vector[i,j] <- "toto"`

Naming

`colnames()` , `rownames()`

Saving data...

I can save the whole data in a single .Rdata object:

```
> save.image(file="Prerequis.RData")
```

Or I can save only some anthropometric data (weight, size and bmi) :

```
> save(weight,size,bmi,file=" anthropo.Rdata")
```

I can load them in a new R session with the following command:

```
> load("anthropo.Rdata")  
> ls()  
[1] "bmi"  "size" "weight"
```