



Bases de R et Rmd

Teachers: Claire Vandiedonck, Antoine Bridier-Nahmias
Helpers: Jacques van Helden, Anne Badel

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

Plan du module et intervenants

Responsables : Claire Vandiedonck et Jacques van Helden

Autres intervenants : Guillaume Achaz, Anne Badel, Magali Berland, Antoine Bridier-Nahmias, Olivier Sand, Natacha Cerisier,

Site Web : <https://du-bii.github.io/module-3-Stat-R/>

Jour	Horaire	Description
4 mars	9h30 - 12h30	Bases de R et Rmd <i>Claire Vandiedonck, Antoine Bridier-Nahmias</i>
5 mars	13h30 - 16h30	Statistiques descriptives, tests d'hypothèses, Figures et Paquets <i>Claire Vandiedonck, Guillaume Achaz</i>
10 mars	14h30 - 17h30	Statistiques pour les données à haut débit <i>Jacques van Helden, Claire Vandiedonck</i>
12 mars	9h00 - 12h00	Classification non supervisée <i>Anne Badel, Jacques van Helden</i>
30 mars	10h00 - 13h00	Analyses exploratoires (ACP/MDS) et analyses d'enrichissement <i>Magali Berland, Jacques van Helden</i>
30 mars	14h30 - 17h30	Classification supervisée et apprentissage <i>Jacques van Helden, Olivier Sand</i>

Plan de la session

1. Start-R: connexion au serveur Rstudio de l'IFB
2. Vérification et consolidation des pré-requis
3. Dataframes

Facteurs

Listes

4. Programmation
 - Executions conditionnelles
 - Boucles
 - Fonctions

5. Rmarkdown

Comment participer ?



WEB

- 1 Connectez-vous sur www.wooclap.com/EGIDTQ
- 2 Vous pouvez participer

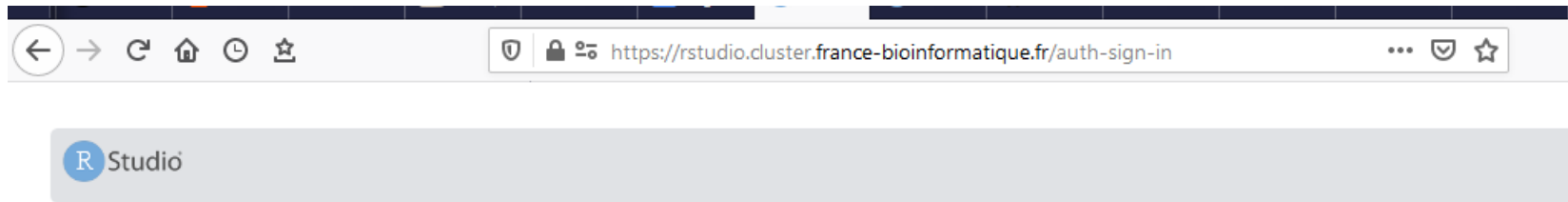


1. Start-R

First steps with R and Rstudio

Connexion au serveur Rstudio de l'IFB

<https://rstudio.cluster.france-bioinformatique.fr/>



Sign in to RStudio

Username:

Password:

☐ Stay signed in

Sign In

Connexion au serveur Rstudio de l'IFB

Browser address bar: <https://rstudio.cluster.france-bioinformatique.fr>

RStudio menu bar: File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, Help

Console pane:

```
/shared/mfs/data/home/cvandiedonck/DUBii/module-3-Stat-R/

R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-redhat-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

Environment pane:

Global Environment

Environment is empty

Files pane:

Name	Size	Modified
..		
.Rhistory	246 B	Mar 2, 2020, 4:32 PM
DUBii		
R		

Tutorial start-R.html

For the next 10 minutes:

start-R activity with the Rstudio server of the IFB cluster by following the instructions of the **start-R.html** file

- at the end of this activity, you must have uploaded in a dedicated folder:
 - the « anthropo.Rdata » generated during the prerequisites activity
 - the script of the slides of this R session 1

2. Prérequis acquis?

Let's check with a quizz!

Quizz on moodle:

- Si vous avez un compte ENT:

<https://moodlesupd.script.univ-paris-diderot.fr/course/view.php?id=10629>

- Si vous n'avez pas encore de compte ENT:

<https://moodlesupd.script.univ-paris-diderot.fr/course/view.php?id=13420>

mot de passe: dubii2020

Summary on vectors

Format	one-dimension
Datatype	<p>homogeneous: only one type of character, numeric, logical, factor.. -> coercion if heterogeneous</p> <ul style="list-style-type: none">- check with <code>class()</code> or <code>mode()</code>- checking type with <code>as.num()</code> , <code>as.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>

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()`

3. dataframes



Dataframe

Dataframe = two-dimensional object that can be heterogeneous,

↳ Create a dataframe with function **data.frame()**

```
data.frame(..., row.names = NULL, check.rows = FALSE,  
           check.names = TRUE, fix.empty.names = TRUE,  
           stringsAsFactors = default.stringsAsFactors())
```

Dataframe created with existing vectors

➡ Create a dataframe with function `data.frame()`

```
> myDataf <- data.frame(weight, size, bmi)
```

```
> myDataf      # it looks pretty much like the matrix 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

```
> class(myDataf)      # but this is well a dataframe and not a matrix
```

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

```
> str(myDataf)      # this one is a homogeneous dataframe with numeric vectors
```

```
'data.frame': 6 obs. of 3 variables:
```

```
$ weight: num 60 72 57 90 95 72
```

```
$ size : num 1.75 1.8 1.65 1.9 1.74 1.91
```

```
$ bmi : num 19.6 22.2 20.9 24.9 31.4 ...
```

```
> dim(myDataf)
```

```
[1] 6 3
```

Important:

If vectors are character chains, use `stringsAsFactors=FALSE` to avoid their conversion into factors

A dataframe can be heterogeneous

➤ create a new vector with characters and include it in the dataframe

```
> gender <- c("Man","Man","Woman","Woman","Man","Woman")
```

```
> gender
```

```
[1] "Man"      "Man"      "Woman"    "Woman"    "Man"      "Woman"
```

```
> myDataf$sex <- gender # or use cbind
```

IMPORTANT: note that I directly specify the name by using a "\$«

AND this method do not transform the vector as a factor!

```
> myDataf
```

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

```
> str(myDataf) # this dataframe is heterogeneous with numeric and character values
```

```
'data.frame':  6 obs. of  4 variables:
```

```
$ weight: num  60 72 57 90 95 72
```

```
$ size  : num  1.75 1.8 1.65 1.9 1.74 1.91
```

```
$ bmi   : num  19.6 22.2 20.9 24.9 31.4 ...
```

```
$ sex   : chr  "Man" "Man" "Woman" "Woman" ...
```


Creating an empty dataframe

↪ creating an empty dataframe?

```
> d <- data.frame()
> d
data frame with 0 columns and 0 rows
> dim(d)
[1] 0 0
```

BUT USELESS : impossible to fill!

↪ Better way: converting a matrix in a dataframe with function **as.data.frame()**

```
> d <- as.data.frame(matrix(NA,2,3))
> d
  V1 V2 V3 # by default, col names are V1, V2, etc...
1 NA NA NA # while if you are using the function
2 NA NA NA # data.frame() and not as.data.frame(),
             #col names are called X1, X2, etc...

> dim(d)
[1] 2 3

> str(d)
'data.frame':  2 obs. of  3 variables:
 $ V1: logi  NA NA
 $ V2: logi  NA NA
 $ V3: logi  NA NA
```

```
> class(myData2)
[1] "matrix"
> class(as.data.frame(myData2))
[1] "data.frame"
```

You may also use data.frame on a matrix generated by binding rows or columns

```
> d2 <- as.data.frame(cbind(1:2, 10:11))
> str(d2)
'data.frame':  2 obs. of  2 variables:
 $ V1: int   1  2
 $ V2: int  10 11
```

Row/Column names of dataframes

✚ Either use same fonctions as for matrices
`rownames()` and `colnames()`

✚ Or better use the ones dedicated to dataframes
`row.names()` and `names()`

```
> row.names(d)
[1] "1" "2"
> names(d)
[1] "V1" "V2" "V3"
```

Important:
each row name
must be unique!

Note: data.frames are a special case of a list of variables of the same number of rows with unique row names

Extracting vectors from dataframes

Getting the vector corresponding to a column from a dataframe:

↪ either by specifying its index

```
> myDataf[,2]
```

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

↪ Or by giving its name within the " " inside the squared brackets

```
> myDataf["size"]
```

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

↪ Or by giving its name after the character « \$ »

```
> myDataf$size
```

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

Extracting rows from dataframes

Getting a « dataframe » corresponding to a row from a dataframe:

↪ either by specifying its index

```
> myDataf
```

	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

```
> myDataf[2,]
```

	weight	size	bmi
Pierre	72	1.8	22.22222

↪ Or by giving its name within the " " inside the squared brackets

```
> myDataf["Pierre",]
```

	weight	size	bmi
Pierre	72	1.8	22.22222

```
> class(myDataf["Pierre",]) # Note this is not a vector but a dataframe
```

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

Let's summarize and give it a try

How do we create a dataframe?

Which are the three methods to slice datrames?

Which command should I use to extract the blue cells of the 3 dataframes below?

V1	V2	V3	V4

V1	V2	V3	V4

V1	V2	V3	V4

How to extract the even columns if I have 500 000 columns?

Filtering dataframes on criteria

It generates a new dataframe

⇒ use `which()` that returns the index of what is TRUE in the condition.

```
> which ( myDataf$sex == "Woman")
[1] 3 4 6
> myDataf [ which ( myDataf$sex == "Woman") , ]
      weight size      bmi    sex
Sandrine    57 1.65 20.93664 Woman
Claire      90 1.90 24.93075 Woman
Delphine    72 1.91 19.73630 Woman
> str(myDataf [ which ( myDataf$sex == "Woman") , ])
'data.frame':   3 obs. of  4 variables:
 $ weight: num   57  90  72
 $ size  : num   1.65 1.9 1.91
 $ bmi   : num   20.9 24.9 19.7
 $ sex   : chr   "Woman" "Woman" "Woman"
```

Important:
you may enter
this without
including
« which »
BUT this would
not deal with NA
values
=> safer to use
which

Or what does not match using `"!="` for "different" or `"!"` for "not" before the test

```
> which ( myDataf$sex != "Man")
[1] 3 4 6
> which ( ! myDataf$sex == "Man")
[1] 3 4 6
```

Filtering dataframes on criteria

✚ use `grep()` that returns the index of what matches (even partially)

```
> grep("Wom", myDataf$sex)
```

```
[1] 3 4 6
```

```
> grep("Woman", myDataf$sex)
```

```
[1] 3 4 6
```

```
> myDataf [grep("Woman", myDataf$sex), ]
```

	weight	size	bmi	sex
Sandrine	57	1.65	20.93664	Woman
Claire	90	1.90	24.93075	Woman
Delphine	72	1.91	19.73630	Woman

```
> grep("a", row.names(myDataf)) # returns indexes of rows with an "a" in its name
```

```
[1] 1 3 4
```

```
> myDataf [grep("a", row.names(myDataf)),]
```

	weight	size	bmi	sex
Fabien	60	1.75	19.59184	Man
Sandrine	57	1.65	20.93664	Woman
Claire	90	1.90	24.93075	Woman

Filtering dataframes on criteria

Subsetting the rows on the columns:

↳ use `subset()` : the easiest and most efficient way!

```
> WomenDataf <- subset(myDataf, sex== "Woman")
```

```
> WomenDataf
```

	weight	size	bmi	sex
Sandrine	57	1.65	20.93664	Woman
Claire	90	1.90	24.93075	Woman
Delphine	72	1.91	19.73630	Woman

Filtering dataframes on several criteria

logical: **&** = and, **|** = or, **!** = not

comparison: **==**, **!=** (different), **>**, **<**, **>=**, **<=**

what is included in a vector using : **%in%**

```
> filteredData <- myDataf [ which ( myDataf$sex == "Woman" & myDataf$weight < 80 & myDataf$bmi > 20), ]
```

```
> filteredData
```

only one woman with these criteria!

```
      weight size  bmi sex
Sandrine    57 1.65 20.93664 Woman
```

Or more easily with **subset()**

```
> subset( myDataf, sex == "Woman" & weight < 80 & bmi > 20)
```

```
      weight size  bmi  sex
Sandrine    57 1.65 20.93664 Woman
```

Tuto on logical values



Learn R, in R.

Let's use the « swirl » library

```
> 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()
```

Then follow the instructions and do **tuto n°8** Logic of the course
"R Programming"
=> To study on your own

```
| 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.
```

Adding new vectors to a dataframe

Either enter one vector at a time as a new variable

```
my_dataframe$new_variable <- my_variable
```

Or several vectors or subsets of dataframes at once

➤ Using `data.frame()`

```
mynew_dataframe <- data.frame(data.frame1, data.frame2)  
# this method will keep the data types of each data.frame
```

➤ Using `cbind()`

```
mynew_dataframe <- cbind(data.frame1, data.frame2)  
# BE CAREFULL: this method will keep the data types only if the  
data.frames 1 and 2 had several variables.  
If they have only one, these variables are converted as vectors and cbind  
will convert charcater strings as factors.
```

Merging dataframes

Merge two dataframes with a key

```
> myDataf$index <- 1:6 #in this example I create a new column for the key,  
# but I may use an existing one
```

```
> myDataf
```

	weight	size	bmi	sex	index
Fabien	60	1.75	19.59184	Man	1
Pierre	72	1.80	22.22222	Man	2
Sandrine	57	1.65	20.93664	Woman	3
Claire	90	1.90	24.93075	Woman	4
Bruno	95	1.74	31.37799	Man	5
Delphine	72	1.91	19.73630	Woman	6

```
> OtherData <- data.frame(c(1:5, 7),rep(c("right-handed","left-handed"),3))
```

```
> names(OtherData) <- c("ID","handedness")
```

```
> OtherData
```

	ID	handedness
1	1	right-handed
2	2	left-handed
3	3	right-handed
4	4	left-handed
5	5	right-handed
6	7	left-handed

Merging dataframes

Merge two dataframes with a key

```
> myDataf$index <- 1:6
```

```
> myDataf
```



	weight	size	bmi	sex	index
Fabien	60	1.75	19.59184	Man	1
Pierre	72	1.80	22.22222	Man	2
Sandrine	57	1.65	20.93664	Woman	3
Claire	90	1.90	24.93075	Woman	4
Bruno	95	1.74	31.37799	Man	5
Delphine	72	1.91	19.73630	Woman	6

```
> OtherData <- data.frame(c(1:5, 7), rep(c("right-handed", "left-handed"), 3))
```

```
> names(OtherData) <- c("ID", "handedness")
```

```
> OtherData
```

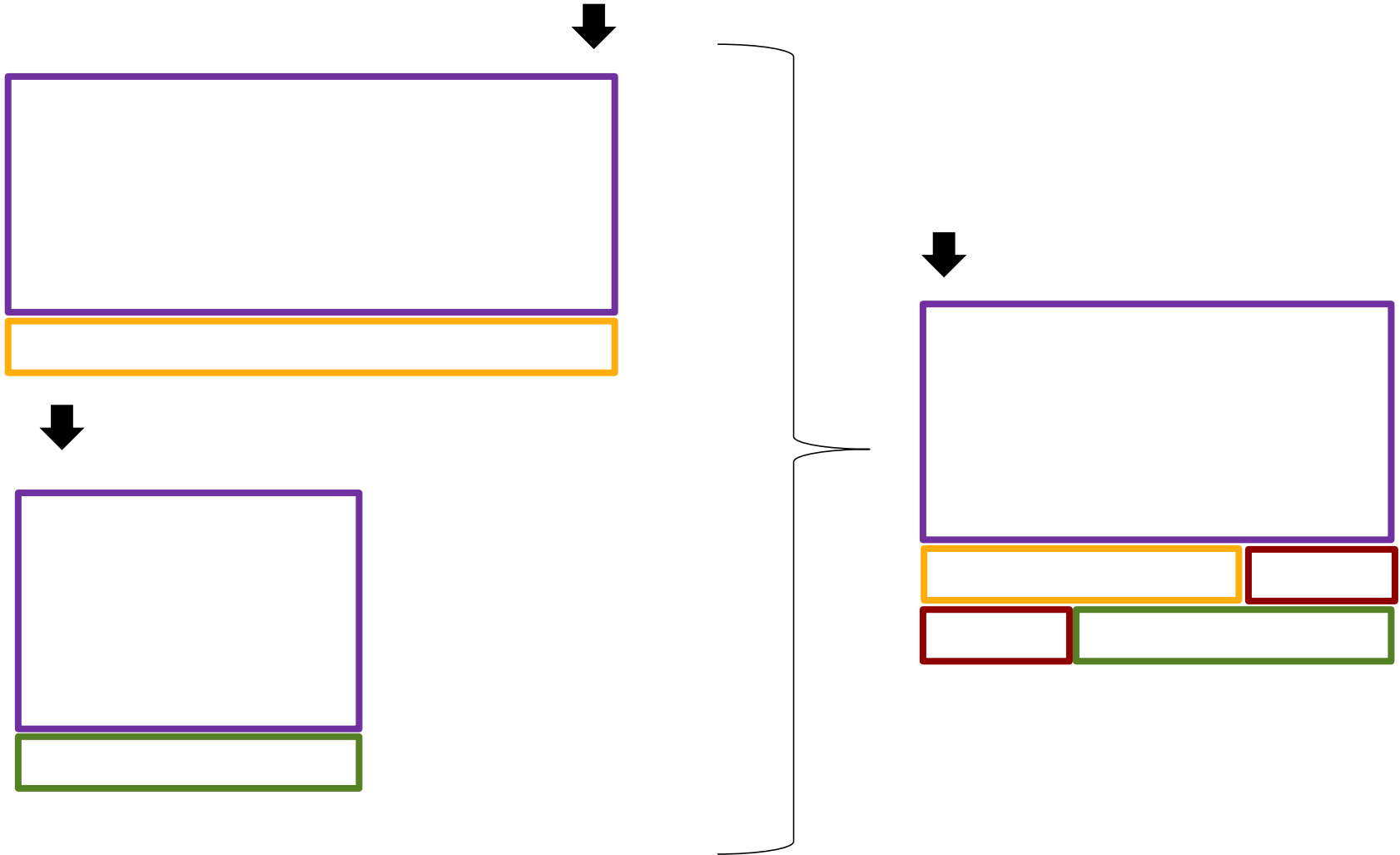


	ID	handedness
1	1	right-handed
2	2	left-handed
3	3	right-handed
4	4	left-handed
5	5	right-handed
6	7	left-handed

Merging dataframes

Merge two dataframes with a key

with `merge(dataframe1,dataframe2,by="key" , all=T, sort=F)`



Merging dataframes

Merge two dataframes with a key

```
> myMergedDataf <- merge(myDataf, OtherData, by.x="index", by.y="ID", all.x=T,  
all.y=T, sort=F)
```

```
> myMergedDataf
```



	index	weight	size	bmi	sex	handedness
1	1	60	1.75	19.59184	Man	right-handed
2	2	72	1.80	22.22222	Man	left-handed
3	3	57	1.65	20.93664	Woman	right-handed
4	4	90	1.90	24.93075	Woman	left-handed
5	5	95	1.74	31.37799	Man	right-handed
6	6	72	1.91	19.73630	Woman	<NA>
7	7	NA	NA	NA	<NA>	left-handed

- ✓ unless the merge is done on the `row.names()`, the `row.names` of initial data.frames are lost -> the new data.frame has its own row names
- ✓ if two columns had the same name, a « .x » or a « .y » is added to the first/second

Reading a text file into R

Read a text file using `read.table()`:

```
> temperatures <- read.table("Temperatures.txt", sep="\t", header=T, stringsAsFactors=F)
```

```
> temperatures
```

	Month	Mean_Temp
1	January	2.0
2	February	2.6
3	March	7.9
4	April	11.2
5	May	15.3
6	June	22.2
7	July	22.9
8	August	22.5
9	September	17.3
10	October	11.7
11	November	5.2
12	December	2.8

```
> str(temperatures)
```

```
'data.frame': 12 obs. of 2 variables:
```

```
$ Month : chr "January" "February" "March" "April" ...
```

```
$ Mean_Temp: num 2 2.6 7.9 11.2 15.3 22.2 22.9 22.5 17.3 11.7 ...
```

specify the field
separator of the
text file

TRUE if
header in
the text file

FALSE to avoid
factorisation of
character vectors

the R object is a dataframe !!!!

Reading a text file into R

Warning: use **stringsAsFactors=F** otherwise vectors of character values converted into factors -> see below, the Months were factorized!!!!

```
> temperatures <- read.table("Temperatures.txt", sep="\t", header=T, stringsAsFactors=T)
> str(temperatures)
'data.frame': 12 obs. of 2 variables:
 $ Month   : Factor w/ 12 levels "April","August",...: 5 4 8 1 9 7 6 2 12 11 ...
 $ Mean_Temp: num  2 2.6 7.9 11.2 15.3 22.2 22.9 22.5 17.3 11.7 ...

> levels(temperatures$Month) # the levels of the factor are in alphabetic order
[1] "April"    "August"   "December" "February" "January"  "July"
[7] "June"     "March"    "May"       "November" "October"  "September"
```

TRUE is by
default

Factors in R

See tutorial [Factors_in_R.html](#)

Much care on:

- levels order
- coercion

=> To study on your own

Reading a text file into R

Caution if:

- fewer names than columns in the header
- fewer columns than names in the header -> add argument `fill=T` to overcome the issue
- some rows with fewer columns -> add argument `fill = T` to overcome the issue
- using `row.names=1` -> this cannot be used when several rows have the same name

Check the data.frame is as expected using:

`str()`

`head()` : displays the first 6 rows

`tail()` : displays the last 6 rows

and by displaying some rows in the middle of the file using their index

-> a general habit with any programming language

Other functions: `read.csv()`, `scan()`

or `read.xlsx()` to read worksheet from an excel file with library « `xlsx` »

Saving a dataframe as a text file in the working directory

Saving a dataframe into a text file using `write.table()`

```
> write.table(myDataf, file="bmi_data.txt", sep="\t", quote=F, col.names=T)
```

specify the name of the
saved text file

Specify the column
separator in the text file
"t" is for tabulation

FALSE to avoid quotes
flanking strings of characters in
the saved text file

to write the column
names in a header

Tutorial = [basic_R-structures.html](#)

Lists



=> To study on your own

Lists

A list is an R object that can contains:

- heterogeneous elements including other lists unlike vectors
- all elements do not need to be of same dimensions unlike dataframes

We have already seen a list when looking at the names of the matrix 'myData2'

```
> str(myData2)
```

```
num [1:6, 1:3] 60 72 57 90 95 72 1.75 1.8 1.65 1.9 ...
```

```
- attr(*, "dimnames")=List of 2
```

```
..$ : chr [1:6] "Fabien" "Pierre" "Sandrine" "Claire" ...
```

```
..$ : chr [1:3] "weight" "size" "bmi"
```

Creating a list

➤ Use `list()` to create an empty list

```
> L0 <- list()
> class(L0)
[1] "list"
```

➤ Use `list()` to create a non-empty list filling it with other R objects

```
> x <- c("A", "B", "C")
> y <- 8:15
> L1 <- list(x, y)
> str(L1)
List of 2                                # the list L1 contains two elements
 $ : chr [1:3] "A" "B" "C"
 $ : int [1:8] 8 9 10 11 12 13 14 15
> L1
[[1]]                                     # the first element's name
[1] "A" "B" "C"                           # the first element's content

[[2]]                                     # the second element's name
[1] 8 9 10 11 12 13 14 15 # the second element's content
```


Creating a list

➤ Add names to elements

```
> names(L1)
NULL
> names(L1) <- c("ID1","ID2")
> L1
$ID1
[1] "A" "B" "C"

$ID2
[1] 8 9 10 11 12 13 14 15
```

➤ Or add names to elements when creating the list:

```
> L2 <- list("ID1"=x,"ID2"=y)
> L2
$ID1
[1] "A" "B" "C"

$ID2
[1] 8 9 10 11 12 13 14 15
```

Accessing to list elements

↪ Use `[[i]]` to get the i^{th} element of the list

```
> L1[[2]]
```

```
[1] 8 9 10 11 12 13 14 15
```

↪ or using the name of the element

```
> L1[["ID2"]]
```

```
[1] 8 9 10 11 12 13 14 15
```

↪ or using `$` followed by the name of the element if there is one

```
> L1$ID2
```

```
[1] 8 9 10 11 12 13 14 15
```

Adding a new element to a list

↪ Use `[[i]]`

```
> L1[[3]] <- matrix(1:4,2,2)
> L1
$ID1
[1] "A" "B" "C"

$ID2
[1]  8  9 10 11 12 13 14 15

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

↪ or give a name

```
> L1[["m1"]] <- matrix(1:4,2,2)
```

↪ or using `$` followed by the name of the element if there is one

```
> L1$m2 <- matrix(1:4,2,2)
```

Removing an element from a list

↪ Use NULL

```
> L1[[3]] <- NULL
```

```
> L1
```

```
$ID1
```

```
[1] "A" "B" "C"
```

```
$ID2
```

```
[1] 8 9 10 11 12 13 14 15
```

```
$m1
```

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

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

```
[2,]     2     4
```

```
$m2
```

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

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

```
[2,]     2     4
```

4. Programming

4.1 Conditional executions

=> To study on your own

Conditional executions: the basis

Aim:

- To perform a test with a logical outcome
 - comparison: `==`, `!=` (different), `>`, `<`, `>=`, `<=`
 - what is included in a vector using : `%in%`
 - etc...
- And if the outcome is « TRUE » to execute the commands between { and }

Syntax:

using **if**(« the condition ») followed by {«commands to be performed **if TRUE**»}

```
if (condition){  
  instruction 1  
  instruction 2  
  ...  
}
```

```
> a <- 0  
+ if (a == 0) {  
+   print ("hello")  
+ }  
[1] "hello"  
  
> if (a != 0) {  
+   print ("a is different from zero")  
+ }  
>  
# here R did not execute print ("a is different from zero")  
# since the condition was FALSE
```

Conditional executions: alternative conditions

Syntax:

using **if**(« the condition ») followed by {«commands to be performed **if TRUE**»}

adding **else**() followed by {«alternative commands if condition was FALSE»}

adding **else if**() followed by {«alternative commands if new condition is TRUE»}

```
a <- 3
if (a < 3.14) {
  print ("a is < 3.14 ")
} else {
  print ("a is > 3.14")
}
[1] "a is < 3.14 "
```

```
a <- 3.14
if (a < 3.14) {
  print ("a is < 3.14 ")
} else if (a > 3.14){
  print ("a is > 3.14")
} else {print ("a is equal to 3.14")}
[1] "a is equal to 3.14 "
```


Conditional executions: multiple conditions (1)

Example with multiple conditions:

using « & » = et, « | » = or

```
a <- 11
if ( (a < 2) & (a < 10) ) {
  print ("both conditions are true")
} else if ( (a < 2) | (a < 10) ) {
  print ("one of the two conditions is true")
} else {
  print ("none of the conditions is verified")
}
[1] "none of the conditions is verified"
```

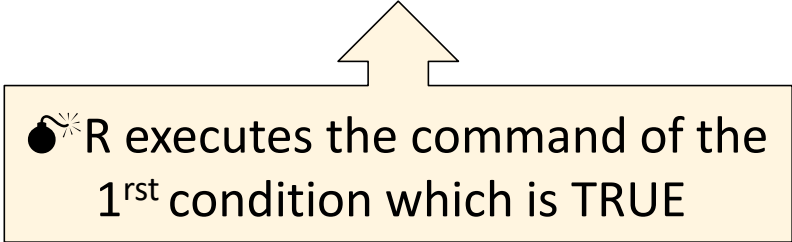
```
a <- 6
if ( (a < 2) & (a < 10) ) {
  print ("both conditions are true")
} else if ( (a < 2) | (a < 10) ) {
  print ("one of the two conditions is true")
} else {
  print ("none of the conditions is verified")
}
[1] "one of the two conditions is true"
```

Conditional executions: multiple conditions (2)

💣 If your conditions are not mutually exclusive, order carefully your conditions

```
a <- 1
if ( (a < 2) & (a < 10) ) {
  print ("both conditions are true")
} else if ( (a < 2) | (a < 10) ) {
  print ("one of the two conditions is true")
} else {
  print ("none of the conditions is verified")
}
[1] "both conditions are verified"
```

```
a <- 1
if ( (a < 2) | (a < 10) ) {
  print ("one of the two conditions is true")
} else if ( (a < 2) & (a < 10) ) {
  print ("both conditions are true")
} else {
  print ("none of the conditions is verified")
}
[1] "one of the two conditions is true"
```



💣 R executes the command of the
1st condition which is TRUE

Fonction ifelse()

ifelse()

- returns a vector of same length as the length of the tested vector
- for each element of the tested vector, the elements of the returned vector are defined depending on whether the condition is TRUE or FALSE

syntax: **ifelse**(test, yes, no)

```
a <- 3
ifelse(a == 3.14, "a is equal to pi", "a is different from pi" )
[1] "a is different from pi"
a <- 3.14
ifelse(a == 3.14, "a is equal to pi", "a is different from pi" )
[1] "a is equal to pi"
```

```
norm_values <- rnorm(10, 0, 1)
below_median <- ifelse(norm_values < median(norm_values), TRUE, FALSE)
table(below_median)
below_median
FALSE TRUE
 5    5
```

4.2. Loops

Loops / iterations

Aim: repeat a command or a set of commands several times for each value of the variable

Syntax: using **for()** followed by **{}**

for(«how to repeat on numbers/values of a variable or on vector indexes») {
«commands to be repeated for each value of the variable» }

```
for(i in 1:6){  
  print(i)  
}
```

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

[1] 6

```
counter <- 0  
for (i in seq(5,8)) {  
  counter <- counter + i  
  cat(counter, "\n")  
}
```

5

11

18

26

```
teachers <- c("Pierre","Claire")  
for (t in teachers) {  
  cat(t,"was one of my bioinformatics teachers \n")  
}
```

Pierre was one of my bioinformatics teachers

Claire was one of my bioinformatics teachers

Loops and condition

Fonction **while()**

- executes the instruction as long as the condition is TRUE, stops as soon as the condition is FALSE

syntax:

```
while( "the condition to be tested" ){  
    "instruction"  
}
```

```
i <- 0  
while (i < 5) {  
  i <- i + 1  
  print(i)  
}  
[1] 1  
[1] 2  
[1] 3  
[1] 4  
[1] 5
```

4.3. Vectorization

Vectorization instead of iterations

Unlike in other programming languages, in R vectorization makes some loops implicit and is computationally more efficient

Exemple:

computing the BMI

```
weight <- c(60, 72, 57, 90, 95, 72)
```

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

➤ Using loops

```
bmi <- numeric(length(weight))
for (i in 1:length(bmi)) {
  bmi[i] <- weight[i] / size[i]^2
}
bmi
[1] 19.59184 22.22222 20.93664 24.93075 31.37799 19.73630
```

➤ Using vectorization (to be preferred): only if vectors of same length!

```
bmi <- weight/size^2
bmi
[1] 19.59184 22.22222 20.93664 24.93075 31.37799 19.73630
```


Vectorization again!

Vectorization is more efficient to replace values in a vector than conditional executions in loops

➤ **Using loops and condition**

```
for (i in 1:length(bmi)){  
  if (bmi[i] > 30) {  
    bmi[i] <- "obesity"  
  } else if ( bmi[i] < 25) {  
    bmi[i] <- "normal"  
  } else (bmi[i] <- "overweight")  
}  
bmi  
[1] "normal" "normal" "normal" "normal" "obesity" "normal"
```

➤ **Using logical indexing of vectors (to be preferred)**

```
bmi[which(bmi > 30)] <- "obesity"  
bmi[which(bmi < 25)] <- "normal"  
bmi[which(bmi <= 30 & bmi >= 25 )] <- "overweight"  
bmi  
[1] "normal" "normal" "normal" "normal" "obesity" "normal"
```

4.4. Writing your own functions

Your own R functions?

Why?

Allows efficient, flexible and rational use of R, if you want **to redo an operation in different situations**

Properties

Similar structure as native R functions, except there is no help menu

- name
- arguments put within brackets to execute the command

```
func1 <- function ( x, y, z ) {  
  ...  
  commands executing actions on x, y and z  
  ...  
  return (results)  
}
```

The diagram illustrates the structure of an R function. A blue box labeled 'arguments' points to the parameters **x, y, z** in the function definition. A green box labeled 'Body of the function' points to the block of code between the opening curly brace and the return statement. An orange box labeled 'results' points to the value **results** returned by the function.

Functions

Syntax uses two functions:

```
myFunction <- function (argument1, argument2) {  
    myResult <- « what you want to do with arguments 1 and 2 »  
    return(myResult)  
}
```

1. **function()** followed by **{}**
 - assign the function to the name of the function (not already implemented in R)
 - specify the names of the created function parameters as « arguments » within the **()**;
you may specify default values with =
2. **return()** inside the **{}** of the function so that the output of the function can be saved outside of the function space; if multiple results, they must be stored in a single output in a list format

Rules in writing your own functions

1. 💣 **The name of your own function must not be a native R function**, otherwise the native R function is overwritten

exemple not to do:

```
mean <- function (x) {  
  return (x^2)  
}
```

```
mean(c(3,4))
```

```
[1] 9 16      # it returns the squared values of 3 and 4 and not the mean of 3 and 4
```

```
rm(mean)    # if you run the above command to restore the native mean function!
```

Rules in writing your own functions

2. The function space is closed

The argument names, all the variables created inside the functions and the results exist only within the enclosed function space!

- All the required objects must be arguments of the function or they must be defined in the body of the function
- Risk to call an R object that is outside your function : by default, if the object is not defined in your function, R looks for it outside the function

```
rm(a)
func <- function (x) {
  x <- x+a
  return(x^2)
}
```

```
func(2)
```

Error in x + a : non-numeric argument to binary operator

```
a <- 2
func <- function (x) {
  x <- x+a
  return(x^2)
}
```

```
func(2)
```

```
[1] 16
```

can also be run on a vector

```
func(c(2,3,10))
```

```
[1] 16 25 144
```

Rules in writing your own functions

2. The function space is closed

The argument names, all the variables created inside the functions and the results exist only within the enclosed function space!

- By defaults, R uses the object defined in your function

```
a <- 2
func2 <- function (x, a) {
  x <- x+a
  return(x^2)
}
```

```
func2(2,3)    # it uses a defined as an argument in the enclosed function
[1] 25         #and not a=2 which is in your R workspace
```

```
func2(2,10)   # idem
[1] 144
```

```
func2(2)
Error in func2(2) : argument "a" is missing, with no default
# a is a required argument and is not passed to the function
# it cannot use a=2 which is in your workspace instead
```

Arguments in your own functions

- Several arguments can be passed to your function

- They can be of different types:

numeric, logical, factors, vectors, matrices, dataframes, lists...and even functions!

- They are defined by their name or by their order

```
func3 <- function (x, a) {  
  x <- x+2*a  
  return(x^2)  
}
```

```
func3(2,5)  
[1] 144
```

```
func3(x=2, a=5)  
[1] 144
```

```
func3(a=5, x=2)  
[1] 144
```

```
func3(5,2)  
[1] 81
```


Rules in writing your own functions

3. Do not hesitate to assign default values to the arguments

- > more robust function

```
func4 <- function (x, a=4) {  
  x <- x+a  
  return(x^2)  
}
```

```
func4(2,5)  
[1] 49
```

```
func4(2)      # since there is a default value for a, there is no need to  
[1] 36        # specify it
```

Function results

By default, the returned result is the last object of the function body

```
func <- function (x) {  
  x ^2  
}  
func(2)  
[1] 4
```

It is recommended to return the result with the function `return()`

If more than one result has to be returned, use a list to store results and return the list

```
func <- function (x) {  
  temp <- x ^2  
  return(temp)  
}  
func(2)  
[1] 4
```

```
func <- function (x) {  
  temp1 <- x ^2  
  temp2 <- temp1^x  
  results <- list(res1=temp1, res2=temp2)  
  return(results)  
}  
func(2)  
$res1  
[1] 4  
  
$res2  
[1] 16
```

Functions

➤ Examples

```
rm ( list=ls() )  
f1 <- function( a,b ){  
  Op <- a + b  
}  
f1(a = 6, b = 20) # the result is not shown  
ls()  
[1] "f1" # the results is not saved in R
```

Add return()



```
rm ( list=ls() )  
f1 <- function( a,b ){  
  Op <- a + b  
  return(Op)  
}  
f1(a = 6, b = 20)  
[1] 26  
ls()  
[1] "f1" # the results is not saved in R
```

Assign the
result of the
function



```
rm ( list=ls() )  
f1 <- function( a,b ){  
  Op <- a + b  
  return(Op)  
}  
res1 <- f1(a = 6, b = 20)  
ls()  
[1] "f1" "res1" # the results is saved in R  
res1  
[1] 26
```

Functions

➤ Examples

```
rm(list=ls())
a <- 27
f1 <- function(a, b){
  Op <- a + b
  return(Op)
}
f1(a = 6, b = 20) # it uses the a value you
                  # assign within the function
[1] 26
ls()
[1] "a" "f1"
a      # a within the R session is not
[1] 27  # modified by the function
```

```
rm(list=ls())
a <- 27
f1 <- function(a=22, b){
  Op <- a + b
  return(Op)
}
f1(a, b = 20) # it uses the a value assigned
              # within the R session
[1] 47
f1(b = 20) # while here it uses the default
[1] 42      # a value specified when
           # creating the function
```



It is thus highly recommended to use different names for your R objects within function space and within your R session to avoid any confusion !

Exemple: writing a function to compute the bmi

➤ **How?**

Tutorial: [R-programming_intro.html](#)

4.5. To go further: writing an R programm

=> To study on your own

Rules of good practice

- ✓ **Use a text editor with R syntactic coloration:** eg TinnR, notepad++
- Pairing (), { }, " ", []...
- With a clear indentation for functions, loops,...

```
all.cp.signif = NULL
acp.eig = PCA(desc.prox.sle, graph = FALSE)$eig[,1]

for (i in 1:nbrSimul){
  #random matrix
  mat.alea = NULL
  for (i in 1:dim(desc.sle.lig)[2]){
    desc.perm = sample(desc.sle.lig[,i])
    mat.alea = cbind(mat.alea, desc.perm)
  }
  colnames(mat.alea) = colnames(mat.desc)
  #compute the CP
  acp.Alea.eig = PCA(mat.alea, graph = FALSE)$eig[,1]
  mat.cp = cbind(acp.eig , acp.Alea.eig)
  vCP.signif = NULL
  for (i in 1:dim(mat.cp)[1]){
    if( mat.cp[i,1] >= mat.cp[i,2]) {
      vCP.signif = c(vCP.signif, i)
    }
  }
  all.cp.signif = c(all.cp.signif, vCP.signif)
}
countCP = table( all.cp.signif)
signifCP.sle = names(which(countCP>= nseuil))
```

```
all.cp.signif = NULL
acp.eig = PCA(desc.prox.sle, graph = FALSE)$eig[,1]

for (i in 1:nbrSimul){
  #random matrix
  mat.alea = NULL
  for (i in 1:dim(desc.sle.lig)[2]){
    desc.perm = sample(desc.sle.lig[,i])
    mat.alea = cbind(mat.alea, desc.perm)
  }
  colnames(mat.alea) = colnames(mat.desc)
  #compute the CP
  acp.Alea.eig = PCA(mat.alea, graph = FALSE)$eig[,1]
  mat.cp = cbind(acp.eig , acp.Alea.eig)
  vCP.signif = NULL
  for (i in 1:dim(mat.cp)[1]){
    if( mat.cp[i,1] >= mat.cp[i,2]) {
      vCP.signif = c(vCP.signif, i)
    }
  }
  all.cp.signif = c(all.cp.signif, vCP.signif)
}
countCP = table( all.cp.signif)
signifCP.sle = names(which(countCP>= nseuil))
```


Rules of good practice

✓ Explicitly name the variables:

- avoid reserved terms = prebuilt R functions or parameters -> identified by proper text editors
- strings of characters in Camel type: starts with small letters, caps for the first letter of each new word, may be separated by « . » or « _ »
- never start with a number
- no special characters
- try to use a letter defining the type of variable: v for vector, ma for matrix, d for dataframe, l for list, f for factor...

```
> theWeightofYourTeachers <- c(60,55,89,44,132)
```



```
> vTeachersWeight <- c(60,55,89,44,132)
```

Rules of good practice

✓ An understandable script for you and others

- one command per line, leave blank lines between blocks
- using clear indentations

```
> valCount <- 0; randomVal <- rnorm(1000,3,5) ; for (i in randomVal) {if (i >=3) { valCount <- valCount +1 }}
```



```
> valCount <- 0
randomVal <- rnorm(1000,3,5)
for (i in randomVal) {
  if (i >=3)
    { valCount <- valCount +1 }
}
```

- well-organized: group paths for inputs and outputs in the same section, group variable assignments together, structure scripts with modular functions that can be reused rather than writing a single very large function
- easy to modify
- easy to understand including later by you!

Rules of good practice

✓ An understandable script for you and others

- annotate your script by functions...but not every single command!

```
> vMeanMarkUE <- NULL           # initialize the mean of the marks by UE
> for (i in 1:dim(mM1MEGMarks)[2]){ # loop on each column of the matrix containing the marks by UE
  valMean <- mean(mM1MEGMarks)[,i] # compute the mean of the marks per UE
  vMeanMarkUE<- c(vMeanMarkUE,valMean) # store the mean values of the ith UE in the
                                         # vector with the means of the other UEs
}
> names(vMeanMarkUE) <- colnames(mM1MEGMarks) # assigns the names of the columns to each value
                                                # of the vector
```



```
#computes the mean value of each UE
> vMeanMarkUE <- NULL
> for (i in 1:dim(mM1MEGMarks)[2]){
  valMean <- mean(mM1MEGMarks)[,i]
  vMeanMarkUE<- c(vMeanMarkUE,valMean)

}
>names(vMeanMarkUE) <- colnames(mM1MEGMarks)
```

Rules of good practice

✓ An understandable script for you and others

- Explain the usage of your script at the beginning, including the format of the input files
- Provide a command line example

✓ For Unix usage without opening R:

- At the first line of your script, add the shebang:
`#!/usr/bin/env Rscript` -> to allow executing your script
- There are 3 ways to run an R script without opening R in Unix:
 1. `R CMD BATCH [options] /path/myscript.R [path/out.file]`
 2. `R --vanilla < myscript.R`
or `R --vanilla --args arg1 arg2 ... < /path/myscript.R` if you want to pass arguments that you can get in the R code using the fonction `commandArgs()`
 3. `Rscript /path/myscript.R arg1 arg2 path/out.file`

Some further help in:

[Genolini-RBonnesPartiques.pdf](#)

Google's R Style Guide: <https://google.github.io/styleguide/Rguide.xml>

5. Rmarkdown

Markdown and R Markdown

Simple Markdown

<https://dillinger.io/>

<https://stackedit.io/>

https://www.tablesgenerator.com/markdown_tables)

R code in markdown

➤ A live session!

Comment participer ?



WEB

- 1 Connectez-vous sur www.wooclap.com/YYLPQH
- 2 Vous pouvez participer

