DSSP
Introduction to R

A. Bichat - E. Le Pennec

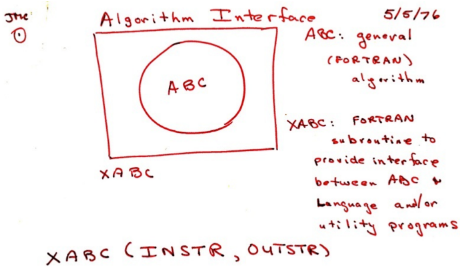
2019 - 2020

Outline

R

- R is a **software environment for statistical computing and graphics** distributed freely at **CRAN** under a GPL 2/3 licence.

R Foundation



- Created in the 90's in the spirit of S (from the 70's):
    - interactive console and graphical windows
    - efficient domain specific language
    - glue with compiled code
- Statistical analysis, visualization and data manipulation...
- and interfacing... and reporting...

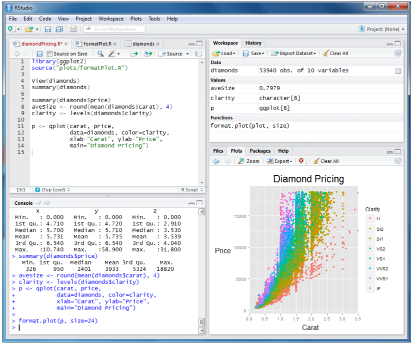
R Features

- Main features:
  - available under many OS (Unix, Linux, OS X, Windows...),
  - dedicated to statistical analysis and visualization,
  - and data manipulation, reading/writing, reporting... (more than 80% of the time is spent on data preparation)
  - more than a statistical environment (SPSS, SAS, ...) a powerful programming language
  - made of a **core** and a huge number of **packages** (collection of functions often calling external code)
  - easy interfacing with many (all?) languages: C, Fortran, Java, Python, JavaScript, C++, ...
  - and many (all?) databases : MySQL, PostgreSQL, Oracle, MS SQL, mongodb, HBase, ...
  - and many big data framework: Hadoop, Spark, H2O...

R Language

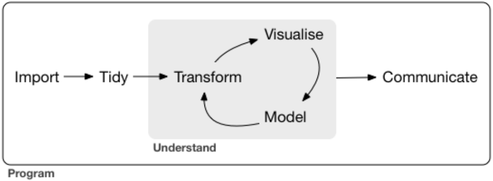
- R is an **interpreted** language
  - it requires a dedicated program, the **R** interpreter, to execute its commands
  - != **compiled** language, such as C or C++, where the code has to be converted in machine language by a compiler before it can be executed.
- Language:
  - functional
  - object based (everything in **R** is an object!)
  - weak typing and no formal variable declaration
- Exploit vector structure:
  - vectorized computations
  - reduced number of loops
- Focus on the short developing time... rather than on raw performance
- Ever growing number of users and developers
  - state of the art methodologies / technologies

R and Rstudio



- Interaction with the data in a console
- Programming/scripting via text file
- Rstudio: successful IDE
  - Free version at http://www.rstudio.com
  - Literate programming with R Markdown and Notebooks
- Other IDE exists: Visual Studio, Emacs. . .

Tidyverse



- Collection of packages to simplify the life of **R** users: https://www.tidyverse.org/
- Principle:
  - functional programming, uniformized syntax, pipe
  - *tidy* dataset
- Large number of packages
  - Programming: **purrr**, **testthat**,...
  - Input/Output: **readr**, **haven**,...
  - Data.frame: **tibble**, **dplyr**,...
  - Object/vector manipulation: **lubridate**, **stringr**, **forcats**,...
  - Graphics: **ggplot2**, **ggraph**...

Self-Teaching Resources

- CRAN: http://cran.r-project.org
- R-bloggers: https://www.r-bloggers.com/
- Rweekly: https://rweekly.org/
- **swirl** package
- Books, MOOC...

Outline

Packages

- Main repository at CRAN : https://cran.r-project.org/
- Installation with the **Packages** in RStudio, or in the console with:

```
install.packages("ibr")
```

- Loading with `library()` or `require()` :

```
library(ibr)
require(ibr)
```

- Namespace can be explicitly used:

```
ibr::ibr(...)
```

Finding Help

- **Help** tab in RStudio
- `?` or `help()` with the name of a function or a package

```
?mean
help(stats)
```

- `??` for a research on a theme

```
??mean
```

Objects

- Main object types:
  - **null** : NULL
  - **logical** (Boolean) : TRUE, FALSE
  - **integer** : 1, 2, 10
  - **numeric** : 1, 10.5, 1e-10
  - **complex** : 2+0i
  - **character** (string) : 'bonjour', "hello"
  - **factor** (factor) : see later...
- Main data structures :
  - **vector** : sequence of same type elements (and **matrix**, **array**),
  - **list** : sequence of possibly different type,
  - **data.frame**: **list** of **vector**(s) of the same size.

Objects

- `ls()` or `objects()` lists the object in memory (**Environment** tab in RStudio)
- Assignation (binding) with `<-` (rather than `=`) with an implicit typing

```r
ls()
```

```
## character(0)
```

```r
x <- 2
ls()
```

```
## [1] "x"
```

Objects

- Concatenation with `c`
- Object structure with `str`

```
y <- c(x, 1, "A")
y
```

```
## [1] "2" "1" "A"
```

```
str(y)
```

```
##  chr [1:3] "2" "1" "A"
```

Objects

- Suppression with **rm()**
- Existence test with **exists()**

```
x <- 1 ;  y <- 2 ; z <- 3
objects()
```

```
## [1] "x" "y" "z"
```

```
rm(x, z)
objects()
```

```
## [1] "y"
```

```
exists("y")
```

```
## [1] TRUE
```

```
exists("z")
```

```
## [1] FALSE
```

Vectors

- **vector**: finite sequence of same type elements
- Creation / Concatenation

```
x <- c(2,-5,-6, 9) # concatenate
x # to print x (or print(x))
```

```
## [1]  2 -5 -6  9
```

```
y <- c(1, 2)
z <- c(x, y)
z
```

```
## [1]  2 -5 -6  9  1  2
```

```
c("a", "b")
```

```
## [1] "a" "b"
```

Vectors

- (Sub)Sequence with [: result is a vector

```
x[c(1,3)] # explicit indices list
```

```
## [1]  2 -6
```

```
x[-3] # indices exclusion list
```

```
## [1]  2 -5  9
```

```
x[(x %% 2) == 0] # logical selection
```

```
## [1]  2 -6
```

```
x[c(1,1,1)] # indices can be repeated
```

```
## [1] 2 2 2
```

Vectors

- Extraction with `[[`: result is a unique element

```r
x[[1]]
```

```
## [1] 2
```

- **Rk :** for vectors, a unique element is a vector of size 1!

```r
x[[1]] == x[1]
```

```
## [1] TRUE
```

- Assignment :

```
x
```

```
## [1]  2 -5 -6  9
```

```
x[c(1,2)] <- c(1,4); x
```

```
## [1]  1  4 -6  9
```

```
x[x %% 2 == 0] <- 0 # implicit value reuse
                       # if left hand side vector length
                       # is a multiple
x                      # of right hand side vector length
```

```
## [1] 1 0 0 9
```

Vectors


```
x[[1]] <- 3; x # unique element assignment
## [1] 3 0 0 9
x[5] <- 10; x # automatic vector growth
## [1]  3  0  0  9 10
```

Vectors

- Useful vector creation command:

```
1:5
```

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

```
1:0
```

```
## [1] 1 0
```

```
1:-1
```

```
## [1]  1  0 -1
```

```
seq(1, 6, by = 2)
```

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

```
seq(1, 6, by = .1)
```

Vectors


```
##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
## [12] 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1
## [23] 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2
## [34] 4.3 4.4 4.5 4.6 4.7 4.8 4.9 5.0 5.1 5.2 5.3
## [45] 5.4 5.5 5.6 5.7 5.8 5.9 6.0
```

```r
rep(1, 4)
```

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

```r
rep(c(1, 2), each = 3)
```

```
## [1] 1 1 1 2 2 2
```

Vectors

- Length

```
length(x)
```

```
## [1] 5
```

- Properties

```
is.numeric(x); is.vector(x)
```

```
## [1] TRUE
## [1] TRUE
```

Vectors

- Named vectors

```
z <- c(a = 2, bb = 3, c = 4); z
```

```
##  a bb  c
##  2  3  4
```

```
z[c("a","bb")] # result is a named vector
```

```
##  a bb
##  2  3
```

```
z["bb"]
```

```
## bb
##  3
```

Vectors


```r
z[["bb"]] # result is an unnamed vector
```

```
## [1] 3
```

```r
names(z)
```

```
## [1] "a"  "bb" "c"
```

```r
names(z) <- c("aa", "bb", "cc"); z
```

```
## aa bb cc
##  2  3  4
```

Vectors

- Matrix and array: multi dimensional extension of vectors

```r
M <-  matrix(c(1,2,3,4,5,6), nrow = 2, ncol = 3)
M # columnwise order
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```r
M[1,c(1,2)]
```

```
## [1] 1 3
```

```r
M[1,]
```

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

Vectors


```r
M[,-2]
```

```
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
```

```r
M[[1,2]]
```

```
## [1] 3
```

Vectors


```r
A <- array(c(1,2,3,4,5,6,7,8), dim = c(2,2,2)); A
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
```

Vectors


```r
A[1,,2]
```

```
## [1] 5 7
```

```r
A[[1,1,2]]
```

```
## [1] 5
```

Lists

- **list**: finite sequence of possibly different type elements
- Creation:

```
mylist <- list(c(1, 5, -3), y)
mylist <- list(myvect = c(1, 5, -3), myy = y,
               myfact = c("a", "b")) # named list
```

Lists

- Concatenation:

```r
c(mylist, list(c(2,3))) # concatenation of 2 lists
```

```
## $myvect
## [1]  1  5 -3
##
## $myy
## [1] 1 2
##
## $myfact
## [1] "a" "b"
##
## [[4]]
## [1] 2 3
```

Lists


```
c(mylist, c(2,3)) # concatenation of 3 lists!
```

```
## $myvect
## [1]  1  5 -3
##
## $myy
## [1] 1 2
##
## $myfact
## [1] "a" "b"
##
## [[4]]
## [1] 2
##
## [[5]]
## [1] 3
```

Lists

- Sub(Sequence): result is a list

```
mylist[c(1,3)]
```

```
## $myvect
## [1]  1  5 -3
##
## $myfact
## [1] "a" "b"
```

```
mylist[-1]
```

```
## $myy
## [1] 1 2
##
## $myfact
## [1] "a" "b"
```

```
mylist[c("myvect", "myy")]
```

Lists


```
## $myvect
## [1]  1  5 -3
##
## $myy
## [1] 1 2
```

```r
mylist[c(TRUE, FALSE, TRUE)]
```

```
## $myvect
## [1]  1  5 -3
##
## $myfact
## [1] "a" "b"
```

Lists


```
mylist[1] # a list of length 1!
```

```
## $myvect
## [1]  1  5 -3
```

- Extraction with `[[` (and `$`) : result is a single element of corresponding type

```
mylist[[1]]
```

```
## [1]  1  5 -3
```

```
mylist[["myfact"]]
```

```
## [1] "a" "b"
```

Lists


```r
mylist$myfact
```

```
## [1] "a" "b"
```

```r
mylist$myv # partial matching
```

```
## [1]  1  5 -3
```

```r
mylist$my  # only if not ambiguous!
```

```
## NULL
```

- Use $ with care (and not in regular code)

Lists

- Assignment

```
mylist
```

```
## $myvect
## [1]  1  5 -3
##
## $myy
## [1] 1 2
##
## $myfact
## [1] "a" "b"
```

Lists


```r
mylist[1:2] <- list(c(1,2,3), c(1,3)); mylist
```

```
## $myvect
## [1] 1 2 3
##
## $myy
## [1] 1 3
##
## $myfact
## [1] "a" "b"
```

Lists


```
mylist[1] <- c(1,2); mylist # attempt to replace a sublist
```

```
## Warning in mylist[1] <- c(1, 2): number of items
## to replace is not a multiple of replacement length
## $myvect
## [1] 1
##
## $myy
## [1] 1 3
##
## $myfact
## [1] "a" "b"
```

Lists


```r
mylist[1] <- list(c(1,2)); mylist
```

```
## $myvect
## [1] 1 2
##
## $myy
## [1] 1 3
##
## $myfact
## [1] "a" "b"
```

Lists


```
mylist[[1]] <- c(1,3); mylist # replace the first element
```

```
## $myvect
## [1] 1 3
##
## $myy
## [1] 1 3
##
## $myfact
## [1] "a" "b"
```

Lists


```
mylist$myfact <- c("c", "d"); mylist
```

```
## $myvect
## [1] 1 3
##
## $myy
## [1] 1 3
##
## $myfact
## [1] "c" "d"
```

Lists


```
mylist$mynewelement <- c("new"); mylist # add an element
```

```
## $myvect
## [1] 1 3
##
## $myy
## [1] 1 3
##
## $myfact
## [1] "c" "d"
##
## $mynewelement
## [1] "new"
```

Lists


```
mylist$mynewelement <- NULL; mylist # remove an element
```

```
## $myvect
## [1] 1 3
##
## $myy
## [1] 1 3
##
## $myfact
## [1] "c" "d"
```

Lists

- Attributes

```
length(mylist)
```

```
## [1] 3
```

```
names(mylist)
```

```
## [1] "myvect" "myy"    "myfact"
```

Lists

- List of lists

```
lists <- list(mylist, alist = list(c(2,3)))
                                # a named list of 2 elements:
                                # a named list of 3 elements
lists                           # and a list of 1 element
```

```
## [[1]]
## [[1]]$myvect
## [1] 1 3
##
## [[1]]$myy
## [1] 1 3
##
## [[1]]$myfact
## [1] "c" "d"
##
##
```

Lists


```
## $alist
## $alist[[1]]
## [1] 2 3
```

```r
length(lists)
```

```
## [1] 2
```

Lists


```
              # list of 1 element
lists[1]      # a named list of 3 elements
```

```
## [[1]]
## [[1]]$myvect
## [1] 1 3
##
## [[1]]$myy
## [1] 1 3
##
## [[1]]$myfact
## [1] "c" "d"
```

Lists


```
lists[[1]]  # a named list of 3 elements
```

```
## $myvect
## [1] 1 3
##
## $myy
## [1] 1 3
##
## $myfact
## [1] "c" "d"
```

Lists


```r
lists$alist # a list of 1 element
```

```
## [[1]]
## [1] 2 3
```

Lists


```
lists[[1]][1:2] # a named list of 2 elements
```

```
## $myvect
## [1] 1 3
##
## $myy
## [1] 1 3
```

```
lists[[1]][[2]] # a single element
```

```
## [1] 1 3
```

```
lists[[1]]$myvect
```

```
## [1] 1 3
```

```
lists$alist[[1]]
```

```
## [1] 2 3
```

Data Frames

- **data.frame**: named list of vector of same size (cf columnar database with matrix flavor)
- Creation

```
v1 <- c("F","M","M","F","M"); v2 <- c(27,54,34,21,57)
v3 <- c(177,183,190,158,178)
data <- data.frame(sex=v1, age=v2, height=v3); data
```

```
##   sex age height
## 1   F  27    177
## 2   M  54    183
## 3   M  34    190
## 4   F  21    158
## 5   M  57    178
```

Data Frames

- Concatenation

```
weight <- c(60,65,89,45, 68)
cbind(data, weight = weight)
```

```
##   sex age height weight
## 1   F  27    177     60
## 2   M  54    183     65
## 3   M  34    190     89
## 4   F  21    158     45
## 5   M  57    178     68
```

Data Frames


```r
data2 <- data.frame(age = 44, sex = "F", height = 180)
rbind(data, data2)
```

```
##   sex age height
## 1   F  27    177
## 2   M  54    183
## 3   M  34    190
## 4   F  21    158
## 5   M  57    178
## 6   F  44    180
```

Data Frames

- (Sub)data.frame: result is a data.frame

```
data[1,]
```

```
##   sex age height
## 1   F  27    177
```

```
data[1,1:2]
```

```
##   sex age
## 1   F  27
```

```
data[1:2,]
```

```
##   sex age height
## 1   F  27    177
## 2   M  54    183
```

```
data[1:2, c("sex", "height")]
```

Data Frames


```
##   sex height
## 1   F    177
## 2   M    183
```

```r
data[, "sex", drop = FALSE] # R cleverness
```

```
##   sex
## 1   F
## 2   M
## 3   M
## 4   F
## 5   M
```

Data Frames


```r
data[2, "height", drop = FALSE] # R cleverness
```

```
##   height
## 2    183
```

- Extraction: result is a vector

```r
data[, "sex"] # implicit simplification!
```

```
## [1] F M M F M
## Levels: F M
```

```r
data[2, "height"] # implicit simplification!
```

```
## [1] 183
```

Data Frames


```r
data[["sex"]]
```

```
## [1] F M M F M
## Levels: F M
```

```r
data[[1]]
```

```
## [1] F M M F M
## Levels: F M
```

```r
data[["height"]][2]
```

```
## [1] 183
```

```r
data$heigh # partial matching
```

```
## [1] 177 183 190 158 178
```

Data Frames

- Assignment

```
data[1,"sex"] <- "M"
data
```

```
##   sex age height
## 1   M  27    177
## 2   M  54    183
## 3   M  34    190
## 4   F  21    158
## 5   M  57    178
```

Data Frames


```r
data[1, ] <- data.frame(sex = factor("F"),
                        age = 44,
                        height = 180) # order matters
data
```

```
##   sex age height
## 1   F  44    180
## 2   M  54    183
## 3   M  34    190
## 4   F  21    158
## 5   M  57    178
```

Data Frames


```
data$heightm <- data[["height"]] /  100; data
```

```
##   sex age height heightm
## 1   F  44    180    1.80
## 2   M  54    183    1.83
## 3   M  34    190    1.90
## 4   F  21    158    1.58
## 5   M  57    178    1.78
```

Data Frames

- Properties

```r
dim(data)
```

```
## [1] 5 4
```

```r
colnames(data)
```

```
## [1] "sex"      "age"      "height"   "heightm"
```

```r
rownames(data) # not mandatory...
```

```
## [1] "1" "2" "3" "4" "5"
```

Data Frames

- Summary

```r
summary(data) # generic function
```

```
##   sex         age            height     
##   F:2    Min.    :21    Min.    :158.0  
##   M:3    1st Qu.:34    1st Qu.:178.0  
##          Median :44    Median :180.0  
##          Mean    :42    Mean    :177.8  
##          3rd Qu.:54    3rd Qu.:183.0  
##          Max.    :57    Max.    :190.0  
##      heightm     
##   Min.    :1.580  
##   1st Qu.:1.780  
##   Median :1.800  
##   Mean    :1.778  
##   3rd Qu.:1.830  
##   Max.    :1.900
```

Dplyr Basics

- **tibble** and **dplyr** provide a modern framework for data.frame.
- **tibble**: stricter version of data.frame

```
data <- as_tibble(data)
data
```

```
## # A tibble: 5 x 4
##   sex     age height heightm
##   <fct> <dbl>  <dbl>   <dbl>
## 1 F        44    180    1.8
## 2 M        54    183    1.83
## 3 M        34    190    1.9
## 4 F        21    158    1.58
## 5 M        57    178    1.78
```

Dplyr Basics


```
data[,1]
```

```
## # A tibble: 5 x 1
##   sex
##   <fct>
## 1 F
## 2 M
## 3 M
## 4 F
## 5 M
```

Dplyr Basics

- **dplyr**: set of *verbs* to work on (grouped) tibble

```r
glimpse(data)
```

```
## Observations: 5
## Variables: 4
## $ sex     <fct> F, M, M, F, M
## $ age     <dbl> 44, 54, 34, 21, 57
## $ height  <dbl> 180, 183, 190, 158, 178
## $ heightm <dbl> 1.80, 1.83, 1.90, 1.58, 1.78
```

Dplyr Basics


```
data %>% group_by(sex) %>%
  summarize(mean_height = mean(height),
            max_age = max(age))
```

```
## # A tibble: 2 x 3
##   sex   mean_height max_age
##   <fct>       <dbl>   <dbl>
## 1 F            169       44
## 2 M            184.      57
```

Missing Values

- Missing values are handled in **R** with the `NA` symbols

```
x <- c(1, 3, 2, 7, -3, NA)
is.na(x)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE  TRUE
```

```
which(is.na(x))
```

```
## [1] 6
```

```
mean(x, na.rm = TRUE)
```

```
## [1] 2
```

Missing Values


```
mean(x[!is.na(x)])
```

```
## [1] 2
```

```
x[is.na(x)]=0
data[1,4]=NA
is.na(data)
```

```
##        sex   age height heightm
## [1,] FALSE FALSE  FALSE    TRUE
## [2,] FALSE FALSE  FALSE   FALSE
## [3,] FALSE FALSE  FALSE   FALSE
## [4,] FALSE FALSE  FALSE   FALSE
## [5,] FALSE FALSE  FALSE   FALSE
```

Missing Values


```r
which(is.na(data))
```

```
## [1] 16
```

```r
which(is.na(data), arr.ind = TRUE)
```

```
##      row col
## [1,]   1   4
```

Factors

- Qualitative variable: **factor** (or **ordered**)
- Function **as.factor()**:

```
f <- c("A", "B", "A")
f <- as.factor(f)
f
```

```
## [1] A B A
## Levels: A B
```

```
levels(f) # levels
```

```
## [1] "A" "B"
```

```
nlevels(f) # number of levels
```

```
## [1] 2
```

```
relevel(f, ref = "B") # reference level
```

```
## [1] A B A
## Levels: B A
```

Factors


```
#(cf variable coding)
```

- Quantization in class with **cut()**

```
x <- 1:10
f <- cut(x, breaks=c(1,2,4,10),include.lowest=TRUE)
f
```

```
##  [1] [1,2]  [1,2]  (2,4]  (2,4]  (4,10] (4,10]
##  [7] (4,10] (4,10] (4,10] (4,10]
## Levels: [1,2] (2,4] (4,10]
```

Factors

- Fusion/renaming of **levels()**

```
levels(f)
```

```
## [1] "[1,2]"  "(2,4]"  "(4,10]"
```

```
levels(f) <- c("[1,4]", "[1,4]", "(4,10]")
f
```

```
##  [1] [1,4]  [1,4]  [1,4]  [1,4]  (4,10] (4,10]
##  [7] (4,10] (4,10] (4,10] (4,10]
## Levels: [1,4] (4,10]
```

- **forcats** package. . .

Import / Export

- Numerous import/export function:
  - CSV: `read.table`, package **readr**
  - Excel: package **readxl**
  - SPSS, Stata, SAS: package **haven**, **foreign**
  - Python: **feather**
  - ...
- Database connectors:
  - SQL: **RMySQL**, **ROracle**, **RPostgreSQL**, **RSQLite**, **RSQLServer** (MS SQL Server),... (**DBI**)
  - Generic: **RJDBC** (generic connection via java), **RODBC**, **odbc**...
  - NoSql: **mongolite**, **sergeant** (drill)...

Example of DB Connectors

- Usual syntax
  - open a connection with `dbConnect()`
  - request with `dbGetQuery()`
  - close the connection with `dbDisconnect()`

```
library(DBI)
conn <- dbConnect("RMySQL", host = "myserver",
                  port = 123, dbnam = "database",
                  user = "eric", password = "aqw")
res <- dbGetQuery (conn, "SELECT * FROM matable")
dbDisconnect(conn)
```

- **dbplyr**:
  - allows to use a remote table as if it was local
  - `tbl(conn,...)` + `dplyr` command: create an action plan.
  - `collect()` to retrieve the data in R / `compute()` to store it in the DB

RData and RDS

- **RData**:
  - `save()`/`load()` allows to write/read one or more objects with their names
  - Compressed format with extension **.RData**
  - Beware of name clash!

```
x <- 1:10; l <- list(a = 1, b = LETTERS[1:3])
save(x, l, file = "data.RData")
load(file = "data.RData")
```

- **RDS**:
  - `saveRDS()`/`readRDS()` allows to write/read a single object.
  - Compressed format with extension **.RDS**
  - No name issue

```
x <- 1:10
saveRDS(x, file = "data.RDS")
y <- readRDS(file = "data.RDS")
```

Outline

if / else


```
if (condition1) {
  print("condition1 is true")
} else if (condition2) {
  print("condition1 is false but condition2 is true")
} else {
  print("both conditions are false")
}
```

- A condition should return a (**and only one**) logical value (TRUE/FALSE)
  - A numerical value is considered as TRUE except if it is equal to 0

ifelse


```
ifelse(condition_vector, true_vector, false_vector)
```

- For each *i*, look at `condition_vector[i]` and output accordingly `true_vector[i]` or `false_vector[i]`

```
x <- 1:2
ifelse(x%%2 == 0, 0, x)      #> [1] 1 0
```

- **Rk:** `if_else` from **dplyr** better handles some corner cases (attributes/NA)

switch

- A different conditional branching:

```
res <-  switch(value,
         case1 = result1,
         case2 = result2,
         case3 = result3,
         otherwise)
```

```
fonction <- "mean"
x <- rnorm(100)
res <-  switch(fonction,
         mean = mean(x),
         median = median(x),
         sum = sum(x))
res
```

```
## [1] 0.150541
```

Logical Operators

- `==`, `!=`, `>`, `<`, `>=`, `<=`

```r
x <- 1

x == 1  # TRUE
x != 1  # FALSE
x < 1   # FALSE

vx <- c(1, 2)
vx != 1 # FALSE TRUE
```

Logical Operators

- `any` : TRUE if at least one condition is TRUE

```
x <- c(1:10)

any(x == 10) # TRUE
any(x > 10)  # FALSE
```

- `all` : TRUE if all the conditions are TRUE

```
x <- c(1:10)

all(x <= 10) # TRUE
all(x < 10)  # FALSE
```

Logical Operators

- `%in%` : TRUE for any elements belonging to the left-hand *set*

```
x <- "Rennes"
x %in% c("Rennes", "Brest") # TRUE

x <- c("Rennes", "Paris")
x %in% c("Rennes", "Brest") # TRUE FALSE
```

- `is.vector`, `is.data.frame`, `is.list`, ...

```
x <- c(1:10)

is.vector(x) # TRUE
```

Logical Operators

- **!** : Negation of a condition

```
x <- 1
y <- 10

(x == 1 & y == 10)    # TRUE
!(x == 1 & y == 10)   # FALSE
```

- & : (vectorwise) AND operator (TRUE if and only if both conditions are TRUE)

```
(x == 1 & y == 10)    # TRUE
(x == 1 & y == 9)   # FALSE
```

Logical Operators

- |: (vectorwise) OR operator (TRUE if and only if at least one condition is TRUE)

```
(x == 1 || y == 10)    # TRUE
(x == 1 || y == 9)     # TRUE
(x == 2 || y == 9)     # FALSE
```

- && and ||: left to right examination of only the first element of each condition!

- xor : exclusive ‘OR’ (TRUE if and only if only one condition is TRUE)

```
xor(TRUE, FALSE) # TRUE
xor(TRUE, TRUE)  # FALSE
```

Loops

- May be inefficient:
  - Call overhead
  - Memory allocation issues
- Use with care in **R**
- Favor **vectorization** and **apply family** (more on this later)

for

- Loop along elements in a vector

```
for(variable in elements){
  ...
}
```

```
for (lettre in LETTERS[1:2]) {
  print(lettre)
}
```

```
## [1] "A"
## [1] "B"
```

while

- Repeat **while** a condition is true
  - if the condition is FALSE at the beginning, nothing happens
  - if it is always TRUE, endless loop!

```
while(condition){
  ...
}
```

```
x <- 1
while(x < 4){
  print(x)
  x <- x+1
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

repeat

- Repeat **until** explicit exit:
    - loop is entered once at least.
    - Explicit `break` to exit

```
repeat{
  ...
  if(condition) break
}
```

```
x <- 1
repeat{
  x <- x+1
  if(x == 3){
    print("x is equal to 3, let's stop.")
    break
  }
}
```

```
## [1] "x is equal to 3, let's stop."
```

break and next

- `break` : Explicit (and immediate) exit of a `for`, `while` or `repeat` loop
- `next` : Explicit (and immediate) jump to the next `for`, `while` or `repeat` iteration

```
for(i in 1:3){
  if(i%%2 != 0) {
    next
  }
  print(i)
}
```

```
## [1] 2
```

Outline

1. R
2. Basics
3. Condition and Loop
4. **Vectorization and Apply Family**
5. Functions
6. Coding Style
7. More on R

Loop and **R**

- R in an **interpreted** language
  - function call overhead
- Lack of efficiency of **for** loop!
- **R** mantra: avoid them!
- Think
  - vectorization
  - map/reduce

Vectorization

The fundamental idea behind array programming (vectorization) is that operations apply at once to an entire set of values. This makes it a high-level programming model as it allows the programmer to think and operate on whole aggregates of data, without having to resort to explicit loops of individual scalar operations. (Wikipedia)

- Many *loop* computation can be *vectorized* using dedicated operators:
  - vectorwise operator, matrixwise operator
  - data.framewise operator
- Gain of performance due to
  - inner use of **C**, **C++**, **Fortran** code
  - inner use of optimized libraries (**BLAS**, **LAPACK**, **FFTW**. . .)

Toy Example: Summing Two Vectors


```r
x <- rnorm(100000)
y <- rnorm(100000)
res <- rep(0, 100000)

# sum with a loop
system.time(for(i in 1:100000){
  res[i] <- x[i] + y[i]
})
```

```
##    user  system elapsed
##   0.010   0.000   0.012
```

```r
# vectorized sum
system.time(res2 <- x + y)
```

```
##    user  system elapsed
##       0       0       0
```

```r
identical(res, res2)
```

```
## [1] TRUE
```

Vectorization Examples

- vector/matrix operation

```
x <- matrix(ncol = 2, nrow = 2, 1)
y <- matrix(ncol = 2, nrow = 2, 2)
z <- x * y
z
```

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

- data.frame operation

```
data <- data.frame(x = 1:10, y = 100:109)
data$z <- data$x + data$y
head(data, n = 2)
```

```
##   x   y   z
## 1 1 100 101
## 2 2 101 103
```

Apply Family

- Often loop correspond to the application of the same independent process to several pieces.
- **Apply family** can replace efficiently the `for` loops
  - `lapply`: apply a function to each element of a list and output a list
  - `sapply`/`vapply`: apply a function to each element of a list and output a *structured* value
  - `mapply`: apply a function element wise to several vectors/lists.
  - `apply`: apply a function to each line/column of a matrix (or an array) or data.frame
  - `rapply`: recursive apply.
- **purrr** provides a unified interface with the `map` function family.

lapply, sapply


```
lapply(X, FUN, ...)

sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
```

- `X` : a vector or a list
- `FUN` : the function to apply to the elements of X
- `...` : supplementary arguments of the function
- `simplify` : Boolean or character specifying if or how to simplify the list result
- `USE.NAMES` : Boolean. If `X` is named, should we reuse the names in the result

Examples

- How to compute the mean for each elements?
- the data :

```
x <- list(a = 1:20, b = rnorm(17))
```

```
## $a
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
## [16] 16 17 18 19 20
##
## $b
##  [1]  0.28978803  0.51454331  1.04304124
##  [4]  1.04505491  0.28488767 -1.24231998
##  [7] -1.67973769  0.66495695 -0.55458026
## [10] -1.15120824  0.05729449  0.32591031
## [13] -0.96568433 -0.27885527 -1.26211651
## [16]  1.57334570 -1.46452653
```

Examples

- `lapply` returns a list

```
lapply(x, FUN = mean)
```

```
## $a
## [1] 10.5
##
## $b
## [1] -0.164718
```

- `sapply` simplify the results in a vector

```
sapply(x, FUN = mean)
```

```
##         a         b
## 10.500000 -0.164718
```

Examples

- How to compute the defaults quantiles, i.e. 5 values by element?
- `lapply` returns a list

```
lapply(x, FUN = quantile)
```

```
## $a
##    0%   25%   50%   75%  100%
##  1.00  5.75 10.50 15.25 20.00
##
## $b
##          0%         25%         50%         75%
## -1.67973769 -1.15120824  0.05729449  0.51454331
##        100%
##  1.57334570
```

Examples

- `sapply` simplifies the result into a matrix

```
sapply(x, FUN = quantile)
```

```
##            a           b
## 0%      1.00 -1.67973769
## 25%     5.75 -1.15120824
## 50%    10.50  0.05729449
## 75%    15.25  0.51454331
## 100%   20.00  1.57334570
```

Examples

- What if the number of returned element varies?

```
la <- lapply(x, FUN = function(elm) elm)
sa <- sapply(x, FUN = function(elm) elm)

identical(la, sa)
```

```
## [1] TRUE
```

apply

```
apply(X, MARGIN, FUN, ...)
```

- `X` : a matrix or a data.frame (or an array)
- `MARGIN` : a vector of integer specifying the *dimension(s)* that will be used to slice the data (1: line, 2: column,…)
- `FUN` : the function to apply
- `...` :supplementary arguments to pass to the function

```
x <- cbind(x1 = 3, x2 = c(NA, 4:1, 2:6))
apply(x, 2, mean)                        # mean by colum
```

```
## x1 x2
##  3 NA
```

```
apply(x, 2, mean, na.rm = TRUE)     # with a supp. argument
```

```
##       x1       x2
## 3.000000 3.333333
```

purr, map, reduce

- **tidyverse** version of `apply`
- `map_*` functions that specify the output format.
- *Pure* functional approach / Map/Reduce framework.

```
x <- list(a = 1:20, b = rnorm(25))
map(x, ~ quantile(., na.rm = TRUE))
```

```
## $a
##    0%   25%   50%   75%  100%
##  1.00  5.75 10.50 15.25 20.00
##
## $b
##         0%        25%        50%        75%
## -1.1962019 -0.4309738 -0.1636642  0.6539324
##       100%
##  1.6072388
```

purr, map, reduce


```r
map_df(x, ~ as_tibble(t(quantile(., na.rm = TRUE))))
```

```
## # A tibble: 2 x 5
##    `0%`  `25%`  `50%`  `75%` `100%`
##   <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
## 1  1      5.75 10.5   15.2    20
## 2 -1.20 -0.431 -0.164  0.654   1.61
```

purr, map, reduce


```r
result <- map2_dbl(c(100,1000), c(10,0),
                   ~ mean(rnorm(.x, .y)))
result
```

```
## [1]  9.89447394 -0.02288114
```

```r
reduce(result, `+`)
```

```
## [1] 9.871593
```

- **Rk**! `Map` and `Reduce` exist in base **R**

(Embarassingly) Parallel Computation

- `lapply` type functions are ideal for parallel computation:
  - independent computation on each element of the list
  - very simple final combination
- Several parallel (os dependent. . .) backends are available:
  - **parallel**: `mclapply` (multicore), `parLapply` (cluster)
  - **parallelMap**: `parallelLapply` (local, multicore, socket, MPI, batchjobs)
  - **future**: `future_lapply` (local, multicore, multiprocess, socket, MPI, batchjobs. . .)

(Embarassingly) Parallel Computation

- **future** example:

```r
library(future.apply)
```

```
## Loading required package: future
```

```r
plan(multiprocess)
```

```
## Warning: [ONE-TIME WARNING] Forked processing
## ('multicore') is disabled in future (>=
## 1.13.0) when running R from RStudio, because
## it is considered unstable. Because of
## this, plan("multicore") will fall back to
## plan("sequential"), and plan("multiprocess")
## will fall back to plan("multisession") - not
## plan("multicore") as in the past. For more
## details, how to control forked processing or
## not, and how to silence this warning in future R
## sessions, see ?future::supportsMulticore
```

(Embarassingly) Parallel Computation


```r
future_lapply(x, FUN = mean)
```

```
## $a
## [1] 10.5
##
## $b
## [1] 0.04134791
```

(Embarassingly) Parallel Computation

- **foreach** proposes a second approach based on a sequential %do% loop

```r
library(foreach)
```

```
##
## Attaching package: 'foreach'
## The following objects are masked from 'package:purrr':
##
##     accumulate, when
```

```r
foreach(i = x) %do% { mean(i) }
```

```
## [[1]]
## [1] 10.5
##
## [[2]]
## [1] 0.04134791
```

(Embarassingly) Parallel Computation

- This becomes parallel by replacing %do% by %dopar% and specifying a **do*** backend (**doParallel**, **doMC**, **doFuture**...)

```
library(doFuture)
```

```
## Loading required package: globals
## Loading required package: iterators
## Loading required package: parallel
```

```
registerDoFuture()
foreach(i = x) %dopar% { mean(i) }
```

```
## [[1]]
## [1] 10.5
##
## [[2]]
## [1] 0.04134791
```

Outline

Function definition

- Syntax:

```
fun <- function(args) expression
```

with

- **fun**: function name
- **args**: list of (named) arguments separated by comma (`formals(fun)`)
- **expression**: the body of the function, either a single expression or several between braces (`body(fun)`)

```
test <- function(x) x^2
test                # function(x) x^2
formals(test)       # $x
body(test)          # x^2
environment(test) # <environment: R_GlobalEnv>
```

Function definition

- A function belongs to an environment (a workspace containing variables), most of the time either a package of the global environment **GlobalEnv**. (`environment(fun)`)

Arguments

- **Default value**
  - set with a = in the function definition,
  - optional variable (beware of the order)

```r
test <- function(x = 2, y){
  x + y
}
test(y = 2)
```

```
## [1] 4
```

```r
test(x = 2, y = 10)
```

```
## [1] 12
```

Arguments


```r
test(2, 10)
```

```
## [1] 12
```

```r
# test(2) fails!
```

- Some convenient functions to deals with args:
  - `missing(arg)` : return `TRUE` if `arg` is not defined
  - `match.arg()` : return a partial match
  - `typeof(arg)`, `class(arg)`, `is.vector()`, `is.data.frame()`, ...

```r
match.arg("mea", c("mean", "sum", "median"))  # "mean"
class(10)                                     # "numeric"
```

Argument evaluation

- An argument can use a previous one

```
# Simple case
test <- function(x, y = x + 10){
  x + y
}
test(5)  # 20
```

Argument evaluation


```
# More complicated one
test <- function(x,
                 fun = if(class(x) %in% c("numeric",
                                          "integer")){
                   "sum"
                 } else {
                   "length"
                 }){

  do.call(fun, list(x = x))

}

test(1:10)              #55
test(LETTERS[1:10])     #10
```

Argument evaluation

- Lazy evaluation: arguments are evaluated only when used inside the function!

Argument evaluation


```
f <- function(x) {
  10
}
f(stop("This is an error!"))

# the function returns 10 although the evaluation of its a
# 10

# use of force
f <- function(x) {
  force(x)
  10
}
f(stop("This is an error!"))

# Error: This is an error!
```

Understanding the . . .

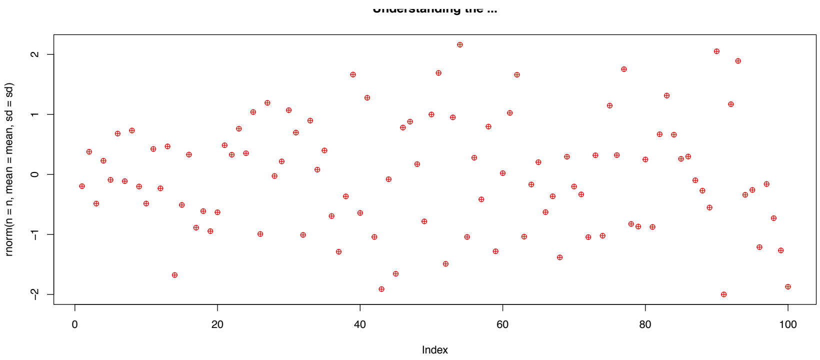
- . . .: corresponds to arguments not explicitly defined.
- Often use as argument of another function.
- Explicit extraction with *list(. . .)*

```
viewdot <- function(arg, ...){
  list(...)
}
viewdot(arg = 1, x = 2, name = "name")

#$x
#[1] 2
#
#$name
#[1] "name"
```

Understanding the . . .

```
rnormPlot <- function(n, mean = 0, sd = 1, ...){
  plot(rnorm(n = n, mean = mean, sd = sd), ...)
}
rnormPlot(n = 100, main = "Understanding the ...", col = "r
```



Returning a value

- By default, a function returns its last value:

```
test <- function(x, y = 2){
  x + y
}
test(2)
```

```
## [1] 4
```

```
sum <- test(x = 2, y = 2)
sum
```

```
## [1] 4
```

- Explicit return before the end with `return()`
- **No need** to use `return()` on the last line.
- To return several values : use a structure (named list, data.frame)
- Use `invisible()` to not return anything.

Explicit return

- Use of return()

```
test <- function(x, y = 2){
  if(y == 0){
    return(x)
  }
  x + y
}
test(2)
```

```
## [1] 4
```

invisible return

- The `invisible()` function > "This function can be useful when it is desired to have functions return values which can be assigned, but which do not print when they are not assigned"

```
test <- function(x, y = 2){
  x + y
  invisible()
}
test(2)  # no print on console
res <- test(2)
res     # and NULL result
```

```
## NULL
```

```
test <- function(x, y = 2){
  invisible(x + y)
}
test(2)  # no print on console
res <- test(2)
res      # but a result !
```

Several outputs

- Named list output:

```
test <- function(x, y = 2){
  list(x = x, y = y)
}
test(2)
```

```
## $x
## [1] 2
##
## $y
## [1] 2
```

- Data.frame output:

```
test <- function(x, y = 2){
  data.frame(x = x, y = y)
}
test(2)
```

```
##   x y
```

Local and global variables

- A variable defined in a function is **local** :
  - it will not exist outside the function
  - it will not replace a variable defined outside.

```
x <- 100
test <- function(x, y){
  x <- x + y
  x
}

# the function returns 10
test(5, 5)
```

```
## [1] 10
```

```
# and x is still equal to 100
x
```

```
## [1] 100
```

Variables and environments

- Non existing variables: basic scenario

```
test <- function(x){
  x + z
}

# Erreur, z n'existe pas
test(5)
```

```
##      [,1] [,2]
## [1,]    7    7
## [2,]    7    7
```

```
#> Error in test(5) : object 'z' not found
```

Variables and environments

- Beware, use the value found in the fist parent environment where the variable is defined!

```r
# If z exists in a parent environment, the value will be us
z <- 5
test(5)
```

```
## [1] 10
```

```r
#> 10
```

- Source of errors: it is better to pass **all** the arguments in parameter and not to rely on this mechanism.

Anonymous functions

- Function without any name (lambda calculus)
  - short function
  - used only once

```
f <- function(x){
  x + 1
}

res1 <- sapply(1:10, f)

res2 <- sapply(1:10, function(x) x + 1)

res1
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

```
res2
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

Communication issues

- Importance of handling possible issues:
  - wrong argument type,
  - non existing files,
  - missing values, infinite values...
- Three possible ways to communicate:
  - `stop()` for a **fatal** error that terminates the execution
  - `warning()` to signal a **potential** issue that should be checked
  - `message()` to communicate something that is **not an issue**.

Communicating Issues


```
test <- function(x){
  # For a better error handling
  if(missing(x)){
    stop("x is missing. Please enter a valid argument")
  }
  if(!class(x) %in% c("numeric", "integer")){
    x <- as.numeric(as.character(x))
    warning("x is coerced to numeric")
  }
  message("compute x*2")
  x*2
}
try(test())
```

```
## Error in test() : x is missing. Please enter a valid arg
```

```
test("5")
```

```
## Warning in test("5"): x is coerced to numeric
## compute x*2
```

Handling Errors

- **R** stops when it encounters an **unexpected** errors!
- Two possible ways to cope with **expected** errors:
  - `try()` to continue the execution nevertheless
  - `tryCatch()`/`withCallingHandlers()` to use a specific error handling code

```
test <- sapply(list(1:5,"a", 6:10), log)
#>Error in FUN(X[[2L]], ...) :
#  non-numeric argument to mathematical function
```

```
test <- sapply(list(1:5,"a", 6:10), function(x) try(log(x)
test
```

```
## [[1]]
## [1] 0.0000000 0.6931472 1.0986123 1.3862944
## [5] 1.6094379
##
## [[2]]
## [1] "Error in log(x) : non-numeric argument to mathemat
## attr(,"class")
```

Documentation!

- A good documentation is important:
  - to explain to the users how to use the functions
  - to help the developers (included yourself) to enhance them
- A solution: self documenting code with **doxygen**:
  - easy to use solution used in several programming language
  - available in _R___ with roxygen2
  - use specific syntax starting with `#'` just after the function header

Roxygen tags

- @param : for the arguments
- @return : for the output
- @examples : for the examples

{

```r
#' Name of the function
#'
#' short description
#' on a few lines
#'
#' @param name : Character. Family name
#' @param surname : Character. Given name
#'
#' @return : Character. Identity
#'
#' @examples
#' # Example of use
#' identifier("Thieurmel", "Benoit")
identifier <- function(name, surname){
  paste0("Name :", name, ", Surname : ", surname)
```

Debuging


```
> f(10)
Error in "a" + d : non-numeric argument to binary operator
                                                    Hide Traceback
                                                    Rerun with Debug
 4 i(c) at exceptions-example.R#3
 3 h(b) at exceptions-example.R#2
 2 g(a) at exceptions-example.R#1
 1 f(10)
```

```
traceback()
# 4: i(c) at exceptions-example.R#3
# 3: h(b) at exceptions-example.R#2
# 2: g(a) at exceptions-example.R#1
# 1: f(10)
```

- Old school : use `print()` in the function.
- Debugger: use R debugger!
  - `traceback()`
  - RStudio GUI
- Profiler:
  - **lineprof** package
  - RStudio GUI

Outline

Coding Style?

Freely inspired from Style Guide, by Hadley Wickham

- Good coding style is important :
  - ease the reading and the comprehension of a code
  - favor a collaborative development
  - often associated to an ease of use (standardization)
- No perfect style!
  - use you own (or the one of the project),
  - (try to) keep it.

```
# hard to read
aze=data.frame(cole=rnorm(1000),refdzf=LETTERS[1:2]);ff=
  lapply(split(aze$cole,aze$refdzf),
         function(x){mean(x)});ff
```

Coding Style


```r
# Much better!
data <- data.frame(value = rnorm(1000),
                   group = LETTERS[1:2])
mean.group <- lapply(
  split(data$value, data$group),
  function(x){
    mean(x)
  })
mean.group
```

Files

- Use **explicit** names and end them with `.R`.

```
# Good                 # Bad           0-download.R
 modelisation.R         toto.r          1-parse.R
```

- Add a number prefix to indicate an order in a sequence

```
00-download.R
10-parse.R
```

Variables and Functions

- **Short** and **explicit** names:
  - lowercase with _ to separate words (other convention exist)
  - avoid . to avoid interaction with some object oriented structure in **R** (or **Java**)
  - variables should be **names**, functions should be **verbs**
  - avoid special characters...

```
# Good
day_one
day_1
```

```
# Bad
first_day_of_the_month
DayOne
mean <- function(x) sum(x)
```

Spacing

- Add spaces **around** most operators (=, +, -, <-, etc.), **in particular** in function calls.
- Add a space **after** a comma, **not before**!
- Add a space to separate the content in a parenthesis, **except in a function call**.

```
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)

# Bad
average<-mean(feet/12+inches,na.rm=TRUE)
```

- No space **around** `:`, `::` and `:::`

```
# Good                         # Bad
x <- 1:10                      x <- 1 : 10
base::get                      base :: get

if (debug) do(x)               if(debug)do(x)
plot(x, y)                     plot (x, y)
```

Braces and Indentation

- An opening brace should **always** be followed by a line break.
- A closing brace should be followed by a line break, except in the case of `else` which should be on the same line.
- Code inside should be indented.

```
# Good
if (y == 0) {
  log(x)
} else {
  y ^ x
}
```

```
# Bad
if (y == 0) {
  log(x)
}
else{ y ^ x}
```

Braces and Indentation

- Indent the code with two spaces **RStudio shortcut: Ctrl+A, Ctrl+I**
- Exception for function definition:

```
long_function_name <- function(a = "a long argument",
                               b = "another argument",
                               c = "another long argument")
  # As usual code is indented by two spaces.
}
```

Assignement

- Use <-, and **never** =, to assign an object.

```
# Good
x <- 5
# Bad
x = 5
```

- Use =, and **never** <-, in named list.

Comments

- Comment the code to help the reader!

```
# Load data ---------------------------


# Plot data ---------------------------
```

- Use a literate scripting/programming style (**R Markdown** / **notebook**)

Outline

More on **R** (Doc)

- The R Manuals : https://cran.r-project.org/manuals.html
- R Contributed Documentation : https://cran.r-project.org/other-docs.html
- How-to go parallel in R - basics + tips : http://gforge.se/2015/02/how-to-go-parallel-in-r-basics-tips/
- State of the Art in Parallel Computing with R : http://www.jstatsoft.org/v31/i01/paper
- R tutorial on the Apply family of functions : http://www.r-bloggers.com/r-tutorial-on-the-apply-family-of-functions/
- A Tutorial on Loops in R - Usage and Alternatives : http://blog.datacamp.com/tutorial-on-loops-in-r/

More on **R** (Books)

- Hands-On Programming with R Write Your Own Functions and Simulations by Garrett Grolemund: O'Reilly (2014)
- Advanced R by Hadley Wickham : Chapman & Hall's R (2014) / http://adv-r.had.co.nz/
- R packages by Hadley Wickham : O'Reilly (2015) / http://r-pkgs.had.co.nz/
- R for Data Science by Hadley Wickham and Garrett Grolemund: O'Reilly (2017) / http://r4ds.had.co.nz/
- Extending R by John Chambers: CRC Press (2016)

More on **R** (Blogs)

- R-Bloggers: http://www.r-bloggers.com/
- Rweekly: http://www.rweekly.org/