R Initiation

B.Thieurmel - benoit.thieurmel@datastorm.fr

Enedis - 19 & 22/11/2018

Introduction

Le logiciel R

R (http://cran.r-project.org/) est un logiciel de Statistique distribué gratuitement par le **CRAN** créé dans les années 90 dans l'esprit de S :

- dédié à l'analyse statistique et à la visualisation
- environ 80% du temps de l'analyse est dédié à la préparation des données... donc **R** sert aussi à la manipulation des données

Structure

- disponible sous de nombreux systèmes d'exploitation
- composé d'un socle et de bibliothèques de fonctions thématiques regroupées sous le nom de **package**
- connectables avec (tous...) les autres langages : *C*, *Fortran*, *Java*, *Python*, *Javacript*, *C++*, ...
- et (toutes...) les bases de données : *MySQL*, *Postgresql*, *Oracle*, *MS sql*, *mongodb*, *Hadoop*, ...
- possible d'appeler **R** depuis *Matlab*, *Excel*, *SAS*, *SPSS*, ...

Les packages

- R a été pensé comme un langage ouvert et modulaire
- Quasiment tous les chercheurs l'utilisent et donc les nouvelles méthodes sont souvent implémentées
- Le passage recherche/industrie est de plus en plus rapide.

Il est fort probable qu'une autre personne que vous ait déjà rencontré le même problème que le votre (packages existants, de discussions R-bloggers, forum, ...)

Les dépôts «officiels» :

- Le CRAN (https://cran.r-project.org/) : plus de 6000 packages déposés et maintenus. Recherches thématiques avec les Task Views (https://cran.r-project.org/web/views/)
- Bioconductor (https://www.bioconductor.org/). Plus de 1200 packages. Populations/analyses fortement liées à la biologie

Les dépôts «personnels» : *GitHub*, *BitBucket*, ... Packages du CRAN et de bioconductor (interactions utilisateurs), ou en cours de développement

---

Caractéristiques

R est différent des autres logiciels donc n'essayez pas de rechercher des analogies, il a sa propre façon de travailler. Par exemple, vous n'avez pas besoin de trier (sort) les données pour les résumer, aggréger, splitter, merger... (http://rforsasandspssusers.com, http://www.statmethods.net)

- R est un langage **interprété**
  - == il requiert un autre programme, l'interprète, pour l'éxécution de ses commandes
  - != des langages **compilés**, comme le C ou le C++, qui sont d'abord convertis en code machine par le compilateur avant de pouvoir être éxécutés
- il est basé sur la notion de vecteur (simplification des calculs, utilisation réduite des boucles)
- **pas de typage ni de déclaration obligatoire des variables**

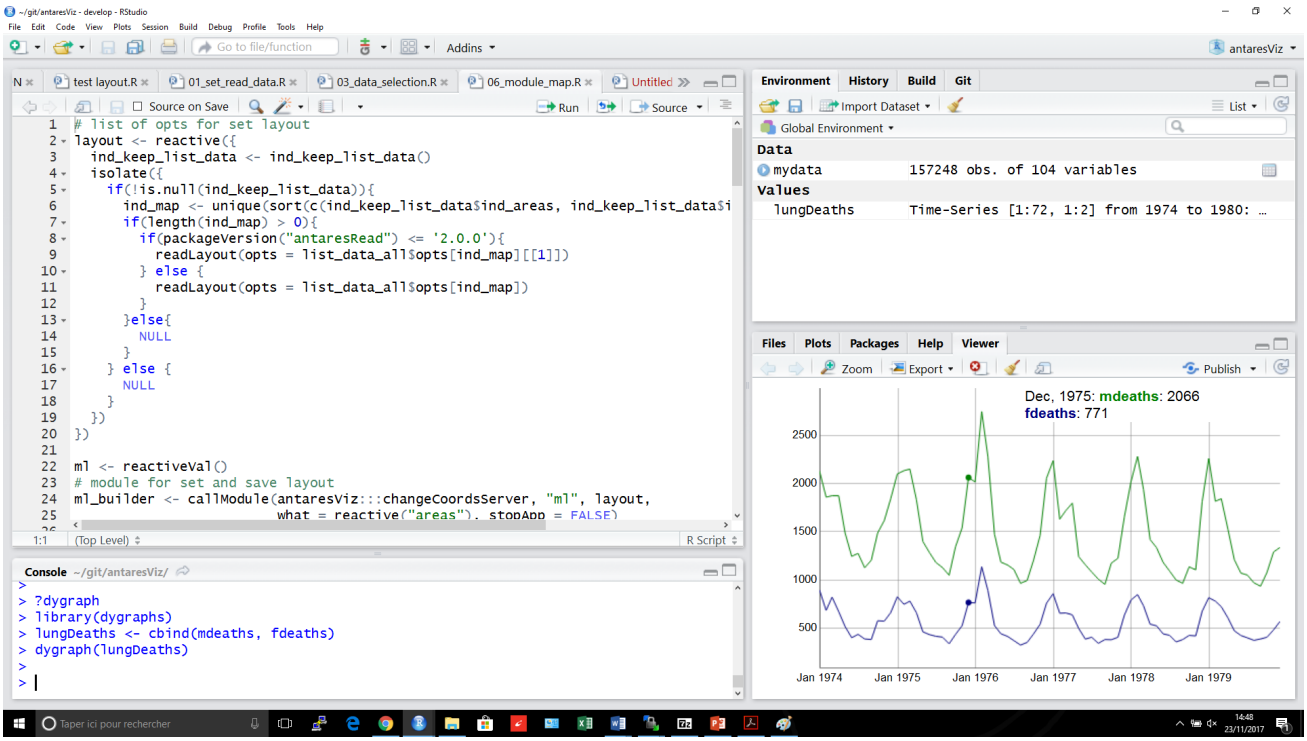


Figure 1:

- **R** est sensible à la casse (minuscules/majuscules)
- **R** travaille **in-memory**
- l'ensemble des données/tables **sont présentes dans la mémoire RAM**

Editeurs de texte / IDE

- Plus confortable (undo-redo, clavier ou souris etc.)
- Plus rapide : dès que l'on recommence !
- Plus clair : commentaires dans le texte

```
#### ma premiere commande
1+1
#### les choses serieuses: importation du fichier
```

- *Tinn-R*, *Notepad++*, *Emacs*, ...
- **Rstudio** s'impose comme le leader actuel.

RStudio

RStudio est un IDE permettant de travailler en **R** dans un environnement de développement riche et complet :

- Editeur de texte, de codes....
- Espace de travail, historique, importation...
- Visualisation, aide, packages
- Console **R**

RStudio Server

	Open Source Edition	Commercial License
Overview	• Access via a web browser • Move computation closer to the data • Scale compute and RAM centrally	All of the features of open source; plus: • Administrative Tools • Enhanced Security and Authentication • Metrics and Monitoring • Advanced Resource Management
Documentation	Getting Started with RStudio Server	RStudio Server Professional Admin Guide
Support	Community forums only	• Priority Email Support • 8 hour response during business hours (ET)
License	AGPL v3	RStudio License Agreement
Pricing	Free	$9,995/server/year Academic and Small Business discounts available

Figure 2:

RStudio

Version **Desktop** et **server** (environnement linux), gratuite ou pro...

- http://www.rstudio.com/products/RStudio
- https://www.rstudio.com/products/rstudio/download/

RStudio

- **Projet** : *file -> New project*. Ensemble de scripts, de données, ...
- **Notebook** : *file -> New file -> R Notebook*. Notebook comme pour **python**, document incluant du code **R**, son évaluation, des commentaires, ... (http://rmarkdown.rstudio.com/r_notebooks.html)
- **R Markdown** : *file -> New file -> R Markdown*. Génération de fichiers *.html*, *.pdf*, *.docx*, ... depuis **R** (http://rmarkdown.rstudio.com/index.html)

Quelques premiers raccourcis utiles :

- **Ctrl + Entrée** : éxécution de la ligne de code courante ou des lignes sélectionnées dans le script
- **Ctrl + Shift + C** : ajout / suppression de commentaires
- **Ctrl + A** : Sélection de tout le script courant
- **Ctrl + I** : Indentation de la sélection

Plus d'informations dans le menu *Tools -> Keyboard shortcuts*

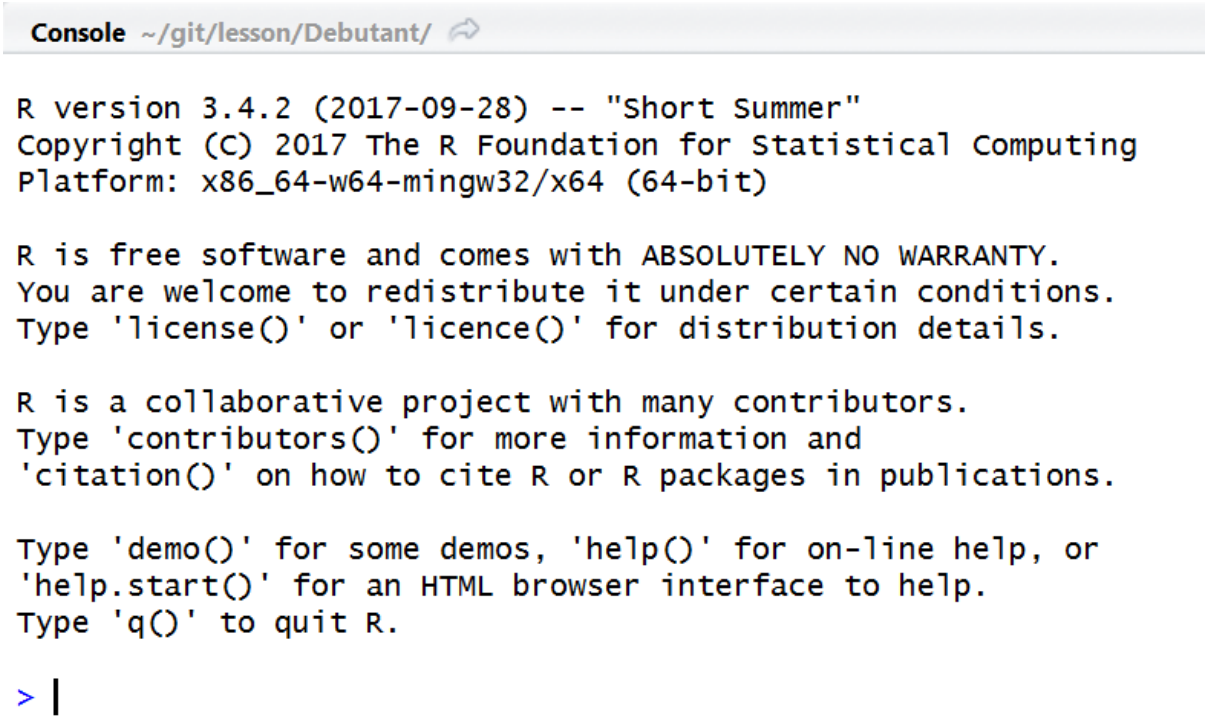


Figure 3:

R optimisé ?

- **R** a été conçu pour utiliser seulement un seul processeur à la fois. Même aujourd'hui, **R** travaille de cette façon par défaut.
- Il est possible de le lié avec les librairies de **BLAS/LAPACK** parallèle.

Microsoft R Open

Microsoft R Open, version de **R** optimisé, inclut ces librairies mathématiques. Cela permet de réduire significativement les temps de calculs pour les opérations matricielles.

https://mran.microsoft.com/rro

https://mran.microsoft.com/documents/rro/multithread

https://mran.microsoft.com/download

Premiers pas

Ouverture d'une session

- R attend une instruction, ceci est indiqué par $>$ en début de ligne.
- Cette instruction doit être validée par *Entrée* pour être *éxécutée*.
- instruction correcte : R éxécute et redonne la main avec $>$
- instruction incorrecte : R retourne $+$, il faut alors compléter l'instruction ou sortir avec *Echap*

Répertoire courant et chemins

Endroit par défaut où *R* écrira/accèdera à des scripts et des fichiers

- **getwd()** : retourne le répertoire courant
- **setwd(path)** : change le répertoire courant
- avec **RStudio** : Session/Set working directory

```
getwd()
```

```
## [1] "C:/Users/Datastorm/Documents/git/lesson/Debutant/Cours"
```

```
setwd("C:\\Users") # notation typée windows, on double les baskslash
setwd("C:/Users") # notation typée linux, un slash
getwd()
```

```
## [1] "C:/Users"
```

Les packages

- Installation via l'onglet **Packages** dans RStudio, ou avec la commande :

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

- Chargement avec **library()** ou **require()** :

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

Trouver de l'aide

- **?** ou **help()**, avec le nom d'un package / d'une fonction
- ou avec l'onglet **Help** dans RStudio

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

Les objets

Les principaux types de données sont :

- **vide** (null) : NULL
- **booléen** (logical) : TRUE, FALSE
- **entier** (integer) : 1, 2, 10
- **numérique** (numeric) : 1, 10.5, 1e-10
- **complexe** (complex) : 2+0i
- **caractères** (character) : 'bonjour', "hello"
- **facteurs** (factor) : type dérivé de **caractères**

Deux grandes familles de structure de données :

- avec des éléments de même type (**vecteur**(vector), **matrice**(matrix))
- avec des éléments de différents types (**liste**(list), **data.frame**(data.frame))

- Création d'un objet par affectation, avec = ou <-
- **objects()** liste les objets en mémoire

```
objects()
```

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

```
x <- 2
objects()
```

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

```
X = 4
objects() # les objets en mémoire
```

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

```
x;X # on peut enchaîner les instructions avec un ;
```

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

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

- **rm()** supprime un ou plusieurs objet
- **exists()** teste l'existance d'un objet

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

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

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

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

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

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

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

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

Vecteur (Logique, Numérique, caractère, . . .)

- création avec la fonction **c()**

```
x <- c(1, 2, 3, 4) ; x
```

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

- ajout d'éléments avec la concaténation

```
c(x, 5)
```

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

- **rep()** : répétitions

```r
rep(c("A", "B"), times = 2)
```

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

```r
rep(c("A", "B"), each = 2)
```

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

- **seq()** : création de séquences

```r
seq(1, 10, by = 2)
```

```
## [1] 1 3 5 7 9
```

```r
seq(1, 10, length = 4)
```

```
## [1]  1  4  7 10
```

```r
1:5 # (=) seq(1, 5, by = 1)
```

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

- vecteur nommé

```r
y <- c(a = 1, b = 2)
y
```

```
## a b
## 1 2
```

- fonctions utiles

```r
is.vector(y)
```

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

```r
class(y);mode(y)
```

```
## [1] "numeric"
## [1] "numeric"
```

```r
length(y)
```

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

```r
names(y)
```

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

```r
names(x)
```

```
## NULL
```

Matrices (Logique, Numérique, caractère, . . .)

- création avec la fonction **matrix()**

```r
matrix(1:3)
```

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

```r
matrix(1:6, nrow = 2, ncol = 3)
```

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

```r
matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)
```

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

- concaténer des matrices :
  - **rbind** pour les lignes
  - **cbind** pour les colonnes

```r
x <- matrix(1:6, nrow = 3, ncol = 2)
y <- matrix(5:6, nrow = 1)
rbind(x, y)
```

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

```r
z <- matrix(5:7)
cbind(x, z)
```

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

les nombres et les noms de lignes/colonnes doivent correspondre

---

- matrice nommée

```r
mat <- matrix(1:4, nrow = 2, ncol = 2, dimnames = list(c("r1", "r2"), c("c1", "c2")))
mat
```

```
##    c1 c2
## r1  1  3
## r2  2  4
```

```r
mat2 <- matrix(1:4, nrow = 2, ncol = 2)
identical(mat, mat2)
```

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

```r
colnames(mat2) <- c("c1", "c2")
rownames(mat2) <- c("r1", "r2")
identical(mat, mat2)
```

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

- fonctions utiles

```r
is.matrix(mat)
```

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

```r
class(mat)
```

```
## [1] "matrix"
```

```r
mode(mat)
```

```
## [1] "numeric"
```

```r
dim(mat) #nrow(mat); ncol(mat)
```

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

```r
colnames(mat);rownames(mat) #dimnames(mat)
```

```
## [1] "c1" "c2"
## [1] "r1" "r2"
```

Listes

- création avec la fonction **list()**

```r
list(1:5, LETTERS[1:2])
```

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

- liste nommée

```r
l <- list(nombres = 1:5, caracteres =LETTERS[1:2])
l
```

```
## $nombres
## [1] 1 2 3 4 5
##
## $caracteres
## [1] "A" "B"
```

- concaténation

```r
l$boolean <- TRUE
l[[2]] <- "remplacement de l'element 2"
l[[5]] <- "nouvel d'un element en 5ième position"
l
```

```
## $nombres
## [1] 1 2 3 4 5
##
## $caracteres
## [1] "remplacement de l'element 2"
##
## $boolean
## [1] TRUE
##
## [[4]]
## NULL
##
## [[5]]
## [1] "nouvel d'un element en 5ième position"
```

- fonctions utiles

```r
is.list(l)
```

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

```r
class(l)
```

```
## [1] "list"
```

```r
sapply(l, class)
```

```
##     nombres  caracteres     boolean
##   "integer" "character"   "logical"      "NULL" "character"
```

```r
length(l)
```

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

```r
names(l)
```

```
## [1] "nombres"    "caracteres" "boolean"    ""           ""
```

---

data.frame

Le **data.frame** est une **liste de vecteurs**

- création avec la fonction **data.frame()**

```r
data.frame(nombres = 1:3, lettres = LETTERS[1:3])
```

```
##   nombres lettres
## 1       1       A
## 2       2       B
## 3       3       C
```

- ajout de rownames

```
data.frame(nombres = 1:3, lettres = LETTERS[1:3], row.names = paste0("r", 1:3))
```

```
##    nombres lettres
## r1       1       A
## r2       2       B
## r3       3       C
```

- concaténer des data.frames :
  - **rbind** pour les lignes
  - **cbind** pour les colonnes

```
x <- data.frame(nombres = 1:2, lettres = LETTERS[1:2])
y <- data.frame(nombres = 3, lettres = "C")
rbind(x, y)
```

```
##   nombres lettres
## 1       1       A
## 2       2       B
## 3       3       C
```

```
z <- c(TRUE, FALSE)
cbind(x, z) # (=) x$z <- c(TRUE, FALSE), list !
```

```
##   nombres lettres     z
## 1       1       A  TRUE
## 2       2       B FALSE
```

les nombres et les noms de lignes/colonnes doivent correspondre

---

- fonctions utiles

```
is.data.frame(x)
```

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

```
is.list(x) # un data.frame est en fait une liste...!
```

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

```
class(x)
```

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

```
sapply(x, class)
```

```
##   nombres   lettres
## "integer"  "factor"
```

```
dim(x) #nrow(x); ncol(x)
```

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

```
colnames(x)
```

```
## [1] "nombres" "lettres"
```

```r
rownames(x)
```

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

```r
head(x)
```

```
##   nombres lettres
## 1       1       A
## 2       2       B
```

```r
tail(x, n = 1)
```

```
##   nombres lettres
## 2       2       B
```

Les facteurs

- fonction **as.factor()**

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

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

```r
levels(f) # les niveaux des facteurs
```

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

```r
nlevels(f) # le nombre de niveaux
```

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

```r
relevel(f, ref = "B") # changer le niveau de référence
```

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

- Découpage en classes avec la fonction **cut()**

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

- Fusion / renommage de niveaux avec **levels()**

```r
levels(f)
```

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

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

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

Sélection dans les objets

- par la position, dans ce cas il faut indiquer un vecteur de position (il peut être longueur différente de l'objet)
- par des noms, si il y en a. . . !, avec le même principe que pour les positions
- par des booléans, dans ce cas le **vecteur de booléen doit être de la longueur de l'objet à sélectionner**. On ne conserve que les `TRUE`

Les opérateurs logiques, renvoyant des booléans :

- ==, !=, >, <, >=, <=
- **%in%** appartenance à un ensemble de caractères
- **&** pour satisfaire plusieurs conditions
- | pour satisfaire au-moins une condition
- **!** la négation de booléans

Exemples sur des vecteurs

Par la sélection avec des positions :

```
x <- c(1:10)
x[5]
```

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

```
x[c(1, 10, 1)]
```

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

Par la suppression :

```
x[-c(1:8)]
```

```
## [1]  9 10
```

Par des logiques :

```
x < 3
```

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

```
x[x < 3]
```

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

Avec des noms :

```
x <- c(a = 1, b = 2, c = 3)
x["a"] ; x[["a"]] ; x[c("a", "b", "a")]
```

```
## a
## 1
```

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

```
## a b a
## 1 2 1
```

Exemples sur les matrices

```
mat <- matrix(1:9, nrow = 3, ncol = 3,
              dimnames = list(c("r1", "r2", "r3"), c("c1", "c2", "c3")))
```

Par les positions :

```
mat[1:2, c(1, 3)]
```

```
##    c1 c3
## r1  1  7
## r2  2  8
```

```
mat[-c(2),]
```

```
##    c1 c2 c3
## r1  1  4  7
## r3  3  6  9
```

Par des logiques :

```
mat[mat[, 1] == 1,]
```

```
## c1 c2 c3
##  1  4  7
```

Avec les noms :

```
mat["r1","c1"]
```

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

Attention quand on sélectionne une seule colonne, R renvoie un vecteur plutôt qu'une matrice. On contrôle cela avec l'option **drop = FALSE**

C'est une des sources d'erreurs de code la plus fréquente !

```
mat[, 1]
```

```
## r1 r2 r3
##  1  2  3
```

```
mat[, 1, drop = FALSE]
```

```
##    c1
## r1  1
## r2  2
## r3  3
```

Exemples sur les data.frame

```
dat <- data.frame(nombres = 1:3, lettres = LETTERS[1:3], row.names = paste0("r", 1:3))

dat[1:2, 1, drop = FALSE]
```

```
##    nombres
## r1       1
## r2       2
```

```
dat[-c(2),]
```

```
##    nombres lettres
## r1       1       A
## r3       3       C
```

```
dat[dat$lettres %in% c("A", "C") == 1,]
```

```
##    nombres lettres
## r1       1       A
## r3       3       C
```

```
dat["r1","lettres"]
```

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

Exemples sur les listes

```
l <- list(nombres = 1:5, caracteres =LETTERS[1:2])
l[[1]] # on récupère l'élément de la liste
```

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

```
l$caracteres
```

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

```
l[1] # on récupère une liste
```

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

```
l[c(1, 1)]
```

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

Afficher des informations sur la structure d'objet

en utilisant la fonction **str()**

```r
## vecteur
x <- 1:10
str(x)
```

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

```r
## data.frame (même affichage pour les matrices)
dat <- data.frame(nombres = 1:3, lettres = LETTERS[1:3])
str(dat)
```

```
## 'data.frame':    3 obs. of  2 variables:
##  $ nombres: int  1 2 3
##  $ lettres: Factor w/ 3 levels "A","B","C": 1 2 3
```

```r
## liste
l <- list(nombres = 1:5, caracteres =LETTERS[1:2])
str(l)
```

```
## List of 2
##  $ nombres   : int [1:5] 1 2 3 4 5
##  $ caracteres: chr [1:2] "A" "B"
```

Résumer l'information

la plupart des objets R possèdent une méthode **summary** résumant l'information

```r
## vecteur
x <- 1:10
summary(x)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    3.25    5.50    5.50    7.75   10.00
```

```r
## data.frame (même affichage pour les matrices)
dat <- data.frame(nombres = 1:3, lettres = LETTERS[1:3])
summary(dat)
```

```
##     nombres    lettres
##  Min.   :1.0   A:1
##  1st Qu.:1.5   B:1
##  Median :2.0   C:1
##  Mean   :2.0
##  3rd Qu.:2.5
##  Max.   :3.0
```

Ordonner les données

- la fonction **sort()** permet de trier un vecteur
- la fonction **order()** retourne les indices des données triées

```r
x <- round(rnorm(10), digits = 2)
sort(x)
```

```
##  [1] -1.31 -0.99 -0.76 -0.61 -0.04  0.20  0.21  0.22  0.43  0.83
```

```
order(x)
```

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

```
sort(x, decreasing = TRUE)
```

```
##  [1]  0.83  0.43  0.22  0.21  0.20 -0.04 -0.61 -0.76 -0.99 -1.31
```

```
order(x, decreasing = TRUE)
```

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

Compter les occurences

- la fonction **table()** permet de compter les occurences d'une variables, ou de plusieurs variables croisées

```
dat <- data.frame(nb = sample(1:10, 100, replace = T),
                  lt = sample(LETTERS[1:3], 100, replace = T))
head(dat, n = 2)
```

```
##   nb lt
## 1  6  C
## 2  2  B
```

```
table(dat$lt)
```

```
##
##  A  B  C
## 38 30 32
```

```
table(dat$lt, dat$nb)
```

```
##
##     1 2 3 4 5 6 7 8 9 10
##   A 4 7 1 2 4 6 4 6 2  2
##   B 5 2 3 2 2 6 3 1 4  2
##   C 0 5 4 2 2 9 7 1 1  1
```

Dédupliquer les données

- la fonction **unique()** renvoie les valeurs uniques d'un objet. Elle peut s'appliquer aux data.frame et matrices
- la fonction **duplicated()** renvoie les valeurs en doubles

```
x <- c(1:5, 2)
x
```

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

```
unique(x)
```

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

```r
duplicated(x)
```

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

```r
duplicated(x, fromLast = T)
```

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

```r
which(duplicated(x, fromLast = T)) # which retourne les indices TRUE
```

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

Importer/exporter des données

fichiers .csv / .txt

- **read.table()**, **write.table()**, **read.csv()**, **write.csv()**
- Principaux arguments :
  - *header* / *col.names* : nom des colonnes
  - *row.names* : nom des lignes
  - *sep* : séparateur de champs
  - *dec* : décimale
  - *skip* : sauter des lignes

```r
dat <- data.frame(x = 1:9, y = LETTERS[1:3])
# exportation
write.table(dat,"C/temp/tab3.csv",sep = ";", row.names = FALSE, col.names = TRUE)

#importation
dat <- read.table("C/temp/tab3.csv",sep = ";", header = TRUE)
```

- importation et exportation plus rapide avec le package **data.table**

```r
fwrite(dat,"C/temp/tab3.csv")
dat <- fread("C/temp/tab3.csv")
```

fichiers .xlsx / .xls

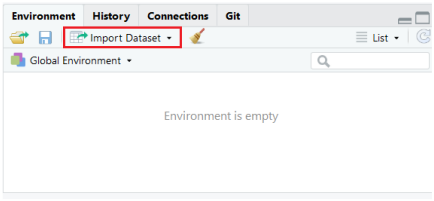
- avec le package **xlsx** ou **XLConnext** (dépend de **Java**)
- avec le package **openxlsx**
  - `read.xlsx()`, `write.xlsx()`

fichiers SAS / SPSS

- avec le package **sas7bdat** (`read.sas7bdat`) ou **haven** (`read_sas`)

```r
require(sas7bdat)
sasdata <- read.sas7bdat("raw_sas.sas7bdat")
```

- avec le package **foreign** (format xport)



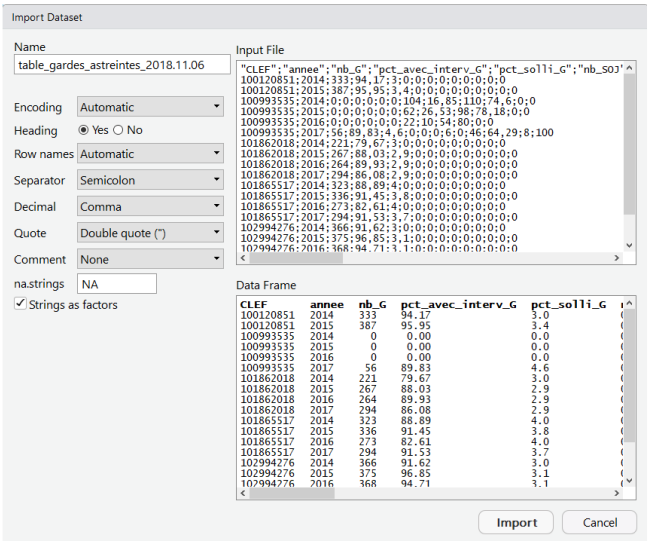


Figure 4:

En utilisant l'interface de RStudio

Quelques manipulations sur les data.frame

- ajouter une nouvelle variable qui résulte d'un calcul entre d'autres variables : **transform()**

```
dat <- data.frame(x = 1:3, y = c(10,20,30))
dat <- transform(dat, z = x+y) # équivalent à dat$z = dat$x + dat$y
head(dat)
```

```
##   x  y  z
## 1 1 10 11
## 2 2 20 22
## 3 3 30 33
```

- faire un subset sur les lignes : **subset()**

```
subset(dat, x > 2) # dat[dat$x > 2, ]
```

```
##   x  y  z
## 3 3 30 33
```

```
subset(dat, x > 1, select = c(y))
```

```
##    y
## 2 20
## 3 30
```

- découper le jeu de données en fonction de critère : **split()**

```
dat <- data.frame(x = 1:9, y = LETTERS[1:3])
split(dat, dat$y)
```

```
## $A
##   x y
## 1 1 A
## 4 4 A
## 7 7 A
##
## $B
##   x y
## 2 2 B
## 5 5 B
## 8 8 B
##
## $C
##   x y
## 3 3 C
## 6 6 C
## 9 9 C
```

On récupère une liste

- faire des calculs aggrégés : **aggregate()**

```
aggregate(dat$x, by = list(dat$y), FUN = max)
```

```
##   Group.1 x
## 1       A 7
## 2       B 8
## 3       C 9
```

```
aggregate(dat$x, by = list(dat$y), FUN = mean)
```

```
##   Group.1 x
## 1       A 4
## 2       B 5
## 3       C 6
```

- Sur plusieurs colonnes :

```
aggregate(state.x77, list(Region = state.region), mean)
```

```
##          Region Population   Income Illiteracy Life Exp    Murder  HS Grad
## 1     Northeast   5495.111 4570.222   1.000000 71.26444  4.722222 53.96667
## 2         South   4208.125 4011.938   1.737500 69.70625 10.581250 44.34375
## 3 North Central   4803.000 4611.083   0.700000 71.76667  5.275000 54.51667
## 4          West   2915.308 4702.615   1.023077 71.23462  7.215385 62.00000
##      Frost      Area
## 1 132.7778  18141.00
## 2  64.6250  54605.12
## 3 138.8333  62652.00
## 4 102.1538 134463.00
```

Pour aller plus loin : ?apply, ?lapply, ?tapply

Fusion de données

- avec la fonction **merge()**
  - contrôle des clés de jointure avec `by` (clés identiques) ou `by.x` et `by.y` (clés différentes)
  - sens de la jointure avec `all`, `all.x` et `all.y`.

```
dat <- data.frame(x = 1:3, y = LETTERS[1:3])
dat2 <- data.frame(x = sample(1:3, 10, replace = T))
merge(dat2, dat, by = "x")
```

```
##    x y
## 1  2 B
## 2  2 B
## 3  2 B
## 4  3 C
## 5  3 C
## 6  3 C
## 7  3 C
## 8  3 C
## 9  3 C
## 10 3 C
```

Introduction aux fonctions

On définit une nouvelle fonction avec la syntaxe suivante :

`fun <- function(arguments) expression`

- **fun** le nom de la fonction
- **arguments** la liste des arguments, séparés par des virgules. *formals(fun)*
- **expression** le corps de la fonction. une seul expression, ou plusieurs entre des accolades. *body(fun)*

```
squared <- function(x){
  x^2
}

squared(2)
```

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

Les arguments

- **Valeur par défaut**
  - via une affetaction, avec '=', dans la définition de la fonction
  - optionnel lors de l'appel

```
mysum <- function(x, y = 2){
  x + y
}
mysum(x = 2)           # 4
mysum(x = 2, y = 10)   # 12
```

- **Dépendances entre arguments**

On peut définir un argument en fonction d'autres arguments

```
# avec une expression simple
mysum2 <- function(x, y = x + 10)  x + y
mysum2(5)  # 20
```

Retourner un résultat

Une fonction retourne par défaut le résultat de la dernière expression

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

somme <- mysum(x = 2, y = 10)
somme
```

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

- Renvoi d'un résultat avant la fin de la fonction : fonction *return()*
- Retour de plusieurs résultats : liste nommée.

```
exemple <- function(x, y){
  if(y == 0){
    return(x)
  }
  list(x = x, y = y)
}
```

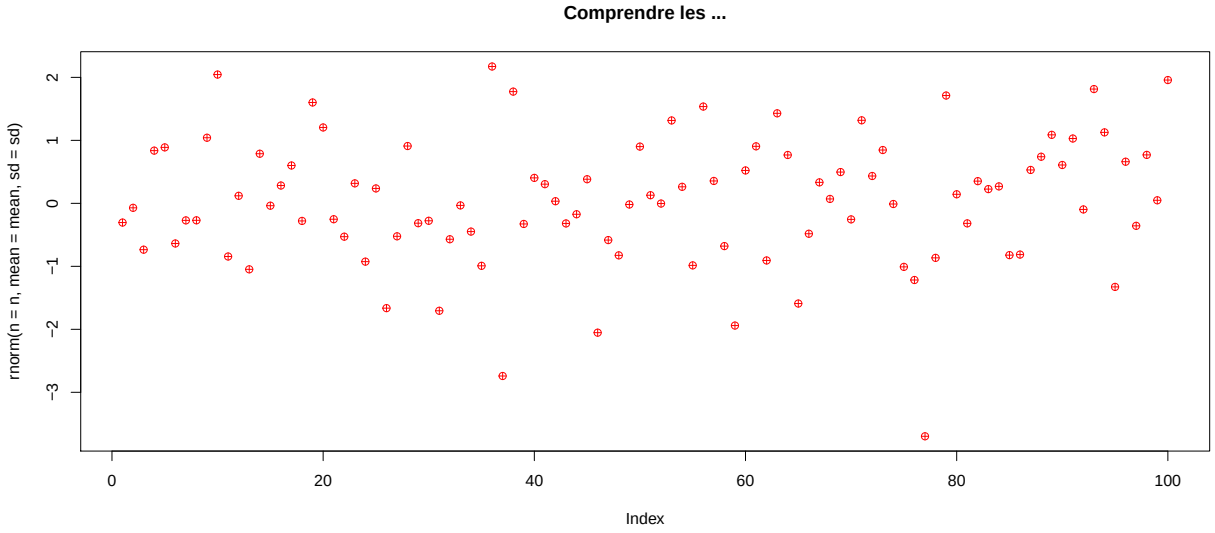
Comprendre les '...'

- Signifie que la fonction accepte d'autres arguments que ceux définis explicitement
- Sert généralement à passer ces arguments à une autre fonction
- Se récupère facilement avec : *list(...)*

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

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

rnormPlot <- function(n, mean = 0, sd = 1, ...){
  plot(rnorm(n = n, mean = mean, sd = sd), ...)
}
rnormPlot(n = 100, main = "Comprendre les ...", col = "red", pch = 10)
```



Sauvegarder et restaurer un ensemble d'objets R

- la fonction **save()** permet d'enregistrer sur le disque un ou plusieurs objets R
- le stockage des objets est compréssé
- on utilise généralement l'extension **.RData**
- on peut recharger les objets avec la fonction **load()**
- **les objets auront le même nom que lors de la sauvegarde**

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

Cette utilisation peut poser problème si des variables du même nom existent déjà dans l'environnement lors du chargement.

Sauvegarder et restaurer un objet R

- la fonction **saveRDS()** permet d'enregistrer un objet
- le stockage des objets est compréssé
- on utilise généralement l'extension **.RDS**
- on peut recharger les objets avec la fonction **readRDS()**
- **on affecte l'objet à la variable de son choix**

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

Connexion à des bases de données

Il existe au minimum un package qui permet de se connecter à chaque type de bases de données, par exemple :

- **RMySQL** : base de données MySQL
- **ROracle** : base de données Oracle
- **RPostgreSQL** : base de données PostgreSQL
- **RSQLServer** : base de données MS SQL Server
- **mongolite** : base de données NoSQL mongodb
- **RSQLite** : base de données SQLite
- **RJDBC** : connexion à différents types de bases de données via du Java
- **RODBC** : connexion à différents types de bases de données via du ODBC

Syntaxe usuelle de connexion

- ouverture de la connexion avec **dbConnect()**
- requêtage avec **dbGetQuery()**
- fermeture de la connexion avec **dbDisconnect()**

```
con <- dbConnect(dbDriver("PostgreSQL"), host = "myserver", port = 123,
                 dbname = "database", user = "benoit", password = "security")
res <- dbGetQuery(con, "SELECT * FROM mytable")
dbDisconnect(con)
```

Et aussi :

- **dbListTables()**, **dbExistTable()**, **dbListFields()**
- **dbReadTable()**, **dbWriteTable()**,
- **dbSendQuery()** (écriture dans la table), **dbColumnInfo()**,
- ...

Les principales fonctions de statistiques descriptives

- **mean()**
- **median()**
- **var()**
- **sd()**
- **sum()**
- **max()**
- **min()**
- **range()**
- **quantile()**

Faire attention aux données manquantes. Utilisation de l'argument `na.rm = TRUE` pour les retirer des calculs.

Fonctions de distributions

- **?Distribution** dans R

- syntaxe :
  - **dxxx()**, densité
  - **pxxx()**, distribution cumulée
  - **qxxx()**, quantile
  - **rxxx()**, génération aléatoire

avec **xxx** nom de la loi (*norm*, *unif*, *pois*, *gamma*, ...)

```
val <- rnorm(100)
mean(val);sd(val)
```

```
## [1] 0.1128202
```

```
## [1] 1.134419
```

Traitement des chaînes de caractères

Règles de construction

- Entourée de simple quote ' ou de double quote ", pas un mélange des deux
- Insertion possible d'un " (resp. ') dans une chaîne délimitée par des ' (resp. ")
- Utilisation de \ dans le cas contraire

```
"double quote"
```

```
## [1] "double quote"
```

```
'simple quote'
```

```
## [1] "simple quote"
```

```
cat("l'insertion se passe \"bien\"")
```

```
## l'insertion se passe "bien"
```

Concaténation

```
paste (..., sep = " ", collapse = NULL)
paste0(..., collapse = NULL) # pas de séparateur, un peu plus rapide
```

- **...** : un ou plusieurs objets **R**
- **sep** : charactère de séparation des termes
- **collapse** : charactère de séparation des résultats

```
paste("Formation R", 1:3, sep = "-")
```

```
## [1] "Formation R-1" "Formation R-2" "Formation R-3"
```

```
paste("Formation R", 1:3, sep = "-", collapse = ", ")
```

```
## [1] "Formation R-1, Formation R-2, Formation R-3"
```

Nombre de caractères

Utilisation de la fonction `nchar()` :

```
nchar("Chaine de 23 caractères")
```

```
## [1] 23
```

Majuscules / minuscules

Utilisation des fonctions `toupper()` et `tolower()` :

```
toupper("Test") ; tolower("Test")
```

```
## [1] "TEST"
```

```
## [1] "test"
```

Extraction / Remplacement

Utilisation de la fonction `substring()` :

```
substring(text, first, last)
```

- **text** : une chaîne de caractères
- **first** : indice du premier élement
- **last** : indice du dernier élement

```
x <- "abcdef"
substring(x, 1, 3)
```

```
## [1] "abc"
```

```
substring(x, 1, 3) <- "123" ; x
```

```
## [1] "123def"
```

Formattage des nombres et des caractères

La fonction `format` permet de formatter des numériques et des caractères :

```
format(x, trim = FALSE, digits = NULL, nsmall = 0L,
       justify = c("left", "right", "centre", "none"),
       width = NULL, na.encode = TRUE, scientific = NA,
       big.mark   = "",   big.interval = 3L,
       small.mark = "", small.interval = 5L,
       decimal.mark = getOption("OutDec"),
       zero.print = NULL, drop0trailing = FALSE, ...)
```

Principaux arguments :

- **x** : vecteur / valeur d'entrée
- **digits** : nombre total de chiffres à afficher
- **nsmall** : nombre de décimales

- **scientific** : notation scientifique N
- **width** : taille minimale (rajout d'espaces le cas échéants)
- **justify** : alignement

Formattage des nombres et des caractères

```
# Maximum de 5 chiffres
format(123.47872, digits = 5)
```

```
## [1] "123.48"
```

```
# Notation scientifique
format(c(12.1, 0.00001), scientific = TRUE)
```

```
## [1] "1.21e+01" "1.00e-05"
```

```
# Nombre de décimale
format(c(23.478989898, 45), nsmall = 5)
```

```
## [1] "23.47899" "45.00000"
```

```
# Taille + alignement
format("CISAD", width = 9, justify = "c")
```

```
## [1] "  CISAD  "
```

Découpage d'un chaîne de caractères

La fonction `strsplit` permet de découper une ou plusieurs chaînes de caractères par rapport à une sous-chaîne ou une expression régulière

```
x <- "10 + 20"

unlist(strsplit(x, split = "+"))
```

```
## [1] "1" "0" " " " "+" " " " "2" "0"
```

```
# fixed = TRUE : désactivation des expressions régulières
unlist(strsplit(x, split = "+", fixed = TRUE))
```

```
## [1] "10 " " 20"
```

```
unlist(strsplit(x, split = "[[:space:]]*[+][[:space:]]*"))
```

```
## [1] "10" "20"
```

Les expressions régulières

- Documentation dans **R** :

```
?regex
```

- Cheatsheet : https://www.rstudio.com/wp-content/uploads/2016/09/RegExCheatsheet.pdf

Quelques exemples de manipulations :

```
v_str <- c("3 enfants et 1 chien", "", "Nombre : 2")
```

Détection de pattern

Fonctions `grep` et `grepl` :

```
grep("[[:digit:]]", v_str)
```

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

```
grep("[[:digit:]]", v_str, value = TRUE)
```

```
## [1] "3 enfants et 1 chien" "Nombre : 2"
```

```
grepl("[[:digit:]]", v_str)
```

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

Remplacement de pattern

Fonctions `gsub` et `sub` :

```
# gsub : remplacement de l'ensemble
gsub("[[:alpha:]]|[[:punct:]]|[[:space:]]", "", v_str)
```

```
## [1] "31" ""   "2"
```

```
# sub : remplacement de la première occurence
sub("[[:alpha:]]|[[:punct:]]|[[:space:]]", "", v_str)
```

```
## [1] "3enfants et 1 chien" ""                    "ombre : 2"
```

Localisation et Extraction de pattern

Fonctions `regexpr`, `gregexpr` et `regmatches` :

```
# regexpr : première occurence trouvée
regmatches(v_str, regexpr("[[:digit:]]", v_str))
```

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

```
# gregexpr : ensemble des occurences trouvées
regmatches(v_str, gregexpr("[[:digit:]]", v_str))
```

```
## [[1]]
## [1] "3" "1"
##
## [[2]]
## character(0)
##
## [[3]]
```

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

Alternative : utilisation du package stringr

Mêmes fonctionnalités, mais avec une syntaxe différente

`str_sub`

`str_replace`

`str_to_lower`

`str_extract`

`str_trim`

`...`

http://stringr.tidyverse.org/

Ressources

- The R Core Team Intro : https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf
- The R Core Team Langage Definition : https://cran.r-project.org/doc/manuals/r-release/R-lang.pdf
- Introduction à la programmation R : https://cran.r-project.org/doc/contrib/Goulet_introduction_programmation_R.pdf
- Cheatsheets RStudio : https://www.rstudio.com/resources/cheatsheets/