Atelier - Introduction à l'environnement R

Renato Henriques-Silva

February 9, 2019

Contents

1 Module I

1.1 L'environnement R

R est un environnement d'analyse de données développé sous licence libre. Très puissant pour réaliser n'importe quel type d'analyses statistiques, il s'avère aussi extrêmement performant dans la visualisation des données.

1.2 Pré-requis pour l'atelier

Installer le logiciel R

- Télécharger le logiciel R sur ce lien LIEN
- Télécharger le logiciel R-Studio sur ce lien LIEN

N'oubliez pas de choisir l'option de téléchargement selon votre système opérationelle (Windows, OS, Linux). R-Studio est un environnement de développement (EDI) qui donne une interface plus "user-friendly" à l'utilisation de R.

1.3 Arithmétique

R peut être utilisé comme une simple calculatrice

- Addition: `+`
- Substraction: `-`
- Multiplication: `*`
- Division: `/`
- Exponent: `^`
- Modulo: `%%`

```
# Pour écrire des commentaires dans votre script, utilisez "#" avant le  texte
# Les commentaires ne sont pas executés quand le script tourne

# Par exemple, la prochaine ligne n'est pas éxécutée donc on a pas le résultat attendu (7)

# 5 + 2

# Si on enlève le '#', l'addition est éxécutée et on obtien le résultat (7)

# Addition
5 + 2
```

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

```
# Substraction
5 - 2
```

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

```
# Multiplication
5 * 2
```

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

```
# Division
5 / 2
```

```
## [1] 2.5
```

```r
# Exponent
5 ^ 2
```

```
## [1] 25
```

```r
# Modulo
5 %% 2
```

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

1.4 Variables

Une variable informatique est un espace mémoire réservé, un objet virtuel manipulé par un programme. Pour affecter une variable il faut faire le suivant:

x <- 5

```r
x <- 5 # afféctation de la variable x

x # affichez le contenu de la variable x
```

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

```r
# Vous pouvez utiliser "=" aussi me ce n'est pas recommandé.
```

1.4.1 Opération avec des variables

Tout comme les chiffres, vouz pouvez faire des opérations arithmétiques avec des variables. Par exemple avec x + 5 vous obtenez 10.

```r
x + 5
```

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

Vous pouvez aussi faire une opération seulement avec des variables (et aussi affecté le résultat à une 3ème variable).

```r
# afféctation de la variable y

y <- 10

# afféctation de la variable z avec le résultat de x/y

z <- x/y

# affichage de z
z
```

```
## [1] 0.5
```

1.4.2 Classes de base des variables sur R

- **numeric**, donc des nombres qui ont des valeurs décimals `3.82`
- **integer**, donc des nombres entier `3` - c'est une sous-classe de "numerics"
- **logical**, donc une variable booléene `TRUE` ou `FALSE`

- **character**, donc du texte (aussi appelé de string) `'le chien'`

Pour créer une variable booléene, il faut seulement affécté les valeurs de `TRUE` ou `FALSE`, ou même `T` et `F`.

```
var_bool <- FALSE
var_bool
```

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

```
var_bool <- F
var_bool
```

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

Pour crérer une variable string, on a juste à encapsuler le contenu par des " " ou des ‘’.

```
var_string <- "le chien"
var_string
```

```
## [1] "le chien"
```

```
var_string <- 'le chat'
var_string
```

```
## [1] "le chat"
```

Notez que R va prioriser la classe “numeric” par rapport à la classe “integer” quand vous affectez une nouvelle variable, même si le chiffre est entier.

Pour déterminer le type d’une variable, on peut utiliser la fonction *class()* .

```
# Par exemple, x est entier mais quand on vérifie son type,
# R nous indique qu'il est 'numeric'
class(x)
```

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

```
# var_bool est "logical"
class(var_bool)
```

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

```
# var_string est "character"
class(var_string)
```

```
## [1] "character"
```

1.4.3 Valeurs Manquantes

Souvant dans les bases de données utilisés vous allez rencontrer des valeurs manquantes. Sur R elles sont affichés sous la forme de `NA` pour *not available*.

```
taille <- c(110, 30, NA, 40, NA)
taille
```

```
## [1] 110  30  NA  40  NA
```

```
# Notez que les valeurs "NA" occupent une position. Donc quand on demande la longueur du vecteur "taill
length(taille)
```

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

Il ne faut pas confondre `NA` avec un autre objet que l'on rencontre sous R appelé `NULL` qui représente l'objet vide. `NULL` ne contient absolument rien du tout. La différence se comprends mieux lorsque que l'on essaie de combiner ces objets. .

```
taille<-c(110, 30, NULL, 40, NA)
# Au contraire de NA, NULL n'occupe pas de position
length(taille)
```

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

```
taille
```

```
## [1] 110  30  40  NA
```

```
length(c(NULL,NULL))
```

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

```
length(c(NA,NA))
```

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

1.4.4 Dates

Un autre type de variable que vous pouvez rencontrer ce sont les *dates*. Vous pouvez créer cette classe de variable avec la fonction *as.Date()*:

```
# En premier créez un vecteur de caractères avec les dates
dates <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92", "02/01/92")
# le type de variable est "character"
class(dates)
```

```
## [1] "character"
```

```
# en suite, vous utilisez la fonction as.Date()
# ATTENTION!! il faut spécifier le format des dates avec
# l'argument format (en utilisant des caractères) dans la fonction

# Sur le premier l'example, il y a le mois (m)
# suivi du jour (d) et ensuite l'année (y).
# Et ils sont séparés par des "/"
# Donc le format est le suivant:
dates <- as.Date(dates, format = "%m/%d/%y")
dates
```

```
## [1] "1992-02-27" "1992-02-27" "1992-01-14" "1992-02-28" "1992-02-01"
```

```
# Et la classe change aussi
class(dates)
```

```
## [1] "Date"
```

```
# Si c'est un autre séparateur, par exemple "-"
dates <- c("02-27-92", "02-27-92", "01-14-92", "02-28-92", "02-01-92")
# la fonction as.Date vous retournera des NA si vous utilisez le mauvais format
as.Date(dates, format = "%m/%d/%y")
```

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

```r
# Si vous utilisez le séparateur approprié
dates <- as.Date(dates, format = "%m-%d-%y")
dates
```

```
## [1] "1992-02-27" "1992-02-27" "1992-01-14" "1992-02-28" "1992-02-01"
```

```r
# Il y a d'autres format qui peuvent être utilisés
# Pour en savoir plus, accedez à la page d'aide de la fonction en utilisant -> ?fonction
# ?as.Date
```

1.4.5 Fonctions de verification et conversion de variables

1.4.5.1 Verification

Ces fonctions retournent une valeur logique, TRUE (oui) et FALSE (non).

- *is.numeric()*
- *is.character()*
- *is.logical()*
- *is.factor()*
- *is.na()*
- *is.null()*

```r
# Est un vecteur numérique ?
is.numeric(5)
```

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

```r
is.numeric('5')
```

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

```r
# Est un vecteur caractère ?
is.character('5')
```

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

```r
is.character('chien')
```

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

```r
# Est un facteur ?
is.factor('chien')
```

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

```r
is.factor(factor('chien'))
```

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

```r
# Est une valeur logique ?
is.logical(TRUE)
```

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

```r
is.logical(FALSE)
```

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

```r
# Est une valeur manquante?
is.na(5)
```

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

```r
is.na(NA)
```

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

```r
# Est une valeur nulle?
x <- NULL
is.null(x)
```

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

1.4.5.2 Conversion

Ces fonctions transforment les variables d'une classe à l'autre. Chacune a des conditions spécifiques pour fonctionner. Utiliser le ? pour en savoir plus.

- *as.character()*
- *as.numeric()*
- *as.factor()*
- *as.logical()*
- *as.Date()*

```r
# Quelques examples

as.numeric('5')
```

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

```r
is.numeric(as.numeric('5'))
```

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

```r
as.character(c(10,2))
```

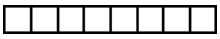
```
## [1] "10" "2"
```

1.5 Objets

Les objets dans R servent à regrouper les variables (chiffres, textes, FALSE ou TRUE etc.).

1.5.1 Vecteurs

Un vecteur est un ensemble de valeurs de classe identique avec 1 dimension.



Il y a plusieurs manières de créer des vecteurs.

```r
# quelques examples

# vecteur à un élément
vec <- 7
vec
```

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

```
# Série d'entiers (de 1 a 12)
vec <- 1:12
vec
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
# Série de 1 a 10 avec pas de 2
vec <- seq(from = 1, to = 10, by = 2)
vec
```

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

```
# répétition de "aa" 5 fois
vec <- rep("aa",5)
vec
```

```
## [1] "aa" "aa" "aa" "aa" "aa"
```

```
# Concaténation (ici caractères)
vec <- c(2,5,-3,8,"a")
vec
```

```
## [1] "2"  "5"  "-3" "8"  "a"
```

```
# Vecteur numérique
vec <- c(1,5,-36,3.66)
vec
```

```
## [1]   1.00   5.00 -36.00   3.66
```

```
# Vecteur logique
vec <- c(T,T,T,F,F)
vec
```

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

Vous pouvez aussi donner des noms à chaque élément d'un vecteur avec la fonction *names()*

```
vec <-c(100,20,45,60)
names(vec)<-c("A","B","C","D")
vec
```

```
##   A   B   C   D
## 100  20  45  60
```

```
# comme vous voyez, on crée un vecteur de 'strings' de la même taille que le vecteur "vec" et on utilis
# la fonction "names" pour indiquer que ça va etre les noms de chaque élément de "vec".

# si vous voulez effacer les noms, utilisez la fonction "unname"
unname(vec)
```

```
## [1] 100  20  45  60
```

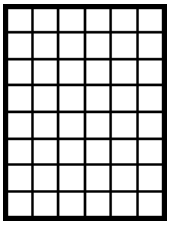
Pour afficher la taille (ou la longueur) d'un vecteur, il faut utiliser la fonction *length()*.

```
# longueur de "vec"
length(vec)
```

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

1.5.2 Matrices

Une matrice est un vecteur avec 2 dimensions.



Il y a plusieurs manières de créer des matrices.

```
# quelques examples

# 4 élements, 2 lignes et 3 colonnes
mat <-matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3)
mat
```

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

```
# 4 élements, 3 lignes et 2 colonnes
mat<- matrix(c(1, 2, 3, 4, 5, 6), nrow = 3, ncol = 2)
mat
```

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

```
# série de 1 à 20 répartis entre 4 lignes et 5 colonnes
mat <- matrix(seq(1,20,1), nrow=4, ncol=5)
mat
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
```

```
# notez que la matrice est remplie par les colonnes en premier
# si vous voulez qu'elle se remplisse par les lignes, il faut activer l'argument "byrow"
# avec le booléan "TRUE" (par défault il est en mode FALSE)
mat <- matrix(seq(1,20,1), nrow=4, ncol=5, byrow = TRUE)
mat
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
## [4,]   16   17   18   19   20
```

```
# cbind = combinaison de deux vecteurs de 4 élements par colonne
mat <- cbind(1:4,5:8)
```

```
mat
```

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

```
# rbind = combinaison de deux vecteurs de 4 élements par ligne
mat <- rbind(1:4,5:8)
mat
```

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

Vous pouvez nommer les lignes et les colonnes d'une matrice avec les fonctions *rownames()* et *colnames()*, respectivement. Vous pouvez aussi vérifier le nombre de lignes et colonnes de la matrice avec les fonctions *nrow()*, *ncol()* et *dim()*.

```
# Par exemple

mat <-cbind(c(10,30,50,60),c(100,50,120,130))
rownames(mat)<-c("placette_1","placette_2","placette_3","placette_4")
colnames(mat)<-c("Avril","Octobre")
mat
```

```
##            Avril Octobre
## placette_1    10     100
## placette_2    30      50
## placette_3    50     120
## placette_4    60     130
```

```
# vous pouvez aussi faire afficher les noms des colonnes (ou lignes) d'une matrice
rownames(mat)
```

```
## [1] "placette_1" "placette_2" "placette_3" "placette_4"
```

```
colnames(mat)
```

```
## [1] "Avril"   "Octobre"
```

```
# Il es possible de nommer les 2 dimensions de la matrice directement
# dans la fonction "matrix" avec l'argument "dimnames".
mat <- matrix(c(10,30,50,60,100,50,120,130), nrow = 4, ncol =2,
              dimnames = list(c("placette_1","placette_2",
                                "placette_3","placette_4"),c("Avril","Octobre")))

# Pour ce il faut créer une "list" avec les 2 vecteurs de noms
# (le premier pour les lignes et le deuxième pour les colonnes)

# On parlera de l'objet de classe "list" plus tard

# Affichez le nombre de lignes et colonnes ou les deux en même temps.
nrow(mat)
```

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

```r
ncol(mat)
```

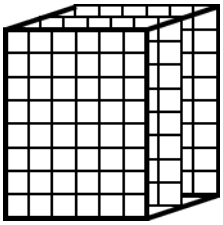
```
## [1] 2
```

```r
dim(mat)
```

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

1.5.3 Array

Une “array” est un objet contenant des valeurs de même classe avec plus de deux dimensions.



Pour une array de 3 dimensions, vous pouvez imaginez un livre: sa “hauteur”" répresente le nombre de lignes, sa “largeur” répresente le nombre de colonnes et son épaisseur c’est la taille de la 3^{eme} dimension.

```r
# Pour une array de 4 lignes, 5 colonnes et 3 "de la 3ême dimension"
# où dans chaque dimension contiennent une séquence de 1 à 20

arr <- array(rep(1:20,4), dim=c(4,5,4))
arr
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
##
## , , 2
##
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
##
## , , 3
##
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
##
## , , 4
##
##      [,1] [,2] [,3] [,4] [,5]
```

```
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
```

Comme vous voyez, rep(1:20,4) repète la séquence "1 à 20". R va en premier completer les colonnes, ensuite les lignes et finalement il cherche la troisième dimension. Donc vous arrez une séquence de 1 à 20 dans les 4 "pages" de la 3ème dimension de l'objet. Vous pouvez toujours ajouter d'autres dimensions aux "arrays" un ajoutant un nouveau élemément à l'argument dim. Ça deviens compliqué à visualiser par contre! :)

1.5.4 Facteurs

Dans certains cas on voudrais avoir des variables catégoriques pour utiliser dans des tests statistiques. Ceci est fait avec les "factors" en utilisant la fonction *factor()*.

```
# Par exemple, le vecteur "sexe" contenant des strings "Mâle" ou "Femelle"

sexe <- c("Mâle","Femelle","Mâle","Mâle","Femelle")
class(sexe)
```

```
## [1] "character"
```

```
sexe
```

```
## [1] "Mâle"    "Femelle" "Mâle"    "Mâle"    "Femelle"
```

```
# Pour le transformer en facteur, on fait simplement:
sexe <- factor(sexe)
class(sexe)
```

```
## [1] "factor"
```

```
sexe
```

```
## [1] Mâle    Femelle Mâle    Mâle    Femelle
## Levels: Femelle Mâle
```

Notez que la classe "factor" peux être nominale ou ordinale.

Une variable nomimale suggère que l'ordre des catégories n'est pas importante. Par exemple il n'y as pas d'ordre entre "Mâle" et "Femelle".

Mais si par exemple vous avez des catégories de température avec ordre ("Froid","Tiède","Chaud") et vous voulez utiliser un test statistique pour variable ordinale, il faut le préciser sur la fonction *factor()*.

```
# Le vecteur "temp" contenant une série de strings "Froid", "Tiède" et "Chaud"

temp <- c("Froid","Chaud","Froid","Tiède","Tiède","Chaud","Froid")


# Pour le transformer en facteur de type ordinale:
temp <- factor(temp, order = TRUE, levels = c("Froid","Tiède","Chaud"))
temp
```

```
## [1] Froid Chaud Froid Tiède Tiède Chaud Froid
## Levels: Froid < Tiède < Chaud
```

```
# avec order = TRUE vous indiquez que l'ordre est importante (par défault il est en mode FALSE)
# l'argument "levels" sert a préciser l'ordre des catégories.
```

Vous pouvez aussi compter combiens d'élements sont de chaque catégorie avec la fonction *summary()*

```
# Combiens de mâles et femelles on trouve dans le vecteur "sexe" ?
summary(sexe)
```

```
## Femelle    Mâle
##       2       3
```

```
# Et sur "temp" ?
summary(temp)
```

```
## Froid Tiède Chaud
##     3     2     2
```

La distinction entre ordinal et nominale est observé quand on compare les catégories.

```
# Par exemple si vous écrivez:

# sexe[2] > sexe[1]

sexe[2]
```

```
## [1] Femelle
## Levels: Femelle Mâle
```

```
sexe[1]
```

```
## [1] Mâle
## Levels: Femelle Mâle
```

```
# C'est à dire, vous affirmez que le deuxième élement du vecteur sexe est supérieure
# au premier élement du vecteur sexe

sexe[2] > sexe[1]
```

```
## Warning in Ops.factor(sexe[2], sexe[1]): '>' not meaningful for factors
```

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

```
# R vous retourne une erreur avec le résultat NA
# Par ce que il y a rien qui indique que la femelle est supérieure au mâle (ou vice-versa)

# Par contre pour le vecteurs de températures, si vous affirmez que
# le deuxième élement du vecteur sexe est supérieure au premier élement du vecteur sexe

temp[2]
```

```
## [1] Chaud
## Levels: Froid < Tiède < Chaud
```

```
temp[1]
```

```
## [1] Froid
## Levels: Froid < Tiède < Chaud
```

```
# C'est à dire que Chaud est supérieure à Froid

temp[2] > temp [1]
```

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

```
# R vous dit TRUE
# Parce que vous avez défini dans "levels" que Chaud viens après Froid.
```

1.5.5 Data frames

Le data frame est un tableau à 2 dimensions où on peux avoir des variables de classes différents. Visuellement c'est très similaire à une matrice mais il fonctionne plutôt comme une liste. C'est un des objets les plus utilisé en R. Vous pouvez créer un "data frame" avec la fonction *data.frame()* :

```
# Par exemple, vous avez des données sur des espèces de poissons.
# Vous pouvez mettre les différentes variables dans un data.frame

# Variables
sexe <- c("Mâle","Femelle","Mâle","Mâle","Femelle")
taille <- c(100, 94, 40, 120, 30)
espece <- c("Esox lucius","Esox lucius", "Barbus barbus","Barbus barbus","Barbus barbus")

# Pour créer le data frame, il faut juste passer les arguments "espece",
# "taille" et "sexe" (les vecteurs) dans la fonction argument
# Par défault les variables vont être mi dans les colonnes (c'est la convention!)
df <-data.frame(espece, sexe, taille)
df
```

```
##          espece    sexe taille
## 1   Esox lucius    Mâle    100
## 2   Esox lucius Femelle     94
## 3 Barbus barbus    Mâle     40
## 4 Barbus barbus    Mâle    120
## 5 Barbus barbus Femelle     30
```

```
# Si vous voulez rajouter une variable ensuite, simplement la combiné
# avec le dataframe avec la fonction "cbind". Pour que ça marche il
# faut qu'ils aient le même nombre de ligne.
age <- c(1, 1 , 2, 3 ,1)

df <- cbind(df, age)
df
```

```
##          espece    sexe taille age
## 1   Esox lucius    Mâle    100   1
## 2   Esox lucius Femelle     94   1
## 3 Barbus barbus    Mâle     40   2
## 4 Barbus barbus    Mâle    120   3
## 5 Barbus barbus Femelle     30   1
```

```
# nombre de lignes/colonnes d'un data frame
dim(df)
```

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

Notez que si vous utiliser la fonction *class()*, R vous indiquera que c'est un data frame.

```
class(df)
```

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

```r
# Si vous voulez savoir la classe de chaque colonne du data frame, il faut faire le suivant
sapply(df,class)
```

```
##    especе      sexe    taille       age
##  "factor"  "factor" "numeric" "numeric"
```

```r
# On parlera des fonctions "apply","lapply","sapply" etc plus tard
```

1.5.6 Listes

Le dernier objet que vous allez voir c'est les listes. Une liste peu contenier des objets de différents classes et de différentes dimensions. Chaque objet est rangé dans un "compartiment" de la liste. Pour créer une liste il faut tout simplement faire appelle à la fonction *list()*:

```r
# Par exemple on va créer une liste avec les objets créés précédemment

liste_a <- list(vec, mat, df)
liste_a
```

```
## [[1]]
##   A   B   C   D
## 100  20  45  60
##
## [[2]]
##            Avril Octobre
## placette_1    10     100
## placette_2    30      50
## placette_3    50     120
## placette_4    60     130
##
## [[3]]
##          especе    sexe taille age
## 1   Esox lucius    Mâle    100   1
## 2   Esox lucius Femelle     94   1
## 3 Barbus barbus    Mâle     40   2
## 4 Barbus barbus    Mâle    120   3
## 5 Barbus barbus Femelle     30   1
```

```r
# Vous pouvez aussi donner des noms à chaque compartiment de la liste

liste_b <-list(echan_inv = vec, ench_pla = mat, camp_peche = df)
# et utiliser le format "list$nom" pour afficher chaque compartiment
liste_b$echan_inv
```

```
##   A   B   C   D
## 100  20  45  60
```

```r
liste_b$ench_pla
```

```
##            Avril Octobre
## placette_1    10     100
## placette_2    30      50
## placette_3    50     120
## placette_4    60     130
```

```
liste_b$camp_peche
```

```
##          especе    sexe taille age
## 1   Esox lucius    Mâle    100   1
## 2   Esox lucius Femelle     94   1
## 3 Barbus barbus    Mâle     40   2
## 4 Barbus barbus    Mâle    120   3
## 5 Barbus barbus Femelle     30   1
```

```
# pour additioner un nouveau élément à la liste, il faut simplement
# la concatener avec cette nouvel élément.
# ATTENTION, si le nouvel élément a plus d'une valeur,
# il faut l'encapsuler dans la fonction list dans la concatenation (voir dessous)

# Si vous ne faites pas ça, R va mettre chaque élément de
# l'objet dans un compartiment différent de la liste
nouv_vec <- seq(1,20,3)
# vous pouvez aussi lui donner un nom spécifique
liste_b <- c(liste_b, vecteur = list(nouv_vec))

liste_b
```

```
## $echan_inv
##   A   B   C   D
## 100  20  45  60
##
## $ench_pla
##            Avril Octobre
## placette_1    10     100
## placette_2    30      50
## placette_3    50     120
## placette_4    60     130
##
## $camp_peche
##          especе    sexe taille age
## 1   Esox lucius    Mâle    100   1
## 2   Esox lucius Femelle     94   1
## 3 Barbus barbus    Mâle     40   2
## 4 Barbus barbus    Mâle    120   3
## 5 Barbus barbus Femelle     30   1
##
## $vecteur
## [1]  1  4  7 10 13 16 19
```

```
# Pour savoir le nombre de compartiments (ou la taille/longueur)
# d'une liste --> fonction length()
length(liste_b)
```

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

```
# pour savoir la  taille des objets dans la liste:
# length() pour les vecteurs et listes
# dim() ,nrwo(), ncol() pour les matrice/array/dataframe
# ex. dimension du compartiment troi
dim(liste_b[[3]])
```

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

```
# nombre de lignes du compartiment 3
nrow(liste_b[[3]])
```

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

2 Module II

2.1 Indexation

Maintenant que vous avez pris connaissances avec les principales classes de variables et types d'objets traiter par R, on va apprendre comment acceder aux différents éléments de ces objets. L'accès aux données contenues dans les objets se fait grâce aux index (positions des données) ou par l'appel explicit des noms de variables.

2.1.1 Indexation des vecteurs

```
# afficher "vec"
vec
```

```
##   A   B   C   D
## 100  20  45  60
```

```
# afficher le premier élément de "vec"
vec[1]
```

```
##   A
## 100
```

```
# si vous voulez afficher plus d'un élément d'un vecteur,
# il faut mettre un vecteur d'indices
# afficher les deux premiers éléments de "vec"
vec[1:2]
```

```
##   A   B
## 100  20
```

```
# afficher le deuxième et le quatrième elément de "vec"
vec[c(2,4)]
```

```
##  B  D
## 20 60
```

```
# il est possible d'indexer les valeurs un désordres
vec[c(3,4,1)]
```

```
##   C   D   A
##  45  60 100
```

2.1.2 Indexation des matrices

```
# afficher "mat"
mat
```

```
##            Avril Octobre
## placette_1    10     100
## placette_2    30      50
## placette_3    50     120
## placette_4    60     130
```

```r
# afficher le 9ème élément de mat
mat[9]
```

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

```r
# les matrice sont indéxés par le format [lignes,colonnes]
# ligne 1, colonne 2
mat[1,2]
```

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

```r
# si vous voulez afficher une ligne (ou colonne entière)
# afficher la deuxième colonne entière
mat[,2]
```

```
## placette_1 placette_2 placette_3 placette_4
##        100         50        120        130
```

```r
# afficher la troisième ligne entière
mat[3,]
```

```
##   Avril Octobre
##      50     120
```

```r
# afficher les trois premiers élément de la deuxième colonne
mat[1:3,2]
```

```
## placette_1 placette_2 placette_3
##        100         50        120
```

```r
# afficher les lignes 1, 2 et 4
mat[c(1,2,4),]
```

```
##            Avril Octobre
## placette_1    10     100
## placette_2    30      50
## placette_4    60     130
```

```r
# afficher les 2 derniers éléments des deux colonnes
mat[3:4,1:2]
```

```
##            Avril Octobre
## placette_3    50     120
## placette_4    60     130
```

```r
# vous pouvez afficher la dernière colononne en utilisant la fonction "ncol"
# (qui donne le nombre de colonnes de l'objet)
mat[,ncol(mat)]
```

```
## placette_1 placette_2 placette_3 placette_4
##        100         50        120        130
```

```r
# la même chose pour les lignes (avec la fonction "nrow")
mat[nrow(mat),]
```

```
##   Avril Octobre
##      60     130
```

```r
# afficher les éléments 1 et 4 de la dernière colonne
mat[c(1,4),ncol(mat)]
```

```
## placette_1 placette_4
```

```
##        100        130
```

```
# afficher les éléments de la deuxième colonne en utilant sont nom
mat[,"Octobre"]
```

```
## placette_1 placette_2 placette_3 placette_4
##        100         50        120        130
```

2.1.3 Indexation des data frames

```
# ligne 1, colonne 2
df[1,2]
```

```
## [1] Mâle
## Levels: Femelle Mâle
```

```
# deux premiers éléments de la quatrième colonne
df[c(1,2),4]
```

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

```
# cinquième élément de la colonne "taille"
df[5,"taille"]
```

```
## [1] 30
```

```
# Vu que le data.frame fonctionne comme une liste,
# une autre manière d'afficher une colonne d'un data frame est avec "$"

# afficher la variable "espèce"
df$especе
```

```
## [1] Esox lucius   Esox lucius   Barbus barbus Barbus barbus Barbus barbus
## Levels: Barbus barbus Esox lucius
```

```
# afficher le troisième élément de la colonne "espèce"
df$especе[3]
```

```
## [1] Barbus barbus
## Levels: Barbus barbus Esox lucius
```

```
# df$variable se comporte comme un vecteur
```

2.1.4 Indexation des listes

```
# afficher le premier compartiment de la liste
liste_b[[1]]
```

```
##   A   B   C   D
## 100  20  45  60
```

```
# après afficher l'objet du compartiment, l'indexation
# dans cette objet suis le format spécifique de l'objet
# pour acceder les éléments 2 à 4 du vecteur contenu
# dans le premier  compartiment de la liste
liste_b[[1]][2:4]
```

```
##  B  C  D
## 20 45 60
```

```r
# pour acceder à la colone espèce du data frame contenu dans le compartiment 3
liste_b[[3]][,1]
```

```
## [1] Esox lucius   Esox lucius   Barbus barbus Barbus barbus Barbus barbus
## Levels: Barbus barbus Esox lucius
```

```r
# autre façon
liste_b[[3]]$espece
```

```
## [1] Esox lucius   Esox lucius   Barbus barbus Barbus barbus Barbus barbus
## Levels: Barbus barbus Esox lucius
```

```r
# autre façon
liste_b[[3]][,"espece"]
```

```
## [1] Esox lucius   Esox lucius   Barbus barbus Barbus barbus Barbus barbus
## Levels: Barbus barbus Esox lucius
```

```r
# Si les compartiments on des noms, vous pouvez les afficher avec "$"
liste_b$camp_peche
```

```
##          espece    sexe taille age
## 1   Esox lucius    Mâle    100   1
## 2   Esox lucius Femelle     94   1
## 3 Barbus barbus    Mâle     40   2
## 4 Barbus barbus    Mâle    120   3
## 5 Barbus barbus Femelle     30   1
```

```r
# Et la l'indexation dans le compartiment ce fait comme l'exemple précédent
liste_b$camp_peche$espece
```

```
## [1] Esox lucius   Esox lucius   Barbus barbus Barbus barbus Barbus barbus
## Levels: Barbus barbus Esox lucius
```

2.1.5 Indexation négative

De la même manière que l'indexation númerique est utiliser pour afficher certains éléments, l'indexation négative est utiliser pour cacher les éléments choisis.

```r
vec
```

```
##   A   B   C   D
## 100  20  45  60
```

```r
# on ne veux pas voir le 2 élément
vec[-2]
```

```
##   A   C   D
## 100  45  60
```

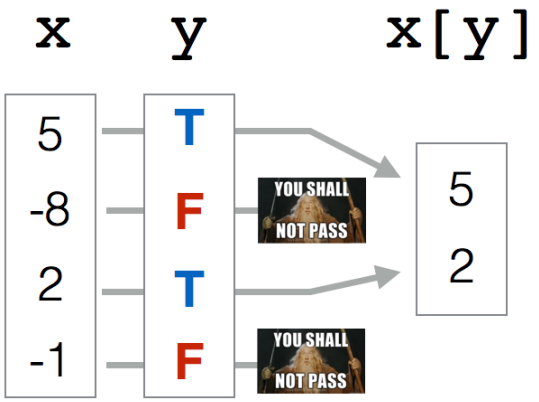
```r
# on ne veux pas voir les éléments 1 et 3
vec[-c(1,3)]
```

```
##  B  D
## 20 60
```

2.2 Indexation par condition

L'indexation par condition, *Logical indexing* en anglais, est un outil très pussaint. C'est à ce moment que vous allez comprendre l'importance des variables booléenes. En gros, ça consiste à fournir un vecteur logique

indicant si chaque élément est inclu (si `TRUE`) ou exclu (si `FALSE`).




```
# Par example
vec <- c(1,2,3,4,5)

vec[c(TRUE,FALSE,TRUE,FALSE,FALSE)]
```

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

Par contre, faire de l'indexation avec des vecteurs logiques créé avec *c()* n'est pas optimale. Une manière plus éfficace de faire de l'indexation est un créant des vecteurs logiques avec des opérations de comparaison.

2.2.1 Opérateurs de comparaison

Voici la liste des opérateurs de comparaison pour obtenir un vecteur logique:

- égale à: `==`
- différent de: `!=`
- strictement supérieur à: `>`
- strictement inférieure à: `<`
- supérieure ou égale à: `>=`
- inférieure ou égale à: `<=`

```
# par example créons des vecteurs de prix et marques de voiture

prix.voiture <- c(5000, 5500, 7000, 13000, 12000, 14000, 15000)
cout.voiture <-c(4000,6000, 6500, 10000, 9000, 11000, 10000)
age.voiture <- c(6, 7 , 4, 3, 2, 1, 1)
marque.voiture <-c("citroen","peugeot","fiat","bmw","citroen","audi",'bmw')

# quelles sont les voitures coutant moins de 10000 ?
prix.voiture < 10000
```

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

```
# comme vous voyez, le résultat est un vecteur logique où R nous indique
# TRUE quand la valeur de l'élément est inférieur à 10000 et #FALSE quand
# la valeur est égale ou supérieure à 10000.

# quelles sont les voitures de marque citroen?
marque.voiture == 'citroen'
```

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

```
# quelles sont les voitures qui ne sont pas de marque citroen?
marque.voiture != 'citroen'
```

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

```
# vous pouvez aussi comparer deux vecteurs de la même longueur. R vas les
# comparer élément par élément  quelles sont les voitures que le cout
# de production est supérieure au prix?
prix.voiture < cout.voiture
```

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

```
# dès que vous avez un vecteur logique, vous pouvez l'utiliser
# pour extraire les valeurs du vecteur désiré.
# quelles sont les voitures coutant moins de 10000 ?
prix.voiture[prix.voiture < 10000]
```

```
## [1] 5000 5500 7000
```

```
# Combiens coutent les voitures de moins de 4 ans?
prix.voiture[age.voiture < 4]
```

```
## [1] 13000 12000 14000 15000
```

2.2.2 Opérateurs logiques

Les conditions peuvent être complexifiées à l'aides des opérateurs logiques:

- et `&`
- ou `|`
- n'est pas `!`
- est contenu dans `%in%`

```
# quelles sont les voitures qui coutent 5000 euros ou plus mais moins de 13000 euros?
prix.voiture >= 5000 & prix.voiture < 13000
```

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

```
# quelles sont les marques de ces voitures?
marque.voiture[prix.voiture >= 5000 & prix.voiture < 13000]
```

```
## [1] "citroen" "peugeot" "fiat"    "citroen"
```

```
# quel est l'âge des voitures qui ne sont pas citroen et qui coutent plus de 10000 euors?
age.voiture[marque.voiture != 'citroen' & prix.voiture > 10000]
```

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

```
# et leurs marques?
marque.voiture[marque.voiture != 'citroen' & prix.voiture > 10000]
```

```
## [1] "bmw"  "audi" "bmw"
```

```
# vous pouvez combiner plusieurs opérateurs logiques

# Indique moi quelles sont les voitures de marque citoren OU fiat qui
```

```
# coutent moins de 10000 ?
(marque.voiture == 'citroen' | marque.voiture == 'fiat') & prix.voiture < 10000
```

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

```
# quel sont les âges de ces voitures?
age.voiture[(marque.voiture == 'citroen' | marque.voiture == 'fiat') & prix.voiture < 10000]
```

```
## [1] 6 4
```

```
# notez que j'ai mi le parenthèse autour de la condition sur les
# marques de voiture.Si je n'avais pas mi le parenthèse, R aurais
# interpretér la question comme "Indique moi les voitures de marque
# citroen (n'importe lesquelles) OU les voitures de marque
# fiat qui content moins de 10000 euros?"

# Indiquez moi les voitures de marque "citroen", "fiat" ou "peugeot"?
marque.voiture == "citroen" | marque.voiture == 'fiat' | marque.voiture == 'peugeot'
```

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

```
# C'est quand même long d'écrire toutes ces conditions.
# Une manière plus facile est d'utiliser l'opérateur "%in%"
marque.voiture %in% c('citroen','fiat','peugeot')
```

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

```
# R interprète ça comme "quels sont les éléments dans
# "marque voiture" qui sont aussi présent dans c("citroen","fiat","peugeot")
```

2.2.3 Comptages et percentages de vecteurs logiques

La plus part des fonctions de R vont interpréter `TRUE` comme 1 et `FALSE` comme 0. Ceci permet de répondre à des questions comme "combiens de voiture sont plus chers que 10000?" avec la fonction *sum()*. "Quel est le pourcentage de voitures avec un prix supérieure à 5000 ET inférieure à 12000?" avec la fonction *mean()*.

```
# Combiens de voiture sont plus chers que 10000?
sum(prix.voiture > 10000)
```

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

```
# Quel est le pourcentage de voitures avec un prix supérieure à 5000 ET inférieure à 12000?
mean(prix.voiture > 5000 & prix.voiture < 12000)*100
```

```
## [1] 28.57143
```

```
# Regardez le vecteur logique
prix.voiture > 5000 & prix.voiture < 12000
```

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

```
# R interprète le TRUE comme 1 et le FALSE comme 0, si vous transformez
# le vecteurs en une variable numérique
as.numeric(prix.voiture > 5000 & prix.voiture < 12000)
```

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

```
# Il y a 2 TRUE sur un total de 7 éléments
(2/7) * 100
```

```
## [1] 28.57143
```

2.2.4 Autres fonctions importantes pour l'indexation

- *is.na()*
- *duplicated()*
- *which()*
- *is.finite()*
- *which.min()*
- *which.max()*

```
vec <- c(5,NA ,3, NA, 20, NA, 2, 3, 5, 7, 8, NA)
# Quelles sont les valeurs mancantes ?
is.na(vec)
```

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

```
# Combiens il y a til de valeur mancantes?
sum(is.na(vec))
```

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

```
# Quelles sont les éléments dupliqués?
duplicated(vec)
```

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

```
# notez que le premier élément qu'il rencontre reçois un FALSE et
# c'est seulement à partir du deuxième que cet élément va recevoir un
# TRUE. Par exemple le premier 3 reçois un FALSE et le deuxième trois
# reçois un TRUE les NA aussi reçoivent des TRUE à partir du deuxième NA

# finalment la fonction which retournes tout les indexes des
# éléments qui satisfont une condition. Par exemple, dans vec
# les NA sont les éléments 2, 4, 6 et 12 Et c'est exactement ce que which retourne:
which(is.na(vec))
```

```
## [1]  2  4  6 12
```

```
# si on veux savoir l'index des voitures citroens dans le vecteur marque.voitures
which(marque.voiture == 'citroen')
```

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

```
# is.finite() retourne toutes les valeurs qui ne sont pas NA, NaN, Inf, -Inf
# Donc si on demande a which(), quels éléments
# qui sont fini il va nous répondre tout saut 2,4,6 et 12
which(is.finite(vec))
```

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

```
vec <- c (1,20, 4 ,5, 6 , 8 , 9, 0.5)
# Où est la position de l'élément avec la valeur maximale?

which.max(vec)
```

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

```r
vec[which.max(vec)]
```

```
## [1] 20
```

```r
# Et la plus petite valeur?
which.min(vec)
```

```
## [1] 8
```

```r
vec[which.min(vec)]
```

```
## [1] 0.5
```

```r
# Une autre façon de trouver la valeur maximale et minimale est en utilisant max() et min()
min(vec)
```

```
## [1] 0.5
```

```r
max(vec)
```

```
## [1] 20
```

```r
#Et pour trouver ça position
which(vec == min(vec))
```

```
## [1] 8
```

```r
which(vec == max(vec))
```

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

```r
# En gros
# which.min(vec) = which(vec == min(vec))
# which.max(vec) = which(vec == max(vec))
# min(vec) = vec[which.min(vec)]
# max(vec) = vec[which.max(vec)]
```

2.2.5 Indexation par comparaison avec matrices et data.frames

```r
# Pour les matrices et les data.frames c'est la même logique

# combinez les vecteurs d'age, prix et cout des voitures
mat.voiture <-cbind(age.voiture, cout.voiture, prix.voiture)
mat.voiture
```

```
##      age.voiture cout.voiture prix.voiture
## [1,]           6         4000         5000
## [2,]           7         6000         5500
## [3,]           4         6500         7000
## [4,]           3        10000        13000
## [5,]           2         9000        12000
## [6,]           1        11000        14000
## [7,]           1        10000        15000
```

```r
colnames(mat.voiture) <-c("age","cout","prix")

# indexez la matrice en séléctionnant seulement les voitures de 3 ans ou plus
# L'âge des voiture est dans le première colonne, donc vous
# indexez en utilisant "mat.voiture[,1]"
mat.voiture[mat.voiture[,1]>3,]
```

```
##      age cout prix
## [1,]   6 4000 5000
## [2,]   7 6000 5500
## [3,]   4 6500 7000
```

```
# trouvez les voitures où le coût de production est égale à 10000 euros
# ET que le prix de vente est inférieure à 14000 euros?

# ici c'est en utilisant le nombre de la colonne
mat.voiture[mat.voiture[,2] == 10000 & mat.voiture[,3] < 14000,]
```

```
##   age  cout  prix
##     3 10000 13000
```

```
# en utilisant le nom de la colonne
mat.voiture[mat.voiture[,"cout"] == 10000 & mat.voiture[,"prix"] < 14000,]
```

```
##   age  cout  prix
##     3 10000 13000
```

```
df.voiture <- data.frame(mat.voiture, marque = marque.voiture, stringsAsFactors = FALSE)

# affichez toutes les propriétés des voitures citroens
df.voiture[df.voiture$marque == "citroen",]
```

```
##   age cout  prix  marque
## 1   6 4000  5000 citroen
## 5   2 9000 12000 citroen
```

```
# affichez toutes les propriétés des voitures citroens et bmw
df.voiture[df.voiture$marque %in% c("citroen","bmw"),]
```

```
##   age  cout  prix  marque
## 1   6  4000  5000 citroen
## 4   3 10000 13000     bmw
## 5   2  9000 12000 citroen
## 7   1 10000 15000     bmw
```

```
# en utilisant la fonction subset
# subset (df, subset = la condition, select = les colonnes désirées)
subset(df.voiture, subset = marque %in% c("citroen","bmw"))
```

```
##   age  cout  prix  marque
## 1   6  4000  5000 citroen
## 4   3 10000 13000     bmw
## 5   2  9000 12000 citroen
## 7   1 10000 15000     bmw
```

```
# vous pouvez maintenant combiner ces fonctions pour calculer des stats
# par exemple, quel est le prix moyene des voitures citroen et bmw dans la base de données?
vec <- subset(df.voiture, subset = marque %in% c("citroen","bmw"), select = 'prix')

mean(vec)
```

```
## Warning in mean.default(vec): argument is not numeric or logical: returning
## NA
```

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

```r
# Pourquoi ca ne marche pas??
class(vec)
```

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

```r
# Parce que la fonction subset retourne un data.frame (même si c'est juste un vecteur)
# Pour acceder au vecteur dans le data frame, il faut utiliser le "$"
mean(vec$prix)
```

```
## [1] 11250
```

```r
#sinon vous pouvez le faire en une ligne en utilisant $

mean(df.voiture$prix[df.voiture$marque %in% c("citroen","bmw")])
```

```
## [1] 11250
```

2.2.6 Autres fonctions utiles

- *with()*
- *unique()*
- *order()*
- *table()*
- *na.omit()*

La fonction *with()* facilite les opérations avec plusieurs colonnes d'un data.frame. La fonction *unique()* retourne toute les valeurs uniques d'un vecteur. La fonction *order()* retourne des indices avec l'ordre croissantes des éléments. Ce vecteur d'indices peux-être utiliser pour changer l'ordre d'un vecteur, matrice ou data.frame. La fonction *table()* retourne un comptage du nombre de fois que chaque catégorie apparait dans un vecteur. Si il y a plusieurs vecteurs de catégorie, la fonction retourne un tableau de contigence avec le nombre d'élément dans chaque combination de catégories entre facteurs. La fonction *na.omit()* elimine toutes les valeurs mancantes. Dans un datatrame ou une matrice, elle élimine toutes les lignes contenant des valeurs mancantes.

```r
# Si vous voulez savoir combien est la marge dans une vente de voiture et après diviser le résultats pa
(df.voiture$prix - df.voiture$cout) /  df.voiture$age
```

```
## [1]  166.66667  -71.42857  125.00000 1000.00000 1500.00000 3000.00000
## [7] 5000.00000
```

```r
# la fonction with() vous permet de faire le même calcule avec moins d'écriture
with(df.voiture, (prix - cout)/ age)
```

```
## [1]  166.66667  -71.42857  125.00000 1000.00000 1500.00000 3000.00000
## [7] 5000.00000
```

```r
# Combiens il y a t'il de marque de voiture?

# Extraire les valeurs uniques du vecteur des "marques"
marques <- unique(df.voiture$marque)
# Calculer la longueur du vecteur
length(marques)
```

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

```r
# en une ligne
length(unique(df.voiture$marque))
```

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

```r
# Donnees-moi l'indice d'ordre croissante des couts des voitures
idx_cout <- order(df.voiture$cout)
df.voiture$cout
```

```
## [1]  4000  6000  6500 10000  9000 11000 10000
```

```r
idx_cout
```

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

```r
# Organiser le tableau données avec l'ordre des couts de voiture
df.voiture[idx_cout,]
```

```
##   age  cout  prix  marque
## 1   6  4000  5000 citroen
## 2   7  6000  5500 peugeot
## 3   4  6500  7000    fiat
## 5   2  9000 12000 citroen
## 4   3 10000 13000     bmw
## 7   1 10000 15000     bmw
## 6   1 11000 14000    audi
```

```r
# Par défault order mets un ordre croissante. Si vous voulez l'ordre décroissante
df.voiture[order(df.voiture$cout, decreasing =TRUE),]
```

```
##   age  cout  prix  marque
## 6   1 11000 14000    audi
## 4   3 10000 13000     bmw
## 7   1 10000 15000     bmw
## 5   2  9000 12000 citroen
## 3   4  6500  7000    fiat
## 2   7  6000  5500 peugeot
## 1   6  4000  5000 citroen
```

```r
# Si vous voulez organiser le tableau par plusieurs colonnes. Par exemple, organiser
# l'ordre des cout et ensuite reordonner par l'ordre des prix
df.voiture[order(df.voiture$cout, df.voiture$prix),]
```

```
##   age  cout  prix  marque
## 1   6  4000  5000 citroen
## 2   7  6000  5500 peugeot
## 3   4  6500  7000    fiat
## 5   2  9000 12000 citroen
## 4   3 10000 13000     bmw
## 7   1 10000 15000     bmw
## 6   1 11000 14000    audi
```

```r
# Il va donner toujours priorité à la première colonne
```

2.3 Gestion du repertoire de travail et import/export des données

2.3.1 Répertoire de travail

Pour vérifier quel est le "working directory" actuel, utilisez la fonction *getwd()*

```r
getwd()
```

```
## [1] "C:/Users/Renato/Documents/github/atelierR"
```

Pour avoir la liste des fichiers dans le "working directory" actuel, utilisez la fonction *liste.files()*

```
list.files()
```

```
##  [1] "Atelier_R_Outline.Rmd" "atelierR.Rproj"
##  [3] "donnes"                "exercices"
##  [5] "fig"                   "Introduction_R.pdf"
##  [7] "Introduction_R.Rmd"    "LICENSE"
##  [9] "README.md"             "script"
```

Pour changer votre working directory, utilisez la fonction *setwd()* avec l'adresse du nouveau repertoire en format caractère

```
# Par exemple
# setwd("D:/work/atelier/")
```

Une autre manière est de combiner la fonction *getwd* avec la fonction *paste()*. La fonction paste sers à concatener des chaines de caractères.

```
paste('chien', 'et','chat')
```

```
## [1] "chien et chat"
```

```
# Vous pouvez definir un séparateur entre les différents éléments
paste('chien', 'et','chat', sep ='.')
```

```
## [1] "chien.et.chat"
```

```
paste('chien', 'et','chat', sep ='+')
```

```
## [1] "chien+et+chat"
```

```
paste('chien', 'et','chat', sep ='/')
```

```
## [1] "chien/et/chat"
```

```
# Vous voyez où on s'en vas?

# setwd(paste(getwd(),dir, sep = '/'))
# Où "dir" seraos un sous-répertoire localisé à l'intérieur du répertoire de travail
```

2.3.2 Imports de données

La fonction de base pour importer vos données dans R est *read.table()*. Le format le plus générique et facilement acessible est le format **".txt"**.

```
# Importons un fichier txt

df <- read.table(file ='donnees_exemple.txt')
```

Il y a une erreur. Allez vérifier le fichier. les colonnes sont séparées par quoi? Par des " ; " Dans *read.table()* la valeur défault de l'argument de séparation `sep = ' '`. Donc ça ne peux pas fonctionner. Il faut changer l'argument sep pour qu'il identifie les ";":

```
df <- read.table(file ='donnees_exemple.txt', sep=';')
```

Si votre fichier n'est pas dans le repertoire de travail, vous devez fournir le chemin absolu du fichier. Vous pouvez également fournir un chemin sur le web pour télécharger vos données.

```
# par exemple:
df <- read.table(file ='https://raw.githubusercontent.com/RenatoHS/parallel_null_modelling/master/data/
```

Si vous voulez voir tout les argument qu'une fonction possède, vous pouvez utilez la fonction *args()*:

```
args(read.table)
```

```
## function (file, header = FALSE, sep = "", quote = "\"'", dec = ".",
##     numerals = c("allow.loss", "warn.loss", "no.loss"), row.names,
##     col.names, as.is = !stringsAsFactors, na.strings = "NA",
##     colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
##     fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
##     comment.char = "#", allowEscapes = FALSE, flush = FALSE,
##     stringsAsFactors = default.stringsAsFactors(), fileEncoding = "",
##     encoding = "unknown", text, skipNul = FALSE)
## NULL
```

Affichez les premières lignes du tableau avec *head()*:

```
head(df)
```

Affichez la structure du tableau avec la fonction *str()*:

```
str(df)
```

Est-ce que vous voyez un problème? Toutes les colonnes sont affichées comme des "facteurs". Ceci viens du fait que la première ligne du fichier "donnees_exemple.txt" est une ligne d'entêtes. Par defaul la fonction assume qu'il n'y a pas de ligne d'entête. Pour changer ça il faut définir l'argument *header = TRUE*. Si vous voulez aussi que les colonnes de caractères ne soient pas considérées des facteurs, il faut définir l'argument *stringAsFactors = FALSE*

```
df <- read.table(file ='donnees_exemple.txt', sep=';', header= TRUE, stringsAsFactors = FALSE)

# Vérifiez la structure de "df"
str(df)
# Maitenant chaque colonne a la classe apropriée aux types d'élément qu'elle contient
```

Si vous voulez importer des données '.csv. il faut utiliser la fonction *read.csv()* qui est exactement pareil à read.table() avec l'exeception que le défault de l'argument de séparation est `sep = ';'`.

2.3.3 Export de données

Pour exporter vos données il y a plusieurs manières. La fonction pair de "read.table" est la fonction *write.table*

```
# creez un sous-group de df avec seulement les lignes qui contiennent "Perca fluviatilis"
# df2 <- subset(df, subset = sp_nom == "Perca fluviatilis")

# Exportez df2 en .txt dans votre répertoire en spéficiant que le séparateur est une tabulation (\t)
# write.table(df2, file ='donnes_perca.txt', sep ='\t')

# Vérifiez votre nouveau fichier.
```

2.3.4 Autres fonctions de gestion du répertoire de travail

- *ls()*
- *rm()*

La fonction *ls()* permet d'afficher tout les objets que vous aves dans votre environnement de travail sur R

```
#ceci retourne un vecteur de caractère avec le nom de tous les objets de votre environnement de travail
ls()
```

```
##  [1] "age"           "age.voiture"   "arr"           "cout.voiture"
```

```
##  [5] "dates"          "df"             "df.voiture"     "espece"
##  [9] "idx_cout"       "liste_a"        "liste_b"        "marque.voiture"
## [13] "marques"        "mat"            "mat.voiture"    "nouv_vec"
## [17] "prix.voiture"   "sexe"           "taille"         "temp"
## [21] "var_bool"       "var_string"     "vec"            "x"
## [25] "y"              "z"
```

La fonction *rm()* permet de "remove" un ou plusieurs objets de votre environnement de travail sur R

```
# ceci éfface les objets "df" et "df2"
# rm(df,df2)

# si vous voulez tout effacer
rm(list = ls())
```

3 MODULE III

3.1 Boucles

Une des "règles d'or"" de la programmation est de jamais répéter une tâche. Vous pouvez la répéter mais c'est certainement une perte de temps. Un des outils de la programmation pour répéter des tâches est la boucle (*for loop* en anglais).

La structure générale d'un loop est

```
for(loop.object in loop.vector) {

  LOOP.CODE

  }
```