Atelier - Introduction à l'environnement R

Renato Henriques-Silva

February 12, 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 éxecutés 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
```

```r
# Substraction
5 - 2
```

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

```r
# Multiplication
5 * 2
```

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

```r
# 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 3$^{\text{ème}}$ variable).

```r
# afféctation de la variable y

y <- 10

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

```r
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`.

```r
var_bool <- FALSE
var_bool
```

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

```r
var_bool <- F
var_bool
```

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

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

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

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

```r
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()* .

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

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

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

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

```r
# 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
```

```
# 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"
```

```
# 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()*

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

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

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

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

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

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

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

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

```
# 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))
```

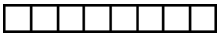
```
## [1] "10" "2"
```

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.

```
# 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
```

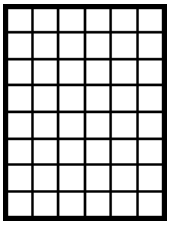
Pour afficher la taille (ou la longueur) d'un vecteur, il faut utiliser la fonction *length()*.

```
# 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)
```

```r
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
```

```r
# 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
```

```r
# 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()*.

```r
# 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
```

```r
# 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"
```

```r
colnames(mat)
```

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

```r
# 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")))
```

```r
# 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)
```

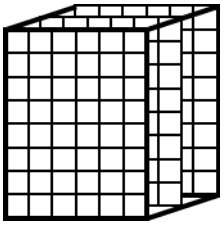
```
## [1] 2
```

```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")
```

```r
# 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
```

```r
# 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()*

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

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

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

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

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

```r
# Par exemple si vous écrivez:

# sexe[2] > sexe[1]

sexe[2]
```

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

```r
sexe[1]
```

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

```r
# 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
# 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
```

```r
temp[1]
```

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

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

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

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

```r
# 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()* :

```r
# 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
```

```r
# 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
```

```r
# 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.

```r
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)
```

```r
# 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
```

```r
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
# 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
##          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
##
## $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
```

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

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

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

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

```
# 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
```

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

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

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

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

```
# 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
```

```
# 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
```

```r
# 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

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

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

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

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

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

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

```r
# 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
```

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

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

```r
# 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
```

```
# 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
```

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

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

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

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

```
# 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
```

```
# 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.

```
vec
```

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

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

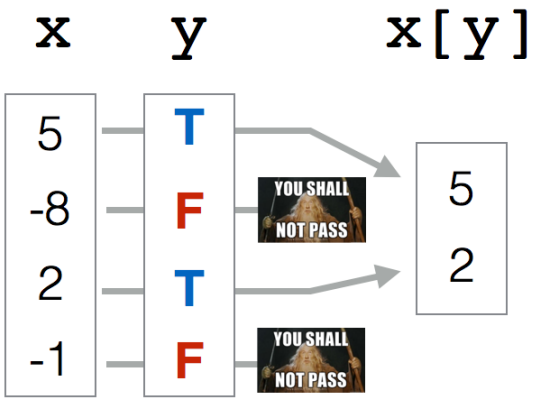
```
##   A   C   D
## 100  45  60
```

```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`).




```r
# 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: `<=`

```r
# par example créons des vecteurs de characteristics de chats

chats.nom <-c("Fluffy","Billy","Minette","Patoune","Maya","Pilou","Luna", "Pacha","Minou","Gribouille")
```

```r
chats.sexe <-c("F", "M", "F", "F", "F", "M", "F", "M", "M", "M")
chats.poid <- c(3.82,3.55,5.6,4,3.66,4.03,3.76,3.52,3.91,6.48)
chats.age <-c(10,4,3,2,11,1,8,9,7,6)
chats.sommeil <-c(18,13,12,13,18,14,15,17,17,18)
chats.race <-c("Persan","Siamois","Main Coon","Goutière","Siamois","Goutière","Angora","Persan","Persan

# quels sont les chats qui dorment moins de 15h par jour ?
chats.sommeil < 15
```

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

```r
# comme vous voyez, le résultat est un vecteur logique où R nous indique
# TRUE quand la valeur de l'élément est inférieur à 15 et #FALSE quand
# la valeur est égale ou supérieure à 15.

# quels sont les chats femelles?
chats.sexe == 'F'
```

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

```r
# quels sont les chats qui ne sont pas femelles?
chats.sexe != 'F'
```

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

```r
# vous pouvez aussi comparer deux vecteurs de la même longueur. R vas les
# comparer élément par élément.
# Par example, vous avez un vecteur des poids des chats à  t2.
chats.poid.t2 <- c( 4.515, 3.305, 5.590, 3.890, 2.650, 3.135, 2.935, 3.085, 4.265, 6.920)
# Quels chats ont grossi entre t1 et t2?
chats.poid.t2 > chats.poid
```

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

```r
# dès que vous avez un vecteur logique, vous pouvez l'utiliser
# pour extraire les valeurs du vecteur désiré.
# Quel est l'âge des chats qui ont moins de 7 ans?
chats.age[chats.age < 7]
```

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

```r
# Ou sinon vous pouvez utiliser un vecteur logique construit avec un vecteur pour extraire des informat
# Combiens d'heures dorment les chats qui ont 8 ans ou plus?
chats.sommeil[chats.age >= 8]
```

```
## [1] 18 18 15 17
```

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%`

```
# quels sont les chats qui pèsent 3.6kg et plus mains moins de 4kg?
chats.poid >= 3.6 & chats.poid < 4
```

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

```
# quels sont les noms de ces chats?
chats.nom[chats.poid >= 3.6 & chats.poid < 4]
```

```
## [1] "Fluffy" "Maya"   "Luna"   "Minou"
```

```
# Combiens d'heures dorment les chats mâle qui pèsent moins de 4kg ?
chats.sommeil[chats.sexe != 'M' & chats.poid < 4]
```

```
## [1] 18 18 15
```

```
# et c'est quoi leurs noms?
chats.nom[chats.sexe != 'M' & chats.poid < 4]
```

```
## [1] "Fluffy" "Maya"   "Luna"
```

```
# vous pouvez combiner plusieurs opérateurs logiques

# Indiquez les chats de race Siamois ou Persan qui
# ont plus de 7 ans?
(chats.race == 'Siamois' | chats.race == 'Persan') & chats.age > 7
```

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

```
# Comment ils s'appellent?
chats.nom[(chats.race == 'Siamois' | chats.race == 'Persan') & chats.age > 7]
```

```
## [1] "Fluffy" "Maya"   "Pacha"
```

```
# 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 "Indiquez moi les chats Siamos (tous) et
# seulement les chats Persans qui ont moins de 7 ans.

# Indiquez moi les chats de race "Main Coon", "Angora" et les chats sans race ("Goutière")
chats.race == "Goutière" | chats.race == 'Main Coon' | chats.race == 'Angora'
```

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

```
# C'est quand même long d'écrire toutes ces conditions.
# Une manière plus facile est d'utiliser l'opérateur "%in%"
chats.race %in% c('Goutière','Main Coon','Angora')
```

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

```
# R interprète ça comme "quels sont les éléments dans
# "chats.race" qui sont aussi présent dans c('Goutière','Main Coon','Angora')
```

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()*.

```r
# Combiens de chats ont plus de 8 ans?
sum(chats.age >= 8)
```

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

```r
# Quel est le pourcentage de chats qui dorment entre 12h et 15h ?
mean(chats.sommeil > 12 &  chats.sommeil < 15)*100
```

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

```r
# Regardez le vecteur logique
chats.sommeil > 12 &  chats.sommeil < 15
```

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

```r
# R interprète le TRUE comme 1 et le FALSE comme 0, si vous transformez
# le vecteurs en une variable numérique
as.numeric(chats.sommeil > 12 &  chats.sommeil < 15)
```

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

```r
# Il y a 3 TRUE sur un total de 10 éléments
(3/10) * 100
```

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

2.2.4 Autres fonctions importantes pour l'indexation

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

```r
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
```

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

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

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

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

```r
# 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
```

```r
# 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
```

```r
# si on veux savoir l'index des chats de race Persan dans le vecteur chats.race
which(chats.race == 'Persan')
```

```
## [1] 1 8 9
```

```r
# 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
```

```r
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

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

# combinez les vecteurs d'age, poids et heures de sommeils
# des chats dans la matrice mat.chats
mat.chats <-cbind(chats.age, chats.poid, chats.sommeil)
mat.chats
```

```
##       chats.age chats.poid chats.sommeil
##  [1,]        10       3.82            18
##  [2,]         4       3.55            13
##  [3,]         3       5.60            12
##  [4,]         2       4.00            13
##  [5,]        11       3.66            18
##  [6,]         1       4.03            14
##  [7,]         8       3.76            15
##  [8,]         9       3.52            17
##  [9,]         7       3.91            17
## [10,]         6       6.48            18
```

```
colnames(mat.chats) <-c("age","poid","sommeil")

# Indexez la matrice en séléctionnant seulement les chats de moins de 5 ans
# L'âge des chats est dans la première colonne, donc vous
# indexez en utilisant "mat.chats[,1]"
mat.chats[mat.chats[,1]<5,]
```

```
##      age poid sommeil
## [1,]   4 3.55      13
## [2,]   3 5.60      12
## [3,]   2 4.00      13
## [4,]   1 4.03      14
```

```
# Trouvez les chats de moins de 10 ans qui dorment exactement 18h par jour

# En utilisant le nombre de la colonne
mat.chats[mat.chats[,1] < 10 & mat.chats[,3] == 18,]
```

```
##     age    poid sommeil
##    6.00    6.48   18.00
```

```
# en utilisant le nom de la colonne
mat.chats[mat.chats[,"age"] < 10 & mat.chats[,"sommeil"] == 18,]
```

```
##     age    poid sommeil
##    6.00    6.48   18.00
```

```
df.chats <- data.frame(nom = chats.nom, sexe = chats.sexe, race =chats.race, mat.chats, stringsAsFactors
df.chats
```

```
##           nom sexe      race age poid sommeil
## 1      Fluffy    F    Persan  10 3.82      18
## 2       Billy    M   Siamois   4 3.55      13
## 3     Minette    F Main Coon   3 5.60      12
## 4     Patoune    F  Goutière   2 4.00      13
## 5        Maya    F   Siamois  11 3.66      18
## 6       Pilou    M  Goutière   1 4.03      14
```

```
## 7        Luna    F    Angora   8 3.76      15
## 8       Pacha    M    Persan   9 3.52      17
## 9       Minou    M    Persan   7 3.91      17
## 10 Gribouille    M Main Coon   6 6.48      18
```

```
# affichez toutes les propriétés des chats de race Persan
df.chats[df.chats$race == "Persan",]
```

```
##      nom sexe   race age poid sommeil
## 1 Fluffy    F Persan  10 3.82      18
## 8  Pacha    M Persan   9 3.52      17
## 9  Minou    M Persan   7 3.91      17
```

```
# affichez toutes les propriétés des chats de race Persan et des chats sans race
df.chats[df.chats$race %in% c("Persan","Goutière"),]
```

```
##       nom sexe     race age poid sommeil
## 1  Fluffy    F   Persan  10 3.82      18
## 4 Patoune    F Goutière   2 4.00      13
## 6   Pilou    M Goutière   1 4.03      14
## 8   Pacha    M   Persan   9 3.52      17
## 9   Minou    M   Persan   7 3.91      17
```

```
# en utilisant la fonction subset
# subset (df, subset = la condition, select = les colonnes désirées)
subset(df.chats, subset = race %in% c("Persan","Goutière"))
```

```
##       nom sexe     race age poid sommeil
## 1  Fluffy    F   Persan  10 3.82      18
## 4 Patoune    F Goutière   2 4.00      13
## 6   Pilou    M Goutière   1 4.03      14
## 8   Pacha    M   Persan   9 3.52      17
## 9   Minou    M   Persan   7 3.91      17
```

```
# l'argument select dans "subset" indique quelles variables du data.frame on voudrais extraire (par déf
df2 <- subset(df.chats, subset = race %in% c("Persan","Goutière"), select = 'sommeil')

# Quel est la moyenne d'heure de sommeil des chats de race Persan et sans race
mean(df2)
```

```
## Warning in mean.default(df2): argument is not numeric or logical: returning
## NA
```

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

```
# Pourquoi ca ne marche pas??
class(df2)
```

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

```
# 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 "$"
mean(df2$sommeil)
```

```
## [1] 15.8
```

```
#sinon vous pouvez le faire en une ligne en utilisant $
mean(df.chats$sommeil[df.chats$race %in% c("Persan","Goutière")])
```

```
## [1] 15.8
```

2.2.6 Autres fonctions utiles

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

La fonction *unique()* retourne toute les valeurs uniques d'un vecteur. La fonction *sort()* organise les valeurs en ordre croissante. 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.

```
# Combiens il y a t'il de race de chats?

# Extraire les valeurs uniques du vecteur des "chats.race"
races <- unique(df.chats$races)
# Calculer la longueur du vecteur
length(races )
```

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

```
# en une ligne
length(unique(df.chats$race))
```

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

```
# Donnees-moi l'indice d'ordre croissante des poids des chats
idx_poids <- order(df.chats$poid)
idx_poids
```

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

```
# Organiser le tableau données avec l'ordre des poids des chats
df.chats[idx_poids,]
```

```
##           nom sexe      race age poid sommeil
## 8       Pacha    M    Persan   9 3.52      17
## 2       Billy    M   Siamois   4 3.55      13
## 5        Maya    F   Siamois  11 3.66      18
## 7        Luna    F    Angora   8 3.76      15
## 1      Fluffy    F    Persan  10 3.82      18
## 9       Minou    M    Persan   7 3.91      17
## 4     Patoune    F  Goutière   2 4.00      13
## 6       Pilou    M  Goutière   1 4.03      14
## 3     Minette    F Main Coon   3 5.60      12
## 10 Gribouille    M Main Coon   6 6.48      18
```

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

```
##           nom sexe      race age poid sommeil
## 10 Gribouille    M Main Coon   6 6.48      18
## 3     Minette    F Main Coon   3 5.60      12
## 6       Pilou    M  Goutière   1 4.03      14
## 4     Patoune    F  Goutière   2 4.00      13
```

```
## 9       Minou    M    Persan   7 3.91      17
## 1      Fluffy    F    Persan  10 3.82      18
## 7        Luna    F    Angora   8 3.76      15
## 5        Maya    F   Siamois  11 3.66      18
## 2       Billy    M   Siamois   4 3.55      13
## 8       Pacha    M    Persan   9 3.52      17
```

```
# Si vous voulez organiser le tableau par plusieurs colonnes. Par exemple, organiser
# l'ordre des heures de sommeil et ensuite reordonner par l'ordre des poids
df.chats[order(df.chats$sommeil, df.chats$poid, decreasing =TRUE),]
```

```
##           nom sexe      race age poid sommeil
## 10 Gribouille    M Main Coon   6 6.48      18
## 1      Fluffy    F    Persan  10 3.82      18
## 5        Maya    F   Siamois  11 3.66      18
## 9       Minou    M    Persan   7 3.91      17
## 8       Pacha    M    Persan   9 3.52      17
## 7        Luna    F    Angora   8 3.76      15
## 6       Pilou    M  Goutière   1 4.03      14
## 4     Patoune    F  Goutière   2 4.00      13
## 2       Billy    M   Siamois   4 3.55      13
## 3     Minette    F Main Coon   3 5.60      12
```

```
# 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()*

```
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.pdf" "Atelier_R_Outline.Rmd"
##  [3] "atelierR.Rproj"        "donnes"
##  [5] "exercices"             "exercices_reso"
##  [7] "fig"                   "Introduction_R.pdf"
##  [9] "Introduction_R.Rmd"    "LICENSE"
## [11] "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"
```

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

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

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

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

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

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

```r
# 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"**.

```r
# 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 ";":

```r
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.

```r
# par exemple:
df <- read.table(file ='https://raw.githubusercontent.com/RenatoHS/
                 parallel_null_modelling/master/data/lake.comm.txt', sep=';')
```

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

```r
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()*:

```r
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'exception 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. Pour copier un .txt il y a la fonction *write.table()*. Il existe également la fonction *write.csv()* pour créer un fichier avec sep = ';'.

```
# 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()
```

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 Les conditions If/Else

Maintenant que vous savez comme utiliser les opérations de comparaison et les opérations logiques, vous pouvez les utilisés dans le cadre des conditions *if()* et *else()*. Ceci permet que vos scripts et fonctions ce comportement de manières différentes selon les objets qui traîtent. La structure générale d'une condition *if* est:

IF

```
if(condition){

  CODE

}
```

Par exemple, si on veux que notre script imprime si une valeur est négative:

```
x <- -3

if(x < 0){

  print('x est un nombre négatif')

}
```

```
## [1] "x est un nombre négatif"
```

```
# si on change x

x <- 4
if(x < 0){

  print('x est un nombre négatif')

}


# rien ne se passe
```

En gros, dès que la condition à l'interieure de *if()* est `TRUE`, le code qui suis est éxécuté. Si la condition retourne un `FALSE`, le code qui suis n'est pas éxécuté.

3.1.1 ELSE

La condition *else()* par contre ne peux pas être utilisé seule. Elle doit venir après une condition *if()*. Quand la condition dans if() n'est pas satisfaire, le script executera le code qui suis la condition *else()* .

Pour traduire en français:

SI la variable vaut ça, ALORS fais ceci, SINON fais cela.

Par exemple:

```
x <- 4

# si x est négatif
if(x < 0){
# fais ceci

print('x est un nombre négatif')

# sinon
}else{
# fais cela

print('x est un chiffre positif')
}
```

```
## [1] "x est un chiffre positif"
```

3.1.2 ELSE IF

R vous permet aussi de créer multiple conditions avec la fonction *else if()*, qui doit être localisé entre *if()* et *else()*

Par exemple dans notre script, qu'est-ce qui se passe si x = 0 ? Vous pouvez inclure cette condition avec *else if()*

```
x <- 0


if(x < 0){

  print('x est un nombre négatif')

}else if(x == 0 ){

  print('x est égale à 0')

}else{

  print('x est un chiffre positif')

}
```

```
## [1] "x est égale à 0"
```

3.1.3 Conditions if-else emboitées

Enfin, vous pouvez créer des scripts avec multiples conditions *else if* et aussi emboiter d'autres conditions à l'intérieur de la condition principale.

```
# fonction sample pour choisir un chat aléatoirement
x <- sample(nrow(df.chats),1)

if(df.chats$race[x] == "Persan"){

  if(df.chats$sexe[x] == "M"){
    print("C'est un chat Persan mâle")

  }else{
    print("C'est un chat  Persan femelle")
  }
}else if (df.chats$race[x] == 'Main Coon'){

  if(df.chats$sexe[x] == "M"){
    print("C'est un chat Main Coon mâle")

  }else{
    print("C'est un chat Main Coon femelle")
  }
}else if (df.chats$race[x] == 'Siamois'){

  if(df.chats$sexe[x] == "M"){
```

```
    print("C'est un chat Siamois mâle")

  }else{
    print("C'est un chat Siamois femelle")
  }

}else if (df.chats$race[x] == 'Angora'){

  if(df.chats$sexe[x] == "M"){
    print("C'est un chat Angora mâle")

  }else{
    print("C'est un chat Angora femelle")
  }

}else{
  if(df.chats$sexe[x] == "M"){
    print("C'est un chat sans race mâle")

  }else{
    print("C'est un chat sans race femelle")
  }
}
```

3.2 Boucles (for loop)

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.iterateur in loop.objet){

LOOP.CODE

}
```

Le *loop.iterateur* est la valeur qui va changer à chaque itération. Genéralement on le nomme de **i**, mais vous pouvez l'appeler de ce que vous voulez. L'itérateur est généralement utilisé dans le loop pour indexer un objet, et à chaque itération l'indexation changera.

Le *loop.object* est l'objet sur lequelle l'itérateur va itérer. Généralement il prend la forme d'un vecteur construit avec *i:j*, *seq()*.

3.2.1 for loop sur un vecteur

Par example

```
chats.nom

for(chats in chats.nom){

print(chats)

}
```

```
# si on veux avoir accès a l'index du loop pour indexerun autre objet, il faut le faire explicitement

for(i in 1:length(chats.nom)){

print(paste(chats.nom[i],"a besoin de", chats.sommeil[i], 'heures de sommeil pour survivre'))

}

#R fait ceci

for(i in 1:10){

print(chats.nom[i])
}
```

Vous pouvez utiliser les options **break** et **next** pour changer le fonctionnement de vos boucles à l'aide d'une condition IF. Le premier interrompt la boucle et le deuxième interromp l'iteration qui es en cours.

```
# Par exemple, si on voudrais que le for loop s'arrète dès qui rencontre un nom de chat avec 4 caractèr
# On utilise la fonction nchar() pour conter le nombre de caractère dans un string
# Il devrait s'arreter dès qui rencontre le nom "MAYA" qui est avant "PATOUNE"
for(chats in chats.nom){

if(nchar(chats)== 4){
break
}

print(chats)

}
# Ici il ignore le reste du code dans la boucle quand il exectute "next" et il passe à la prochaine ité
# Si notre code marche, il ne doit pas imprimer les noms "MAYA" et "LUNA"
for(chats in chats.nom){

if(nchar(chats)== 4){
next
}

print(chats)

}
```

3.2.2 for loop sur une liste

Si vous utiliser la sintaxe de for loop direct sur les listes, la boucle va considérer chaque compartiment comme un élément à iterer.

```
# version direct sans obtenir l'index
list.chats <- list(chats.nom, chats.poid, chats.age, chats.race, chats.sexe, chats.sommeil)

for (chat in list.chats){

print(chat)

}
```

```
##  [1] "Fluffy"     "Billy"      "Minette"    "Patoune"    "Maya"
##  [6] "Pilou"      "Luna"       "Pacha"      "Minou"      "Gribouille"
##  [1] 3.82 3.55 5.60 4.00 3.66 4.03 3.76 3.52 3.91 6.48
##  [1] 10  4  3  2 11  1  8  9  7  6
##  [1] "Persan"    "Siamois"   "Main Coon" "Goutière"  "Siamois"
##  [6] "Goutière"  "Angora"    "Persan"    "Persan"    "Main Coon"
##  [1] "F" "M" "F" "F" "F" "M" "F" "M" "M" "M"
##  [1] 18 13 12 13 18 14 15 17 17 18
```

```
# version avec obtention de l'index du loop
for(ch in 1:length(list.chats)){

print(list.chats[[ch]])


}
```

```
##  [1] "Fluffy"     "Billy"      "Minette"    "Patoune"    "Maya"
##  [6] "Pilou"      "Luna"       "Pacha"      "Minou"      "Gribouille"
##  [1] 3.82 3.55 5.60 4.00 3.66 4.03 3.76 3.52 3.91 6.48
##  [1] 10  4  3  2 11  1  8  9  7  6
##  [1] "Persan"    "Siamois"   "Main Coon" "Goutière"  "Siamois"
##  [6] "Goutière"  "Angora"    "Persan"    "Persan"    "Main Coon"
##  [1] "F" "M" "F" "F" "F" "M" "F" "M" "M" "M"
##  [1] 18 13 12 13 18 14 15 17 17 18
```

3.2.3 for loop sur une matrice et data.frame

Si vous utiliser la sintaxe de for loop direct sur les matrices et data.frames, la boucle va considérer chaque colonne comme un élément à iterer.

```
# Supposez que les chats dorment le même nombre d'heures par jour tout les jours.
# On va créer un loop pour calculer combiens d'heures ils ont dormi toute leur vie.
# PS: on peux faire cela sans un loop, mais juste pour une démonstration.

# Vu que vous voulez obtenier une donné par chat, vous devez tourner la boucle
# à travers les lignes. Donc vous devez utiliser la sintaxe qui vous fourni un index.


for(i in 1:nrow(mat.chats)){

# calculez le nombre de jours qui a vécu un chat
# Assumez que toutes les années on 365 jours
n_jours <- mat.chats[i,"age"]*365

n_heures_sommeil <- n_jours * mat.chats[i,"sommeil"]


}
n_heures_sommeil
```

```
##   age
## 39420
```

Est-ce qu'il a un problème dans ce loop? Oui, le résultat "n_heures_sommeil" est écrasé à chaque itération et à la fin du loop on a seulement le résultat du dernier chat. Comment résoudre ce problème? Une façon:

```r
# Créez un objet vide
n_heures_sommeil<-NULL
for(i in 1:nrow(mat.chats)){

# calculez le nombre de jours qui a vécu un chat
# Assumez que toutes les années on 365 jours
n_jours <- mat.chats[i,"age"]*365

n_heures_sommeil <- c(n_heures_sommeil, n_jours * mat.chats[i,"sommeil"])
}
n_heures_sommeil
```

```
##   age   age   age   age   age   age   age   age   age   age
## 65700 18980 13140  9490 72270  5110 43800 55845 43435 39420
```

Cette façon n'est pas optimale parce que a chaque itération, n_heures_sommeil change de taille. Quand un objet change de taille il deviens un nouveau objet. Ceci fait que à chaque fois que l'objet change, R demande a l'ordinateur de le stocker dans la mémoire. Ceci ralenti l'execution du code énormement.

Une autre fácon est de créer un objet qui à la taille atendu du résultat final. On appelle cette procédure d'allocation. Comme vous savez combiens de chat il y a, vous savez quel sera la taille du vecteur de résultats. Vous pouvez utiliser le nombre de chats pour définir la longueur de "n_heures_sommeil".

```r
# Cette fois-si il faut utiliser NA et pas NULL parce que NULL n'occupe pas d'espace
n_heures_sommeil<-rep(NA, length = nrow(mat.chats))
for(i in 1:nrow(mat.chats)){

# calculez le nombre de jours qui a vécu un chat
# Assumez que toutes les années on 365 jours
n_jours <- mat.chats[i,"age"]*365

n_heures_sommeil[i] <-  n_jours * mat.chats[i,"sommeil"]
}
n_heures_sommeil
```

```
##  [1] 65700 18980 13140  9490 72270  5110 43800 55845 43435 39420
```

vous pouvez mélanger un for loop avec des conditions if. Imaginez que vous voulez les heures de sommeils seulement des chats siamois et persans

```r
n_heures_sommeil<-rep(NA, length = nrow(mat.chats))
for(i in 1:nrow(mat.chats)){

if(chats.race[i] %in% c('Siamois','Persan')){
# calculez le nombre de jours qui a vécu un chat
# Assumez que toutes les années on 365 jours
n_jours <- mat.chats[i,"age"]*365

n_heures_sommeil[i] <-  n_jours * mat.chats[i,"sommeil"]
}

}
n_heures_sommeil
```

```
##  [1] 65700 18980    NA    NA 72270    NA    NA 55845 43435    NA
```

3.3 Fonctions

Les fonctions sont utilisée partout dans R. Elle fonctionnent comme une "black box" où on rentre un input (entrée de données) dans la fonction, elle performe son calcul sur ce input et ensuite nous retourne un output (un résultat sortant). Par exemple la fonction *sd()* calcule l'écart-type des éléments d'un vecteur.

```
# Quel est l'écart-type de "1, 4, 7 et 9"
sd(c (1, 4, 7, 9))
```

```
## [1] 3.5
```

```
# Vous pouvez affecter ce vecteur à "chiffres"
chiffres <- c (1, 4, 7, 9)
# et ensuite faire
sd(chiffres)
```

```
## [1] 3.5
```

```
# vous pouvez affecter le output de sd() aussi
sd_x <- sd(chiffres)
sd_x
```

```
## [1] 3.5
```

Chaque fonction a ça propre documentation qui est accessible en utilisant une autre fonction, la fonction *help()*. Une autre façon est de mettre un **?** devans la fonction qu'on voudrais acceder la documentation.

```
help(sd)

?sd
```

3.3.1 Arguments dans les fonctions

On observe que la fonction *sd()* viens avec deux argument de "input", x et na.rm = FALSE. Comment ça se fait t'il qu'elle fonctionne quand on inclus seuleument un argument? Ce qui se passe c'est que l'argument "na.rm" viens déjà avec une valeur défault (= FALSE). Cet argument indique si l'utilisateur veux que la fonction *sd()* élimine les valeurs mancantes pour calculer l'écart-type. Donc si l'utilisateur n'inclus pas de valeur pour na.rm, la fonction adopte une valeur FALSE par défault, et la fonction est éxécutée. Par contre si l'utilisateur ne spécifie pas "x", R retourne une erreur.

```
#sd()
```

R reconnais chaque argument par ça position dans la fonction. C'est pour ça qu'on a pas besoin de mettre comme input un x. La fonction va considérer automatiquement n'importe quel objet qu'on met dans la première position dans la fonction comme "x". Donc si vous invertez les leurs de x est de na.rm il y aura une erreur

```
#sd(TRUE,chiffres)
```

Donc la position des arguments est importante.

```
# Incluez des valeurs mancantes dans "chiffres"
chiffres <- c( chiffres, NA, NA)
chiffres
```

```
## [1]  1  4  7  9 NA NA
```

```
# Maintenant, si on ne spécifie pas na.rm, il ne vas pas éliminer les valeurs mancantes parce que le dé
sd(chiffres)
```

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

```r
# Et le résultats est une valeur mancante.

# Changez na.rm pour TRUE
sd(chiffres,TRUE)
```

```
## [1] 3.5
```

```r
# Bingo, vous obtenez l'écart-type
```

Une autre façon de spéficier les arguments dans la fonction est de les nommés explicitement. Si vous faites ceci, vous pouvez utiliser l'ordre que vous désiré pour les arguments.

```r
sd(na.rm=TRUE,x = chiffres)
```

```
## [1] 3.5
```

À retenir

- Les fonction de R: input –> blackbox –> output
- Les arguments des fonctions peuvent être spécifié par **position** ou par **nom**
- Quelques arguments des fonctions R ont déjá une option/valeur défault qui est reconnaisable par le "=".
- D'autres arguments n'ont pas d'option de défault, est la fonction retourne une erreur si ceux-ci ne sont pas spécifié.

3.3.2 Creez votre propre fonction

Quand vous realizez que vous utilisez un script de R trop souvant, c'est le moment de le transformer en fonction. R permet de créer des fonctions qui sont spécifique au problème que vous voulez résoudre. Ça dépend seulement de votre créativité.

La structure d'une fonction sur R est la suivante

```r
my_fun <- function(arg1, arg2, ...){


Corp de la fonction




}
```

Pour commencer, un exemple très simple Vous voulez créer une fonction qui calcule le carré d'un chiffre.

```r
carre <- function(x){
x^2
}
carre(9)
```

```
## [1] 81
```

Si vous ne spécifiez pas, la fonction vous retourne le dernier objet créé. Pour spécifiez le output, vous devrez utiliser la fonction *return()*

```r
carre <- function(x){
y <- x^2
return(y)
}
carre(9)
```

```
## [1] 81
```

Vous pouvez bien sur affécté le résultat à une variable

```
z <- carre(9)
z
```

```
## [1] 81
```

Si vous voulez généraliser la fonction pour n'importe quel exponent, ajoutez un deuxième argument

```
my_exp <- function(x, exp){
y <- x^exp
return(y)
}
my_exp(3,3)
```

```
## [1] 27
```

Si vous voulez que l'argument *exp* soit optionnel, vous pouvez lui donner une valeur défault.

```
my_exp <- function(x, exp = 2){
y <- x^exp
return(y)
}
my_exp(3)
```

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

Une autre example. Une fonction que divisie le premier argument par le deuxième, et ensuite somme avec la multiplication de ces deux arguments.

```
my_fun_math <- function(x,y){

x/y + x*y


}
my_fun_math(4,2)
```

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

Maintenant avec le deuxième argument en default = 1

```
my_fun_math <- function(x,y = 1){

x/y + x*y


}
my_fun_math(4)
```

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

Vous vous demandez peut-etre, à quoi sert la fonction *return()* si notre fonction retourne l'objet désiré sans l'utiliser? Si la fonction *return()* est utilisé au millieu de votre fonction, elle va retourner le output désiré et le restant du corps de la fonction est ignoré. Ceci est utiles avec des conditions IF. Voici un example:

En R, si vous divisez un nombre par 0, il retorune *Inf* (infini)

```
4/0
```

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

Vous voulez éviter que le résultats de votre fonction retoure infini.

```
my_fun_math <- function(x,y = 1){

if(x | y == 0){
return(0)
}

x/y + x*y
}
my_fun_math(4,0)
```

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

```
my_fun_math(0)
```

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

```
my_fun_math(0,4)
```

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

Ici vous voyez que la fonction *return()* retourne le output 0 et le reste du code est ignoré. Dernier points:

Les variables définis à l'intérieur d'une fonction ne sont pas accessible à l'extérieur de la fonction (i.e., dans l'environnement global).

À moins que vous le demandez explicitement, le input d'une fonction ne changera pas à l'éxtérieur de la fonction. Voici un example

```
increment <-function(x, inc = 1){
x + inc
x
}
count <- 5
increment(count,5)
```

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

D'après vous, la variable **count** est 5 ou 10?

3.3.3 Packages

Les packages sont des bibliothèques de fonctions programmées par les utilisateurs de R et mises à disposition de la communauté des utilisateurs de R. Il y a des centaines, voir des milliers de packages disponible sur le net.

3.3.3.1 Instalation de package

```
install.packages("lubridate")
```

3.3.3.2 Activation du package

```
library("lubridate")
```

```
## Warning: package 'lubridate' was built under R version 3.5.2
```

```
##
## Attaching package: 'lubridate'
```

```
## The following object is masked from 'package:base':
##
##     date
```

```
# ou
require("lubridate")
```

3.3.3.3 Affichez toutes les fonctions d'un packages

```
ls("package:lubridate")

# Vous pouvez auassi verifier la documentation d'un package avec ?package
```

Le package que vous avez importé et activé sers à aider la manipulation des variables de dates

```
# Par example, retournons à notre vecteur "dates" du début
dates
```

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

```
# Pour extraire le mois de chaque date il y à la fonction month()
month(dates)
```

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

```
# Pour extraire l'annnée de chaque date il y à la fonction year()
year(dates)
```

```
## [1] 1992 1992 1992 1992 1992
```

```
# Pour extraire le jour de chaque date il y à la fonction day()
day(dates)
```

```
## [1] 27 27 14 28  1
```

Si vous créez une fonction qui utilise des fonctions d'un package, il ne faut pas oublier de faire appel au package dans le corp de votre fonction avec *require()* !! Sinon, votre fonction ne marchera pas.

3.3.4 Quelques ressources sur R

Pour trouvez la liste de package dans le serveur CRAN, allez vers se LIEN.

Vous pouvez aussi visiter le site de R-Bloggers

Si vous avez des questions par rapport à R, allez au site stackoverflow

3.4 Graphiques

Le logiciel R est très performant quand le sujet est la visualisation des données. R permet de manipuler chaque composante d'un graphique et donc possibilite une personnalisation de vos graphiques à vos besoins particuliers.De nombreux packages on été dévéloppé pour faciliter la production de graphique sur R. Le plus connu de tous est ggplot2, mais il y en as des dizaines d'autres (lattice‘, grid, etc). Par contre, pour cette formation on va utiliser le package graphics, qui est la librarie de base de R pour produire des graphiques. Pour avoir un aperçu des possibilités de ce package

```
demo(graphics)
```

Pour avoir la liste de fonctions de *graphics*

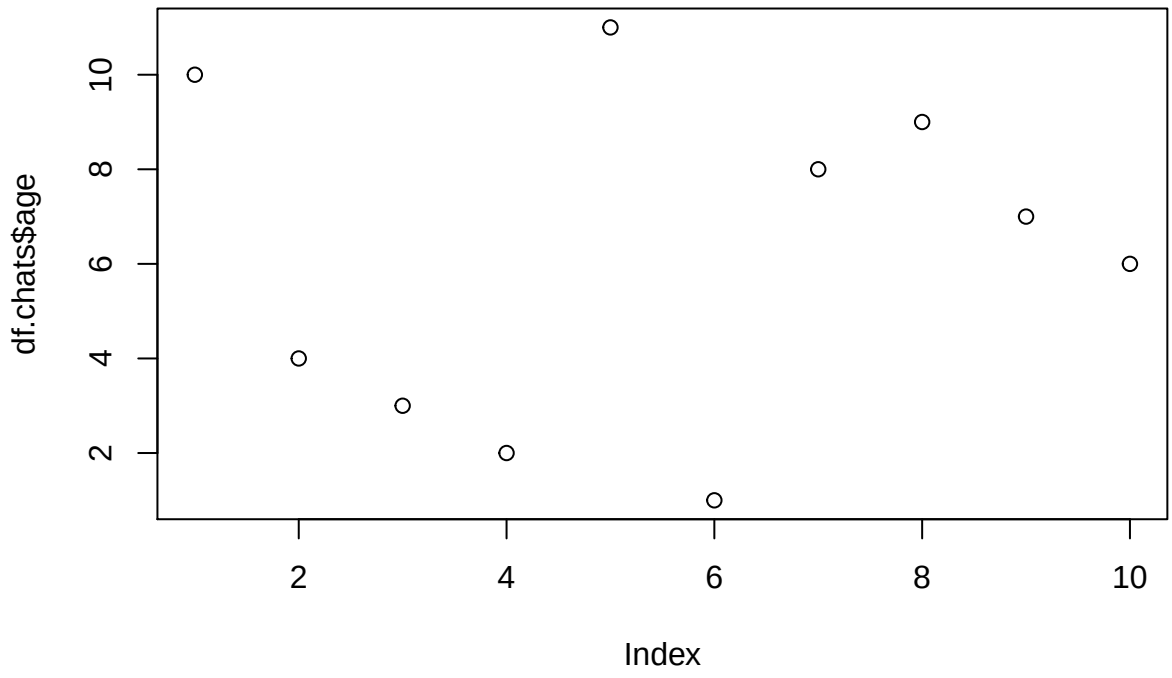
```
ls("package:graphics")
```

3.4.1 Diagramme de dispersion (Scatterplot)

Vous pouvez créez un diagrame de dispersion avec la fonction *plot()*

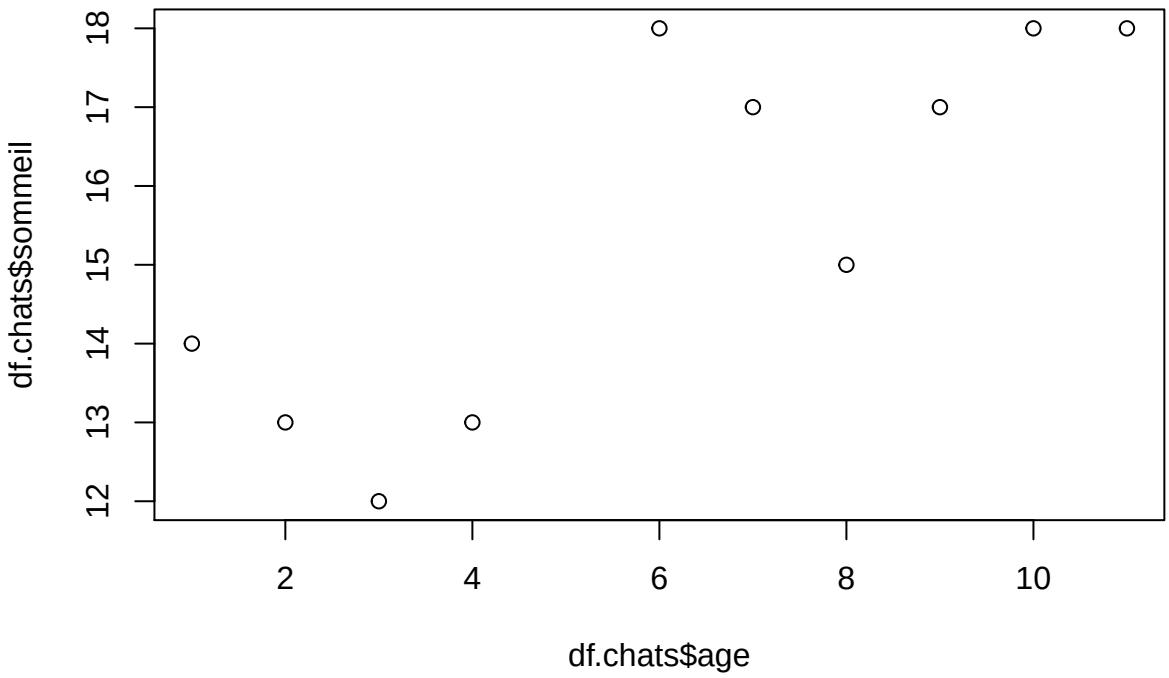
```
# Si votre "input" consiste a seulement un vecteur, la fonction matchera votre vecteur avec une séquenc

# Par example, si on veut voir un diagramme de dispersion de l'âge des chats
plot(df.chats$age)
```




```
# Comme il y a 10 chats, la fonction vous créé automatiquement un axe de 1 à 10


# Si vous voulez visualiser le nombre d'heures de sommeil des chats versus leurs âge
plot(df.chats$age, df.chats$sommeil)
```


df.chats$sommeil
12
13
14
15
16
17
18
2
4
6
8
10
df.chats$age