Formation R

R. Drouilhet

LJK FIGAL

Plan

Installation et présentation R (Studio)

1. Si ce n'est pas déjà fait : télécharger RStudio à `www.rstudio.com`.
2. Présentation des 4 sous-fenêtres.
3. Prise en main rapide : taper les instructions ci-dessous dans la partie console.

```
> runif(5)      # 5 réels au hasard dans [0;1]
[1] 0.6712563 0.3910138 0.3548334 0.4303511
[5] 0.3617345
> a <- runif(5) # même chose dans la variable a
> a             # afficher l'objet a
[1] 0.3508840 0.7037245 0.5300490 0.3686393
[5] 0.7616423
> ls()
[1] "a"
> rm(a)
> ls()
character(0)
```

Historique instructions

4. Jouer avec les touches **[flêche-haut]**, **[flêche-bas]** pour parcourrir l'historique des instructions précédemment saisies. Alternativement, sélectionner dans la sous-fenêtre "historique". Obtenir la première instruction.

```
1  > # après recherche dans l'historique
2  > a <- runif(5) # valider
```

5. Phase de développement typique : créer, affecter un objet.

```
1  > runif(3)*5 # créer votre objet R
2  [1] 4.629078 3.509460 1.863706
3  > # ravi! enregistrez-le dans a en tapant :
4  > # [flêche haut],[-],[>],[a] et [Entrée]
5  > runif(3)*5->a
6  > # puis tapez [a] puis [Entrée]
7  > a
8  [1] 4.756076 3.580609 4.127598
```

Persistance d'une session et documentation

6. Persistance d'une session :
   - vérifier votre espace de travail :

```
> ls()
character(0)
```

   - quitter le logiciel R ou RStudio : Il vous demande de sauvegarder votre session. Cliquer sur le bouton **[Save]**.
   - Reouvrir le logiciel R ou RStudio.
   - vérifier que votre espace de travail est le même :

```
> ls()
character(0)
```

   - Vérifier aussi votre historique des instructions.
7. Documentation : Sous-fenêtre RStudio ou dans la console : **help(<instr>)** ou **?<instr>**, **??<instr>** et **apropos(<instr>)** **demo()**, **example(<instr>)**

Plan

Nature des données

- Chaîne de caractères (**character**)

```
> ch <- "Voici une chaîne" # ou 'Voici une chaîne'
```

- Numérique (**numeric**)

```
> nb <- 101   # un réel, i.e. équivalent à 101.0
> nb
[1] 101
> is.integer(nb)
[1] FALSE
> i <- 101L   # un entier!
> i
[1] 101
> is.integer(i)
[1] TRUE
```

Nature des données (suite)

- Booléen (**logical**)

```
> c(TRUE,FALSE,T,F)
[1]  TRUE FALSE  TRUE FALSE
> nb==i
[1] TRUE
> identical(nb,i)
[1] FALSE
> as.integer(nb)==i
[1] TRUE
> identical(as.integer(nb),i)
[1] TRUE
```

Nature des données (suite)

Quelques autres natures d'objets R :

```
> # Non Available
> NA
[1] NA
> # Not a Number
> c(NaN,log(-1),0/0)
[1] NaN NaN NaN
> c(Inf,1/0,-1/0)
[1]  Inf  Inf -Inf
> a <- 1+2i
> a
[1] 1+2i
> is.complex(a)
[1] TRUE
```

Structures de données de base

- Vecteur (**vector**) : suite ordonnée de données de même nature.

```
> v <- c(1,3,2)
> v
[1] 1 3 2
> 1:10          # équivalent à seq(1,10)
 [1]  1  2  3  4  5  6  7  8  9 10
> rep(2,3)
[1] 2 2 2
```

En R , un élément atomique (la plus petite structure possible) est représenté par un vecteur de longueur 1.

- Liste (**list**) : suite ordonnée d'objets R de tout type.

```
> # liste non nommée
> l <- list(1:10,c("toto","titi"))
> # liste nommée
> l2 <- list(a=1:10,b=c("toto","titi"))
```

Attributs

Un attribut est une métadonnée attachée à tout objet R utilisée la plupart du temps par les développeurs (transparent pour utilisateur). Les noms dans les listes nommées sont gérés grâce à la notion d'attributs.

```
> l2
$a
 [1]  1  2  3  4  5  6  7  8  9 10

$b
[1] "toto" "titi"

> attributes(l2)      # tous les attributs
$names
[1] "a" "b"

> attr(l2,"names")    # un seul attribut
[1] "a" "b"
```

Attributs (suite)

Fort heureusement, il existe toujours la version utilisateur

```
> names(l2)   # attribut "names"
[1] "a" "b"
```

Le point fort du R est la modification des attributs sans changement de la structure principale.

```
> # attribut modifié (mode développeur)
> attr(l2,"names") <- c("b","a")
> # équivalent à (mode utilisateur)
> names(l2) <- c("b","a")
> l2
$b
 [1]  1  2  3  4  5  6  7  8  9 10

$a
[1] "toto" "titi"
```

*Matrice (**matrix**) et tableau (**array**)*

Une matrice ou un tableau est un vecteur (**vector**) avec un attribut dimension (**dim**), vecteur spécifiant les dimensions de la matrice ou tableau. La dimension d'une matrice est égale à 2 correspondant aux nombres de lignes et colonnes.

```
> mat <- matrix(1:8,nrow=2)
> mat
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
> is.vector(mat)
[1] FALSE
> attributes(mat)
$dim
[1] 2 4
```

Matrice et tableau (Suite)

En changeant uniquement l'attribut **dim** de la matrice (en fait, le vecteur) **mat**, il est possible de le modifier.

```
> # change le nombre de lignes
> attributes(mat)$dim <- c(4,2)   # développeur
> dim(mat) <- c(4,2)              # utilisateur
> mat
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

Matrice et tableau (Suite)

```
> dim(mat) <- c(2,2,2) # matrice => tableau
> mat
, , 1

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

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8

> is.matrix(mat)
[1] FALSE
```

Matrice et tableau (Suite)

```
> is.array(mat)
[1] TRUE
> is.vector(mat)
[1] FALSE
> dim(mat) <- NULL     # tableau => vecteur
> # équivalent plus usuel
> # mat <- as.vector(mat)
> mat
[1] 1 2 3 4 5 6 7 8
> is.vector(mat)
[1] TRUE
```

*Facteur (**factor**) ou Variable qualitative*

Une variable qualitative (ou nominale), aussi appelée facteur dans le contexte de l'analyse de la variance, peut être vu comme un vecteur de chaînes de caractères (**character**). Afin d'économiser l'espace mémoire, le R dispose de la classe **factor** permettant de recoder le facteur via un vecteur d'indices où chaque valeur d'indice correspond à un unique niveau du facteur. L'attribut (**levels**) correspond alors au vecteur des niveaux du facteur.

```
> ch <- c(rep("constitution",2),rep("democracy",3))
> ch
[1] "constitution" "constitution" "democracy"
[4] "democracy"    "democracy"
> fa <- factor(ch)
> fa
[1] constitution constitution democracy
[4] democracy    democracy
Levels: constitution democracy
```

Facteur (Suite)

```
> c(class(ch),class(fa))
[1] "character" "factor"
> attributes(fa)
$levels
[1] "constitution" "democracy"

$class
[1] "factor"

> as.integer(fa)
[1] 1 1 2 2 2
> levels(fa)
[1] "constitution" "democracy"
```

Le R saît notamment comment traiter un facteur en tant que variable explicative dans un modèle linéaire.

*Matrice de données (**data.frame**)*

Une matrice de données est une liste (**list**) de vecteurs (**vector**) de même taille représentant les variables en colonnes de la matrice de données. Trois attributs sont utilisées pour différencier une liste d'une matrice de données :

1. *class* : vecteur spécifiant la (ou les) classe(s) de l'objet (voir programmation objet)
2. *names* : vecteur des noms de variables (colonnes)
3. *row.names* : vecteurs des noms des individus (lignes)

```
> df <- data.frame(
+         a=1,
+         b=rep(c(T,F),2),
+         c=paste("niveau",1:4,sep=""),
+         row.names=0:3
+     )
> is.list(df)
[1] TRUE
```

Matrice de données (Suite)

```
> df
  a     b       c
0 1  TRUE niveau1
1 1 FALSE niveau2
2 1  TRUE niveau3
3 1 FALSE niveau4
> attributes(df)
$names
[1] "a" "b" "c"

$row.names
[1] 0 1 2 3

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

Matrice de données (Suite)

```
> df
  a     b       c
0 1  TRUE niveau1
1 1 FALSE niveau2
2 1  TRUE niveau3
3 1 FALSE niveau4
> # mode utilisateur
> names(df)
[1] "a" "b" "c"
> rownames(df)
[1] "0" "1" "2" "3"
> class(df)
[1] "data.frame"
```

Plan

Identifier le type d'un objet R

Pour déterminer le type (interne) d'un objet, appliquer **typeof()**. Pour déterminer sa classe, appliquer **class()**. Les fonctions de la forme **is.<type>()** permettent de déterminer si l'objet est du type **<type>**.

```
> a <- 1L
> c(typeof(a),class(a),is.integer(a),is.numeric(a))
[1] "integer" "integer" "TRUE"    "TRUE"
> a <- 1
> c(typeof(a),class(a),is.double(a),is.numeric(a))
[1] "double"  "numeric" "TRUE"    "TRUE"
> a <- "1"
> c(typeof(a),class(a),is.character(a))
[1] "character" "character" "TRUE"
> c(is.vector(a),is.list(a))
[1]  TRUE FALSE
> a <- list(1,1L)
> c(is.vector(a),is.list(a))
[1] TRUE TRUE
```

Conversion d'un objet R

Les fonctions de la forme **as.<type>()** permettent de convertir un objet dans le type <**type**>.

```
> a <- as.integer(1)
> typeof(a)
[1] "integer"
> a <- as.double(1L)
> typeof(a)
[1] "double"
> as.list(a)
[[1]]
[1] 1

> a <- as.factor(c("to","ti","to"))
> a
[1] to ti to
Levels: ti to
```

Autoconversion pour un vecteur

Puisqu'un vecteur est une structure d'éléments du même type que se passe t'il si on met ensemble des éléments de natures différentes ?

```
> a <- c(TRUE,2L)
> a
[1] 1 2
> class(a)
[1] "integer"
> a <- c(a,3)
> a
[1] 1 2 3
> class(a)
[1] "numeric"
> a <- c(a,"quatre")
> a
[1] "1"      "2"      "3"      "quatre"
> class(a)
[1] "character"
```

Instructions courantes pour créer des vecteurs

```
> # "empty" constructors
> logical(2)
[1] FALSE FALSE
> integer(1)
[1] 0
> numeric()
numeric(0)
> character(3)
[1] "" "" ""
> # sequence
> 1:5 # équivalent à c(1,2,3,4,5)
[1] 1 2 3 4 5
> -5:5
 [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
> 5:(-5)
 [1]  5  4  3  2  1  0 -1 -2 -3 -4 -5
```

Instructions courantes pour créer des vecteurs (Suite)

```
> # idem avec variables
> i1 <- -5
> i2 <- 5
> i1:i2
 [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
> i2:i1
 [1]  5  4  3  2  1  0 -1 -2 -3 -4 -5
> seq(1,10)   # équivalent à 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> class(1:10)
[1] "integer"
> # plus fin
> seq(1,2,by=0.1)
 [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
> seq(1,2,length=11)
 [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

Instructions courantes pour créer des vecteurs (Suite)

```
> # déjà vu mais pas encore ça
> c()
NULL
> c(NULL,1)
[1] 1
> a <- NULL
> a <- c(a,1)
> a
[1] 1
> a <- c(a,c(a,3),2)
> a
[1] 1 1 3 2
```

Instructions courantes pour créer des vecteurs (Suite)

```
> # repéter
> rep(1,3) # équivalent à c(1,1,1)
[1] 1 1 1
> rep(1:2,3)
[1] 1 2 1 2 1 2
> rep(1:2,2:3)
[1] 1 1 2 2 2
> rep(c("toto","tutu"),2:1)
[1] "toto" "toto" "tutu"
> # coller
> paste(c("toto","tutu"),1:2)
[1] "toto 1" "tutu 2"
> paste(c("toto","tutu"),1:2,sep="")
[1] "toto1" "tutu2"
> paste(c("toto","tutu"),1:2,collapse=" & ")
[1] "toto 1 & tutu 2"
```

Instructions courantes pour créer des vecteurs (Suite)

unlist() détruit la structure de type liste pour la transformer en vecteur.

```
> unlist(l)
 [1] "1"    "2"    "3"    "4"    "5"    "6"
 [7] "7"    "8"    "9"    "10"   "toto" "titi"
> unlist(l2)
    b1     b2     b3     b4     b5     b6     b7
   "1"    "2"    "3"    "4"    "5"    "6"    "7"
    b8     b9    b10     a1     a2
   "8"    "9"   "10" "toto" "titi"
> unlist(df) # plus difficile à comprendre!
a1 a2 a3 a4 b1 b2 b3 b4 c1 c2 c3 c4
 1  1  1  1  1  0  1  0  1  2  3  4
```

Instructions courantes pour créer des listes

L'instruction **c()** n'est pas uniquement applicable aux vecteurs mais aussi aux listes.

```
> c(l,l2)
[[1]]
 [1]  1  2  3  4  5  6  7  8  9 10

[[2]]
[1] "toto" "titi"

$b
 [1]  1  2  3  4  5  6  7  8  9 10

$a
[1] "toto" "titi"
```

Instructions courantes pour créer des listes (Suite)

Si au moins un des arguments de **c()** est une liste le résultat est une liste.

```
> c(l,v) # équivalent à c(l,as.list(v))
[[1]]
 [1]  1  2  3  4  5  6  7  8  9 10

[[2]]
[1] "toto" "titi"

[[3]]
[1] 1

[[4]]
[1] 3

[[5]]
[1] 2
```

Instructions courantes pour créer des listes (Suite)

```
> c(v,l) # équivalent à c(as.list(v),l)
[[1]]
[1] 1

[[2]]
[1] 3

[[3]]
[1] 2

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

[[5]]
[1] "toto" "titi"
```

*Instructions **cbind()** et **rbind()** pour matrices*

cbind() et **rbind()** sont les extensions de **c()** pour les matrices.

```
> (rbind(c(1,3,2),3:1) -> mat2)
     [,1] [,2] [,3]
[1,]    1    3    2
[2,]    3    2    1
> (cbind(mat2,5) -> mat2)
     [,1] [,2] [,3] [,4]
[1,]    1    3    2    5
[2,]    3    2    1    5
> rbind(mat2,1:2)
     [,1] [,2] [,3] [,4]
[1,]    1    3    2    5
[2,]    3    2    1    5
[3,]    1    2    1    2
```

Notons que pour les deux dernières instructions R a opéré une règle de "recycling".

Instructions ***cbind()*** *et* ***rbind()*** *pour matrices de données*

```
> (data.frame(a=c(1,3),b=3:2,c=2:1) -> df2)
  a b c
1 1 3 2
2 3 2 1
> (cbind(df2,d="2",e=c("to",'ti')) -> df2)
  a b c d  e
1 1 3 2 2 to
2 3 2 1 2 ti
> rbind(df2,1:2)
  a b c d    e
1 1 3 2 2   to
2 3 2 1 2   ti
3 1 2 1 2 <NA>
```

Dans la dernière instruction, le "recycling" convertit le dernier élément de la dernière variable en **NA** car il la traite comme un facteur.

Extraction de vecteur

- par indexation

```
> v
[1] 1 3 2
> v[c(1,3)]          #sous-vecteur par indexation
[1] 1 2
```

- par condition

```
> # même extraction par condition logique
> v[v %in% c(1,2)]  #sous-vecteur par condition
[1] 1 2
> # décomposé étape par étape
> v %in% c(1,2)  #vecteur logical même longueur que v
[1]  TRUE FALSE  TRUE
> v[c(TRUE,FALSE,TRUE)] # équivalent à v[c(1,3)]
[1] 1 2
```

Extraction de matrice et tableau

Les opérations d'extraction pour les matrices et tableaux sont similaires aux vecteurs excepté que le nombre d'indices doit respecter la dimension de la structure. En particulier, ne pas spécifier d'indice signifie "tous les indices".

```
> mat <- matrix(1:8,nr=2)
> mat[1,]          # 1ère ligne
[1] 1 3 5 7
> mat[,2]          # 2ème colonne
[1] 3 4
> mat[,]           # équivalent à mat
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
> # sous-matrice avec élément 1ère colonne >= 2
> mat[mat[,1]>=2,]
[1] 2 4 6 8
```

Extraction d'élément(s) de liste

L'opérateur d'extraction d'élément en R est généralement `[[...]]`.

```
> l2[[1]]   # élément de l2 par index
 [1]  1  2  3  4  5  6  7  8  9 10
> l2$a      # élément "a" de l2 (mode utilisateur)
[1] "toto" "titi"
> l2[["a"]] # même chose (mode développeur)
[1] "toto" "titi"
> clé<-"a"  # si clé variable contenant "a" (ou 1)
> l2[[clé]] # permet l'extraction par variable
[1] "toto" "titi"
> ## ATTENTION!
> l2$clé    # car clé pas élément de l2
NULL
> names(l2) # comme il est montré ici
[1] "b" "a"
```

Extraction de sous-liste

L'opérateur d'extraction de sous-structure (ici sous-liste) en R est généralement `[...]`.

```
> l2
$b
 [1]  1  2  3  4  5  6  7  8  9 10

$a
[1] "toto" "titi"

> # Ce n'est pas le 1er élément de l2!
> l2[1]
$b
 [1]  1  2  3  4  5  6  7  8  9 10

> # mais la sous-liste contenant le 1er élément!
> is.list(l2[1])
[1] TRUE
```

Extraction de sous-liste (Suite)

```
> # sous-liste avec permutation des éléments
> l2[c(2,1)]
$a
[1] "toto" "titi"

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

Extraction de matrice de données

Une matrice de données (**data.frame**) peut à la fois être manipulée comme une matrice via une double indexation et une liste via une simple indexation (correspondant aux variables).

```
> df[3,]      # 3ème ligne vue comme une matrice
  a    b       c
2 1 TRUE niveau3
> df[,3]      # 3ème colonne vue comme une matrice
[1] niveau1 niveau2 niveau3 niveau4
Levels: niveau1 niveau2 niveau3 niveau4
> df[[3]]     # 3ème colonne mais vue comme une liste
[1] niveau1 niveau2 niveau3 niveau4
Levels: niveau1 niveau2 niveau3 niveau4
```

Extraction de matrice de données (Suite)

```
> # sous-matrice de données (comme une sous-liste)
> df[3]
        c
0 niveau1
1 niveau2
2 niveau3
3 niveau4
> is.data.frame(df[3])
[1] TRUE
> # Ne pas confondre df[3][1] équivalent à df[3]
> # et df[3][[1]] équivalent à df[[3]]
> df[3][[1]]
[1] niveau1 niveau2 niveau3 niveau4
Levels: niveau1 niveau2 niveau3 niveau4
```

Complément sur `[...]` et `[[...]]`

Il est bien de connaître la différence entre l'extraction par simple et double crochets :

- l'extraction par *simple crochet* retourne une sous-structure du même type que l'objet extrait
- quand l'extraction par *double crochet* retourne l'élément précisé par l'indexation
- **Question** : pour un vecteur `v`, quelle est la différence entre `v[1]` et `v[[1]]` ?
- **Réponse** : il n'y en a pas !
  La raison a été donnée précédemment : **l'entité la plus petite (i.e. atomique) en R est le type vector de longueur 1**. Ainsi, l'élément d'une structure est au pire un (sous-)vecteur de longueur 1.

Complément sur `[...]` et `[[...]]`

Il est bien de connaître la différence entre l'extraction par simple et double crochets :

- l'extraction par *simple crochet* retourne une sous-structure du même type que l'objet extrait
- quand l'extraction par *double crochet* retourne l'élément précisé par l'indexation
- **Question** : pour un vecteur `v`, quelle est la différence entre `v[1]` et `v[[1]]` ?
- **Réponse** : il n'y en a pas !
  La raison a été donnée précédemment : **l'entité la plus petite (i.e. atomique) en R est le type vector de longueur 1**. Ainsi, l'élément d'une structure est au pire un (sous-)vecteur de longueur 1.

Complément sur [...] et [[...]] (Suite)

L'extraction d'un sous-vecteur de longueur 1 par simple crochet du vecteur est donc équivalente à l'extraction par double crochet.
Elle est donc plus naturellement utilisée (car plus simple à saisir au clavier) pour le type **vector** et ses types dérivées **matrix** et **array**.

```
> v <- c(3,1,2)
> v[1]
[1] 3
> v[[1]]
[1] 3
> mat[1,2]
[1] 3
> # équivalent à
> # (et non précisé avant par souci de simplicité)
> mat[[1,2]]
[1] 3
```

Modification éléments d'une structure

De la même manière que l'on peut extraire des éléments ou sous-structures, il est possible de les modifier.

```
> v[2] <- 10 # ou v[[2]] <- 10
> v
[1]  3 10  2
> v[1:2] <- c(30,1)
> v
[1] 30  1  2
> v[v %% 2 == 0] <-  v[v %% 2 == 0] / 2
> v
[1] 15  1  1
> mat[1,2] <- 30 # ou mat[[1,2]] <- 30
> mat
     [,1] [,2] [,3] [,4]
[1,]    1   30    5    7
[2,]    2    4    6    8
```

Plan

*Environnement (**environment**)*

Un environnement est une partie allouée de l'espace mémoire rassemblant tout type d'objets R . L'espace de travail **.GlobalEnv** gardant en mémoire tous les objets courants dans une session R est un objet **environment**.

```
1   > e <- new.env()
2   > l2$a              # élément "a" de la liste "l2"
3   [1] "toto" "titi"
4   > e$a <- "toto"   # comme une liste nommée
5   > # un exemple de fonction
6   > change <- function(obj) obj$a <- "titi"
7   > change(l2)
8   > l2$a              # liste est statique
9   [1] "toto" "titi"
10  > change(e)
11  > e$a               # environnement est dynamique
12  [1] "titi"
13  > ls(e)             # liste des objets
14  [1] "a"
```

Environnement (Suite)

Grâce à la fonction **local()** il est possible d'exécuter un bloc d'instructions R dans un environnement isolé du reste du système où ses éléments sont considérés comme les variables.

```
> e$b <- rnorm(5) # nouvelle variable 'b'
> local({
+    a <- paste(a,"et toto")
+    a
+    mean(b)     # dernière instruction retournée!
+ },e)
[1] 0.2938503
> local({
+    print(a)    # print pour afficher "a"
+    mean(b)     # dernière instruction retournée!
+ },e) -> moy
[1] "titi et toto"
> moy
[1] 0.2938503
```

Environnement (Suite)

Par défaut, un nouvel environnement a pour parent l'environnement **.GlobalEnv** (et de ses parents) lui permettant ainsi de reconnaître toutes les instructions du système R et toutes les variables de l'espace de travail.

```
> ls(e)
[1] "a" "b"
> ls()
 [1] "ch"     "change" "clé"    "df"     "e"
 [6] "fa"     "i"      "l"      "l2"     "mat"
[11] "moy"    "nb"     "v"
> # e a pour parent .GlobalEnv
> parent.env(e)
<environment: R_GlobalEnv>
> # moy est ainsi reconnu dans un bloc local dans e
> local(mean(b - moy),e)
[1] 0
```

Gestion de l'espace de travail

```
> a
Erreur : objet 'a' introuvable

> names(l2)
[1] "b" "a"
> l2$a              #  "a" dans l2
[1] "toto" "titi"
> search()
[1] ".GlobalEnv"        "package:stats"
[3] "package:graphics"  "package:grDevices"
[5] "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"
[9] "package:base"
> attach(l2)      # accès direct éléments de l2
> a               # "a" comme l2$a
[1] "toto" "titi"
```

Gestion de l'espace de travail (Suite)

```
> a                  # "a" comme l2$a
[1] "toto" "titi"
> search()
 [1] ".GlobalEnv"        "l2"
 [3] "package:stats"     "package:graphics"
 [5] "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"
 [9] "Autoloads"         "package:base"
> ls(2)              # objets de l2 en 2ème position
[1] "a" "b"
> detach("l2")   # l2 détaché
> a
Erreur : objet 'a' introuvable
```

Sérialisation en R

```
> result <- rnorm(5) # create an object
> result
[1]  1.0333873 -2.9054564 -1.9555856  0.5336679
[5] -0.1467854
> save(result,file="result.RData") # save it in a file
> rm(result)
> result
Erreur : objet 'result' introuvable

> attach("result.RData")
> search()
 [1] ".GlobalEnv"        "file:result.RData"
 [3] "package:stats"     "package:graphics"
 [5] "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"
 [9] "Autoloads"         "package:base"
```

Sérialisation en R (Suite)

```
> result
[1]  1.0333873 -2.9054564 -1.9555856  0.5336679
[5] -0.1467854
> ls()
 [1] "ch"     "change" "clé"    "df"     "e"
 [6] "fa"     "i"      "l"      "l2"     "mat"
[11] "moy"    "nb"     "v"
> detach()              # plus attaché
> result
Erreur : objet 'result' introuvable

> load("result.RData") # result dans workspace
> result
[1]  1.0333873 -2.9054564 -1.9555856  0.5336679
[5] -0.1467854
```

Sérialisation en R (Suite)

Lorsque R vous demande en quittant la session de sauvergarder votre espace de travail, il applique exactement le procédé de sérialisation décrit précédemment. Il existe aussi la fonction **save.image()** qui permet d'enregistrer tout l'espace de travail dans une fichier image.

```
1  > save.image("MonEspaceDeTravail.RData")
2  > # sauvegarde de l'espace de travail
3  > save.image("~ /.RData")  # ou save.image()
```

Environnement (Niveau +)

On peut spécifier à la création d'un environnement son parent. Dans l'exemple ci-dessous, cela permet de créer un environnement isolé de l'espace de travail.

```
> # environnement isolé de .GlobalEnv
> e <- new.env(parent=baseenv())
> e$b <- rnorm(4)
> moy <- mean(e$b)
> local(mean(b - moy),e)
Erreur dans mean(b - moy) : objet 'moy' introuvable
```

On peut surtout créer un nouvel environnement en le faisant hériter d'un environnement parent de sorte que le bloc d'instructions peut aussi accéder aux objets de l'environnement parent (et de ses parents, et de ses ...).

```
> e <- new.env()
> e$b <- rnorm(4)
> moy <- mean(e$b)
> local(mean(b - moy),e)
[1] -5.551115e-17
> # environnement héritant de e
> e2 <- new.env(parent=e)
> # environnement de chaque "b"?
> local({
+    a <- b
+    b <- a+2
+ },e2)
> e$b
[1] -0.9571411  0.2782109 -0.5260109 -1.0751503
> e2$b    # pas le même "b" que e
[1] 1.0428589 2.2782109 1.4739891 0.9248497
> e2$a    # copie de "b" dans e
[1] -0.9571411  0.2782109 -0.5260109 -1.0751503
```

Plan

Séquence d'instructions

En mode utilisateur interactif, notez la différence entre les appels d'une même instruction avec ou sans parenthèses. Le symbole ";" permet de placer plusieurs instructions sur une même ligne.

```
> # SANS valeur de retour
> v <- c(1,3,2)
> # AVEC valeur de retour
> (v <- c(1,3,2))
[1] 1 3 2
> # instructions sur une même ligne
> (v <- c(1,3,2));v*2
[1] 2 6 4
> # équivalent à
> (v <- c(1,3,2))
[1] 1 3 2
> v*2
[1] 2 6 4
```

Bloc d'instructions

Notez alors la différence avec la notion de bloc d'instructions qui est un regroupement d'instructions sur une ou plusieurs lignes placées entre parenthèses. Un bloc d'instructions est alors considéré comme une seule instruction et ayant pour valeur de retour celle retournée par la dernière instruction.

```
> # bloc d'instructions vu comme une instruction
> {(v <- c(1,3,2));v*2}
[1] 2 6 4
> # équivalent à
> {
+    (v <- c(1,3,2))
+    v*2
+ }
[1] 2 6 4
```

Cette fonctionnalité permet de définir les éléments de programmation uniquement en fonction d'instructions simples.

Instruction ***if***

```
> v
[1] 1 3 2
> # sans "else"
> if(length(v)>1) cat("v multiple\n")
v multiple
> if(length(v)==1) cat("v simple\n")
> # avec "else" sans affectation
> if(length(v)>1) cat("plus\n") else cat("un\n")
plus
> # avec affectation
> v2 <- if(length(v)>2) v[1:2] else v
> v2 # vecteur tronqué aux 2 premiers éléments
[1] 1 3
```

*Instruction **switch** en fonction de valeur entière*

```
> case <- 2
> switch(case,"un","deux","trois")
[1] "deux"
> res <- switch(case,"un","deux","trois")
> res
[1] "deux"
> switch(4,"un","deux","trois")
> res <- switch(4,"un","deux","trois")
> # pas d'affichage car
> res
NULL
> # blocs
> switch(case,case+10,{a<-10;case*a},100)
[1] 20
```

*Instruction **switch** en fonction de chaîne de caractères*

```
> case <- "toto"
> switch(case,toto="ToTo",titi="tItI","Autre")
[1] "ToTo"
> switch("rien",toto="ToTo",titi="tItI","Autre")
[1] "Autre"
> switch(case,toto=,TOTO="ToTo",titi="tItI","Autre")
[1] "ToTo"
> switch("to",toto=,to="ToTo",titi="tItI","Autre")
[1] "ToTo"
> # et avec un entier, ça marche?
> switch(3,toto=,to="ToTo",titi="tItI","Autre")
[1] "tItI"
> switch(4,toto=,to="ToTo",titi="tItI","Autre")
[1] "Autre"
> switch(10,toto=,to="ToTo",titi="tItI","Autre")
```

Instructions de Boucle ***for*** *et* ***while***

```
> # for loop
> {
+    cat("v contient: ")
+    for(elt in v) cat(elt," ")
+    cat("\n")
+ }
v contient: 1  3  2
> # while loop
> {
+    i<-0;cat("v contient: ")
+    while( (i<-i+1) <=length(v)) cat(v[i]," ")
+    cat("\n")
+ }
v contient: 1  3  2
```

Instructions ***lapply*** *et* ***sapply***

lapply() et **sapply()** ont pour premier argument un vecteur ou une liste et applique à chacun des éléments la fonction donné en deuxième argument. La différence entre ces deux fonctions est que la première retourne une liste quand la deuxième cherche à retourner un vecteur (ou matrice ou tableau) si possible.

```
> # lapply retourne une liste
> lapply(v,function(e) e*2)
[[1]]
[1] 2
...
[[3]]
[1] 4

> # sapply retourne un vecteur
> sapply(v,function(e) e*2)
[1] 2 6 4
```

Instructions ***sapply*** *(Suite)*

sapply() peut retourner une matrice (voire un tableau).

```
> # sapply retourne un vecteur
> sapply(v,function(e) c(e*2,e^2))
     [,1] [,2] [,3]
[1,]    2    6    4
[2,]    1    9    4
> # t() est la fonction transposée
> t(sapply(v,function(e) c(e*2,e^2)))
     [,1] [,2]
[1,]    2    1
[2,]    6    9
[3,]    4    4
```

Instruction ***apply***

C'est une extension de la fonction **sapply** applicables aux matrices et tableaux.

```
> mat
     [,1] [,2] [,3] [,4]
[1,]    1   30    5    7
[2,]    2    4    6    8
> apply(mat,1,sum)    # marginal en lignes
[1] 43 20
> apply(mat,2,sum)    # marginal en colonnes
[1]  3 34 11 15
> # différence entre apply et sapply
> apply(mat,1:2,function(e) e*2)  # une matrice
     [,1] [,2] [,3] [,4]
[1,]    2   60   10   14
[2,]    4    8   12   16
> sapply(mat,function(e) e*2)     # un vecteur
[1]  2  4 60  8 10 12 14 16
```

*Fonctions (**function**)*

- Le R est un langage fonctionnel : la plupart des actions dans le système R correspondent à des appels de fonctions (de façon directe voire indirecte).
- Une fonction en R est un objet comme les autres qui peut donc être affectée à une variable (cas le plus fréquent) ou pas (voir usage avec **sapply()**.
- Une fonction est définie en fournissant après la clause **function** une liste nommée d'arguments fournis entre parenthèses. Si un nom d'argument est suivi d'un "=" puis d'un objet R , cela vaut de valeur par défaut.
- Un appel de fonction consiste à saisir son suivre de parenthèses contenant éventuellement des valeurs des arguments. Le dernier objet R exécuté dans le corps de la fonction est retourné lors de l'appel de la fonction

Fonctions (Suite)

```
1   > # fonction non stockée dans une variable
2   > (function(x) x^2)(3)
3   [1] 9
4   > # fonction stockée dans une variable
5   > carré <- function(x) x^2
6   > carré(3)
7   [1] 9
8   > # fonction avec argument ayant valeur par défaut
9   > carré <- function(x=3) x^2
10  > carré()     # parenthèses objigatoires pour appel
11  [1] 9
12  > carré       # retourne le contenu de l'objet R
13  function (x = 3)
14  x^2
```

Fonctions (Suite)

```
> # invisible() évite affichage du résultat
> carré <- function(x=3) invisible(x^2)
> carré(3:5)    # cela ne semble pas marcher!
> c3 <- carré(3:5)
> c3          # l'objet retourné mais pas visible
[1]  9 16 25
```

Lorsqu'on a besoin de retourner plusieurs objets R , cela se fait indirectement en retourner un objet R composite comme une liste **list** ou un environnement **environment**.

```
> puissance <- function(x=3) list(carré=x^2,cube=x^3)
> puissance()
$carré
[1] 9

$cube
[1] 27

> res <- puissance(2:9)
> res
$carré
[1]  4  9 16 25 36 49 64 81

$cube
[1]   8  27  64 125 216 343 512 729

```

Opérateur défini par utilisateur et Ellipsis ...

Les deux actions définies ci-dessous sont assez communes dans beaucoup d'autres langages. L'utilisateur (mode développeur) peut facilement les définir.

```
> # définition de l'opérateur
> "%+%" <- function(ch,ch2) paste(ch,ch2,sep="")
> # application
> "toto" %+% "titi"
[1] "tototiti"
> # application successive
> "toto" %+% " et " %+% "titi"
[1] "toto et titi"

> # "..." comme un copier-coller
> join <- function(...,sep="")
+         paste(c(...),collapse=sep)
> join("toto"," et ","titi")
[1] "toto et titi"
```

Programmation Objet S3 en R (++)

Déclaration d'objets d'une certaine classe : la classe d'un objet R est ni plus ni moins qu'un attribut **class** attaché à l'objet manipulable indirectement par les fonctions **class()** et **class()**<-.
La fonction ci-dessous sert de constructeur à la classe Human. Il n'est pas nécessaire en R de développer une telle fonction mais cela est généralement un bon usage.

```
1  > Human <- function(firstName,lastName) {
2  +     being <- list(last=lastName,first=firstName)
3  +     class(being) <- "Human"
4  +     being
5  + }
6
7  > agent007 <- Human("James","Bond")
```

L'affichage de l'objet se fait ici en utilisant la méthode par défaut **print.default**.

Programmation Objet : fonction générique

L'affichage de l'objet lors de session interactive se fait indirectement via le mécanisme OOP. En fait, l'objet retourné est affiché via l'appel automatique de la fonction **print()**. Cette fonction, dite **générique**, a pour but de rediriger l'appel vers la méthode relative à la classe de l'objet. Si l'objet n'a pas de méthode spécifique à sa classe (fonction de la forme **print.<classe>()**), la méthode par défaut **print.default()** est appelée comme dans l'exemple suivant.

```
1  > agent007 # print(agent007) => print.default(agent007
2  $last
3  [1] "Bond"
4
5  $first
6  [1] "James"
7
8  attr(,"class")
9  [1] "Human"
```

Programmation Objet : méthode de classe

Après définition de la méthode d'affichage spécifique à la classe **Human**, l'affichage est modifié :

```
> print.Human <- function(human) {
+     cat("My name is",human$first,human$last,"\n")
+ }
> agent007 # print(agent007) => print.Human(agent007)
My name is James Bond
> print.default(agent007) # affichage de base!
$last
[1] "Bond"

$first
[1] "James"

attr(,"class")
[1] "Human"
```

Programmation Objet : nouvelle méthode de classe

Avant de définir une nouvelle méthode de classe, il faut s'assurer de l'existence de la fonction générique associée (voir par exemple, **print()**) :

```
1  > print # sans parenthèses!
2  function (x, ...)
3  UseMethod("print")
4  <bytecode: 0x7fba464d7a68>
5  <environment: namespace:base>
```

Appliquons ce mécanisme à l'action **chante** qui diffèrera selon la nature de la classe de l'objet appelé.

```
1  > chante <- function(obj,...) UseMethod("chante")
2  > chante.Human <- function(obj,texte)
3  +    cat("la! la! la!",texte,"\n")
4  > chante(agent007,"mon nom est Bond!")
5  la! la! la! mon nom est Bond!
```

Programmation Objet : héritage de classe

La gestion d'héritage est simple en R puisque l'attribut **class** peut prendre plusieurs valeurs de classe. Le mécanisme R est très simple puisque la méthode appliquée est la première valeur de l'attribut **class** à pouvoir s'appliquer. Si aucune méthode n'est trouvée, la fonction **<méthode>.default()** (si existante) est appliquée.

```
> Worker <- function(first,last,job="Secret Agent") {
+     obj <- Human(first,last)
+     obj$job <- job
+     class(obj) <- c("Worker","Human")
+     obj
+ }

> worker007 <- Worker("James","Bond")

> worker007 # i.e. print.Human(worker007)
My name is James Bond
```

Programmation Objet : surchage de méthode

```
> print.Worker <- function(worker) {
+     print.Human(worker) # et non print(worker)!
+     cat("My job is",worker$job,"\n")
+ }
> worker007
My name is James Bond
My job is Secret Agent
```

Argument de **UseMethod** peut différer du nom de fonction générique !

```
> sing <- function(obj,...) UseMethod("chante")
> chante.Worker <- function(w,texte)
+     cat("Yo! Yo! Je suis",w$last,"le",w$job,"!")
> sing(worker007)
Yo! Yo! Je suis Bond le Secret Agent !
> chante(worker007)
Yo! Yo! Je suis Bond le Secret Agent !
```

Manipulation code R en R (++)

Pour information, soulignons que le R est un langage permettant la manipulation introspective de code R . En particulier, en complément des fonctions, il est possible de définir des greffons de code R dont l'exécution pourra être différée.

```
1   > expr <- expression(a + b) #expression
2   > expr
3   expression(a + b)
4   > class(expr)
5   [1] "expression"
6   > as.list(expr)          #liste des "call"
7   [[1]]
8   a + b
9
10  > as.list(expr)[[1]]    #le "call"
11  a + b
12  > class(as.list(expr)[[1]])
13  [1] "call"
```

Manipulation code R en R (Suite ++)

```
> code <- "a    +   b"
> expr2 <- parse(text=code) # comme expr
> expr2
expression(a    +   b)
> as.list(expr2)[[1]]
a + b
> class(expr2)
[1] "expression"
```

Une expression peut ensuite être évaluée à partir de la fonction **eval()**

```
> a<-2;b<-3
> eval(expr)
[1] 5
> eval(expr2)
[1] 5
```

Plan

Import et export des données

```
> # import données (à distance)
> imcEnf <- read.table(
+    "http://www.biostatisticien.eu/springeR/imcenfant.txt",
+    header=TRUE
+ )
> names(imcEnf)
[1] "SEXE"   "zep"    "poids"  "an"     "mois"   "taille"
> names(imcEnf) <- tolower(names(imcEnf))
> head(imcEnf)
  sexe zep poids an mois taille
1    F   O  16.0  3    5  100.0
2    F   O  14.0  3   10   97.0
3    G   O  13.5  3    5   95.5
4    F   O  15.4  4    0  101.0
5    G   N  16.5  3    8  100.0
6    G   O  16.0  4    0   98.5
> # export des données
> write.table(imcEnf,file='imcEnf.txt')
```

Pour compléter les imports, voir aussi : **read.csv()**, **read.csv2()**, **read.delim()**, **read.delim2()** et **read.xls()** du package gdata.

Ajout de variables

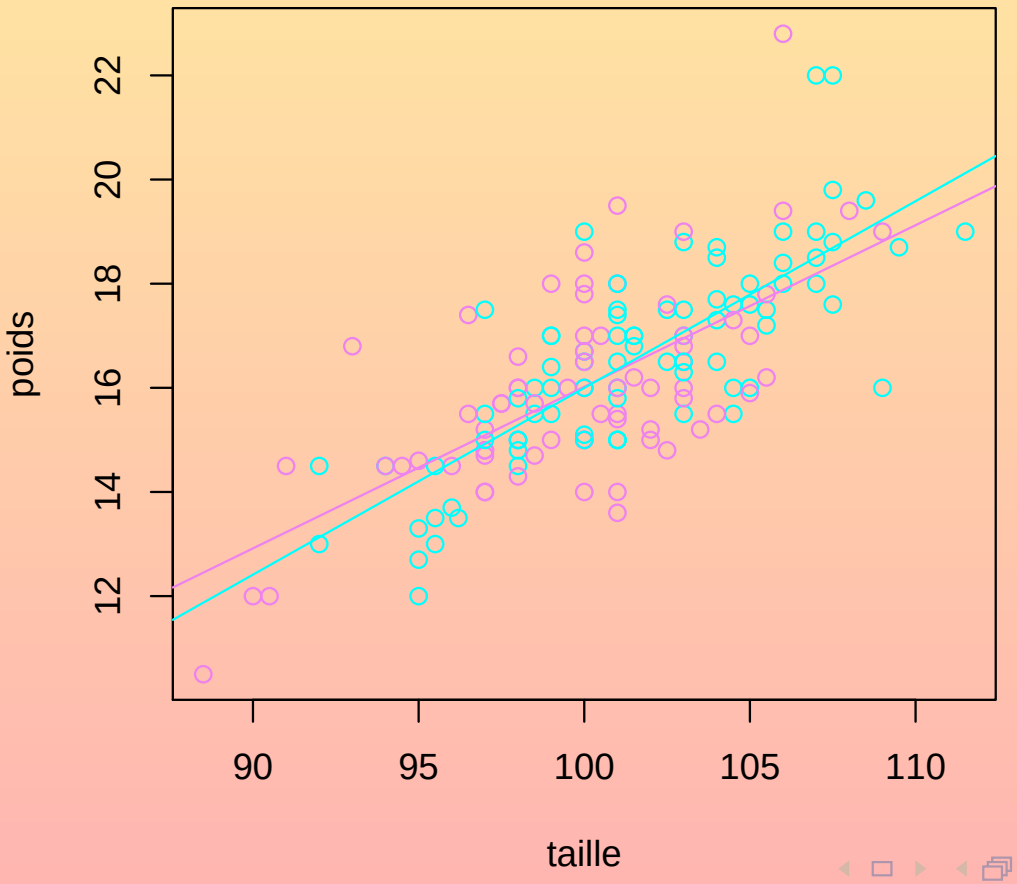
```
> attach(imcEnf)
> # essayer
> poids/(taille/100)^2
  [1] 16.00000 14.87937 14.80223 15.09656 16.50000 16.49102 16.02413 15.41
...
[145] 14.86545 19.11577 13.29640 16.50125 17.34517 14.88963 15.99245 16.18
> # enregistrer
> poids/(taille/100)^2 -> imcEnf$imc
> detach()
> head(imcEnf)
  sexe zep poids an mois taille      imc
1    F   O  16.0  3    5  100.0 16.00000
2    F   O  14.0  3   10   97.0 14.87937
3    G   O  13.5  3    5   95.5 14.80223
4    F   O  15.4  4    0  101.0 15.09656
5    G   N  16.5  3    8  100.0 16.50000
6    G   O  16.0  4    0   98.5 16.49102
```

Ajout de variables (solution alternative)

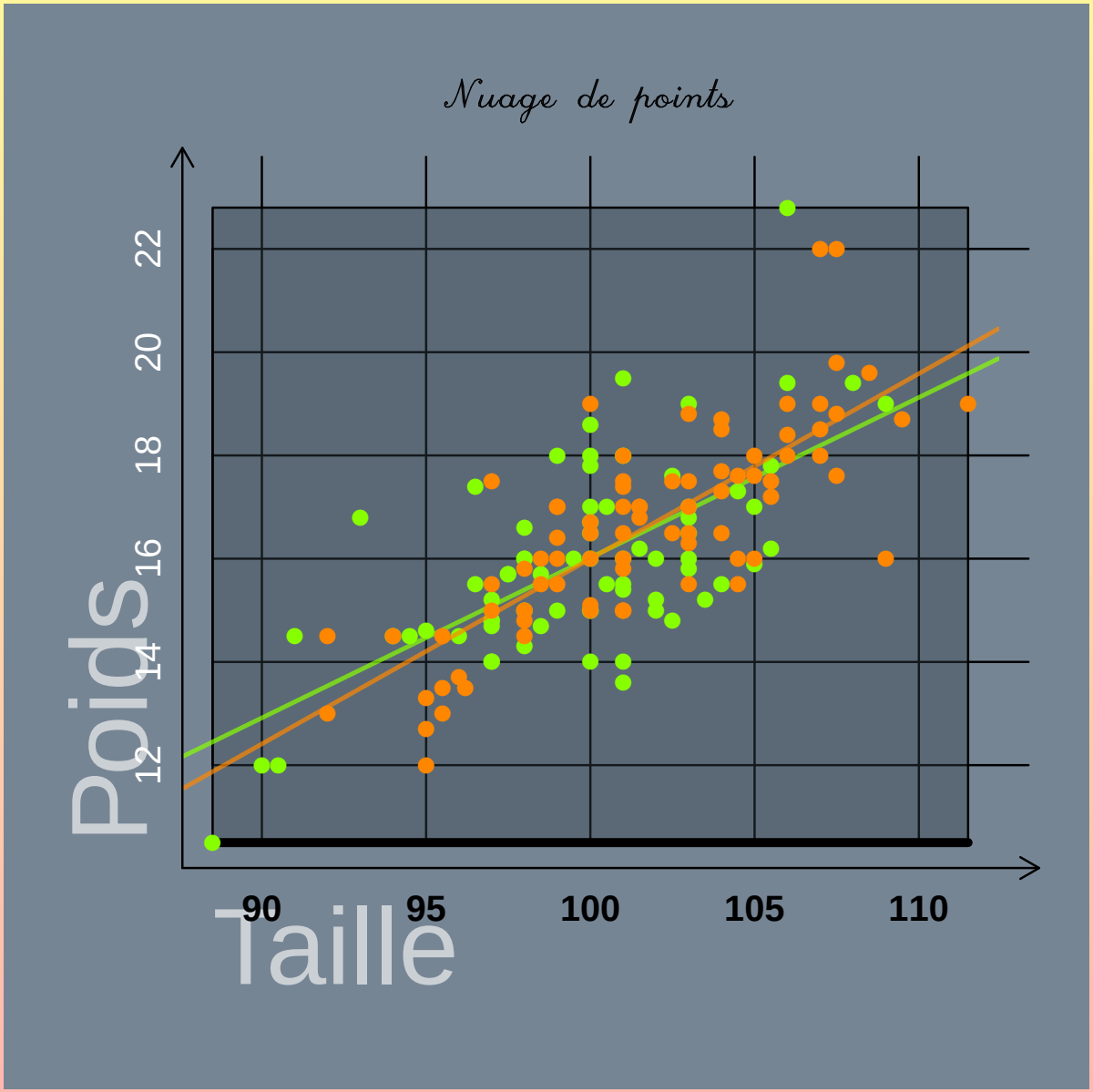
```
> # essayer
> with(imcEnf,poids/(taille/100)^2)
  [1] 16.00000 14.87937 14.80223 15.09656 16.50000 16.49102 16.02413 15.41
...
[145] 14.86545 19.11577 13.29640 16.50125 17.34517 14.88963 15.99245 16.18
> within(imcEnf,imc <- poids/(taille/100)^2)
    sexe zep poids an mois taille      imc
1      F   O  16.0  3    5  100.0 16.00000
2      F   O  14.0  3   10   97.0 14.87937
3      G   O  13.5  3    5   95.5 14.80223
...
150    F   N  14.3  3    4   98.0 14.88963
151    F   N  17.8  3   11  105.5 15.99245
152    F   N  15.7  3    7   98.5 16.18181
> # enregistrer
> within(imcEnf,imc <- poids/(taille/100)^2) -> imcEnf
> head(imcEnf)
  sexe zep poids an mois taille      imc
1    F   O  16.0  3    5  100.0 16.00000
2    F   O  14.0  3   10   97.0 14.87937
3    G   O  13.5  3    5   95.5 14.80223
4    F   O  15.4  4    0  101.0 15.09656
5    G   N  16.5  3    8  100.0 16.50000
6    G   O  16.0  4    0   98.5 16.49102
```

Nuage de points

```
> attach(imcEnf);col <- c("violet","cyan")
> # poids~taille est un objet R de classe "formula"
> plot(poids~taille,col=col[sexe])
> abline(lm(poids~taille,subset=sexe=="F"),col=col[1])
> abline(lm(poids~taille,subset=sexe=="G"),col=col[2])
```



Flashy nuage de points

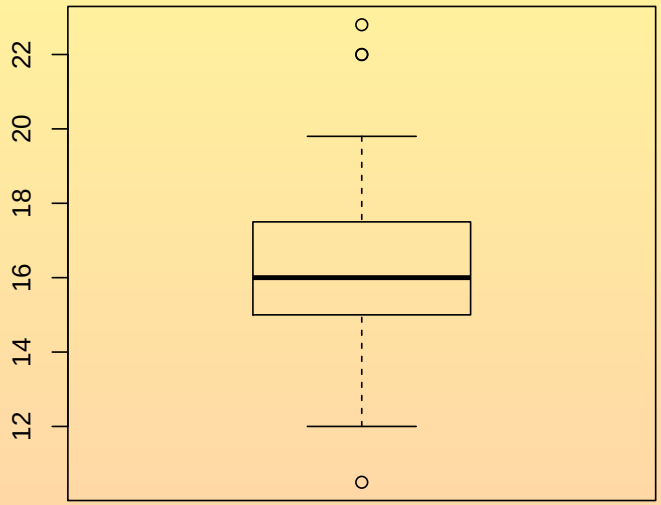


Flashy nuage de points (une partie du code)

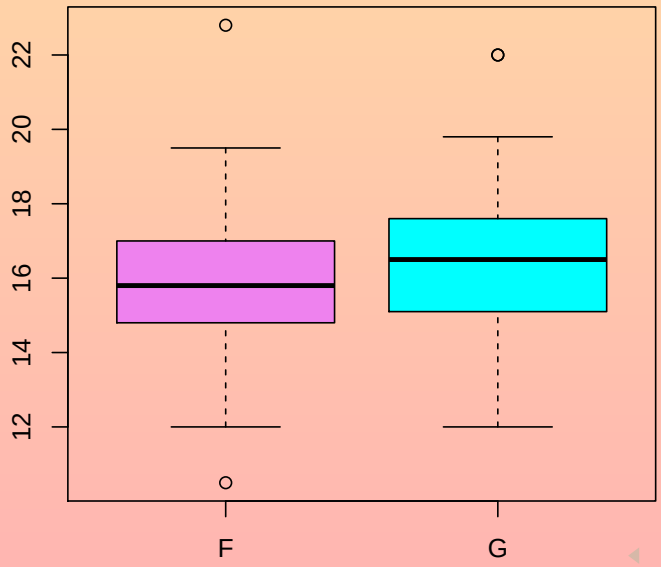
```
> flashy.plot <- function(x,y,facteur,xlab="Taille",
+ ylab="Poids",family="HersheyScript") {
+    par(bg = "#768594", tck = 1)
+    plot.new()
+    mx <- min(x)
+    Mx <- max(x)
+    my <- min(y)
+    My <- max(y)
+    plot.window(xlim = c(mx, Mx), ylim = c(my, My))
+    text(145, 95, "@", cex = 10, col = "#404F5E", xpo
+    text(mx, my - 2, xlab, col = "#cbd0d5", cex = 3,
+        adj = 0)
+    text(mx - 2, my, ylab, col = "#cbd0d5", cex = 3,
+        srt = 90, adj = c(0, 0))
+    axis(1, pos = my, font = 2)
+    axis(2, pos = mx, col.axis = "white")
+    polygon(c(mx, Mx, Mx, mx), c(my, my, My, My), col
```

Boîte à moustaches

```
> boxplot(poids)
```




```
> boxplot(poids~sexe,col=col)
```



Histogramme

```
> hist(poids,prob=TRUE,col="red",main="Histogramme")
> # ajout estimation densité
> lines(density(poids),lwd=3,col="blue")
```

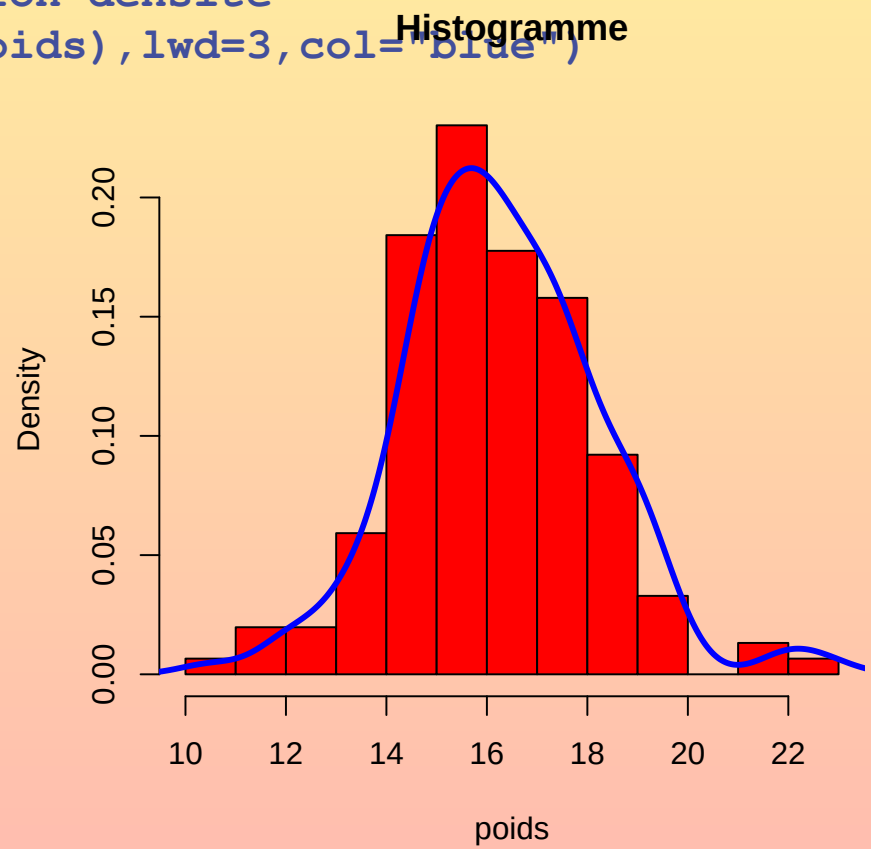
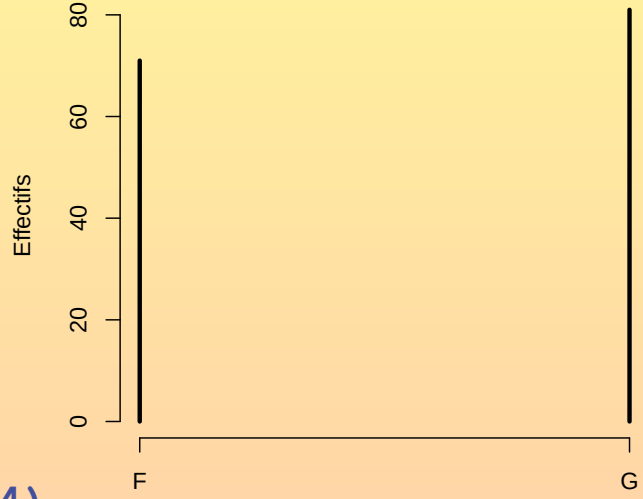


Diagramme en bâton

```
> plot(table(sexe),lwd=3,ylab="Effectifs")
```




```
> cPoids <- cut(poids,4)
> plot(table(cPoids),lwd=3,ylab="Effectifs",col=1:4)
```

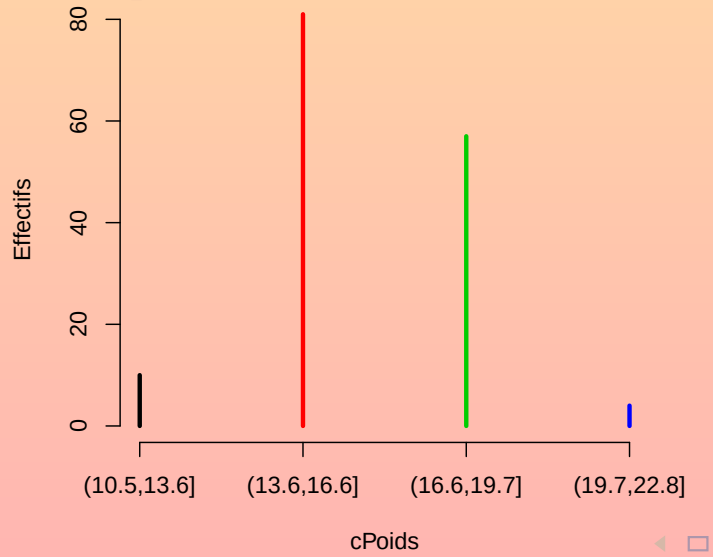


Diagramme en point

```
> dotchart(table(cPoids), col=1:4)
```

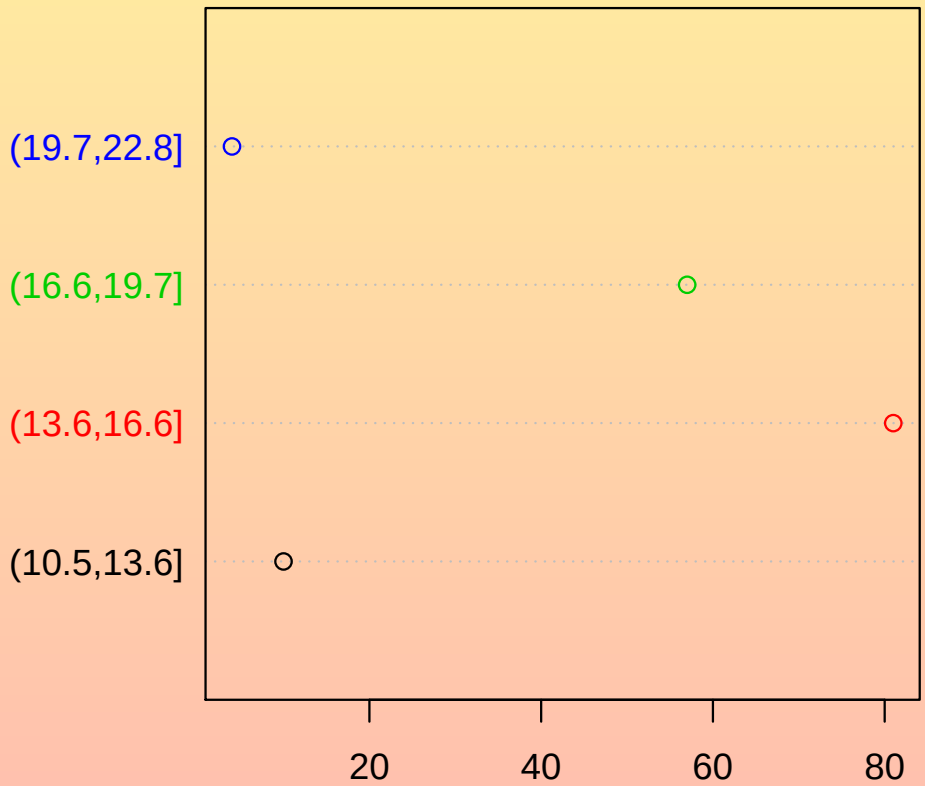
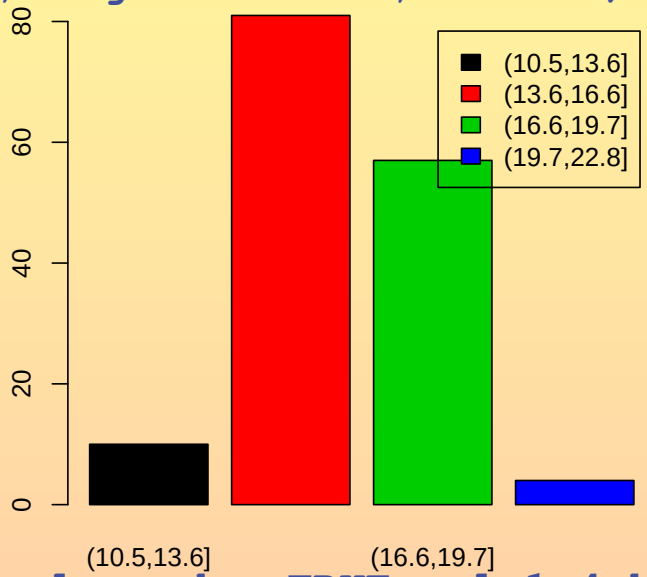


Diagramme en barre

```
> barplot(table(cPoids), legend = TRUE,col=1:4)
```




```
> barplot(table(cPoids), legend = TRUE,col=1:4,horiz=TRUE)
```

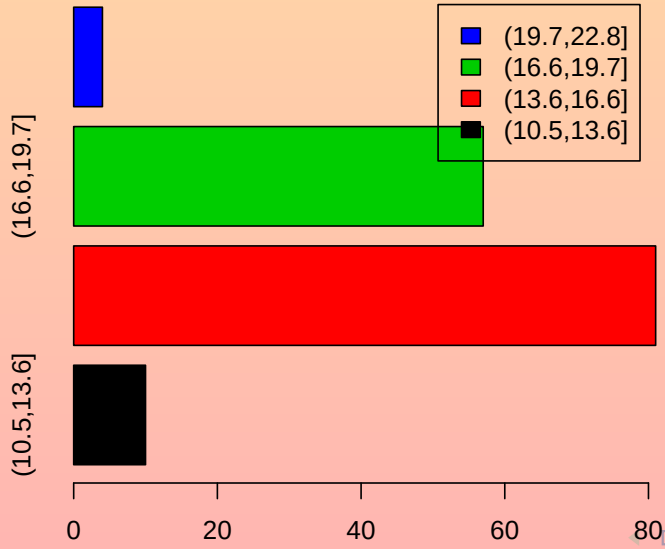


Diagramme circulaire (camembert)

```
> pie(table(cPoids), col=1:5)
```

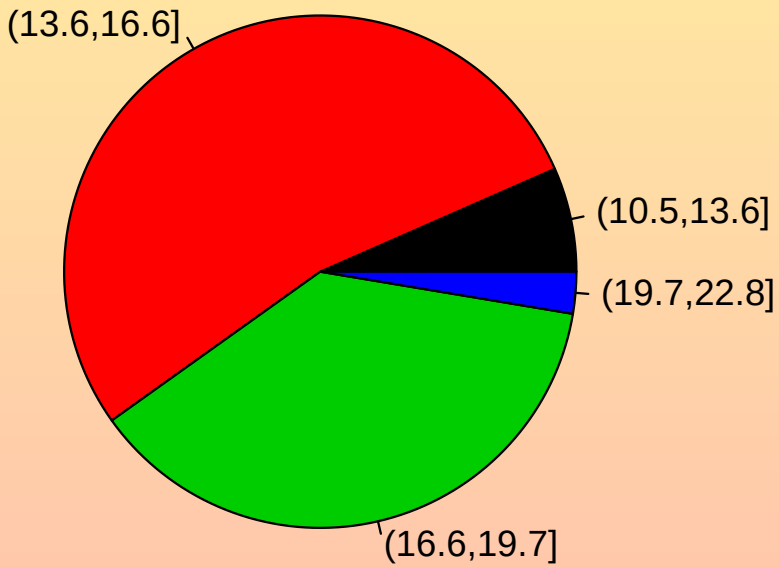
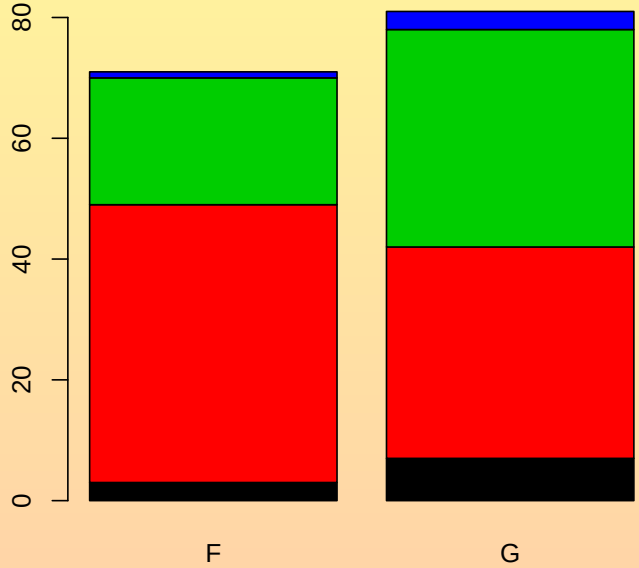
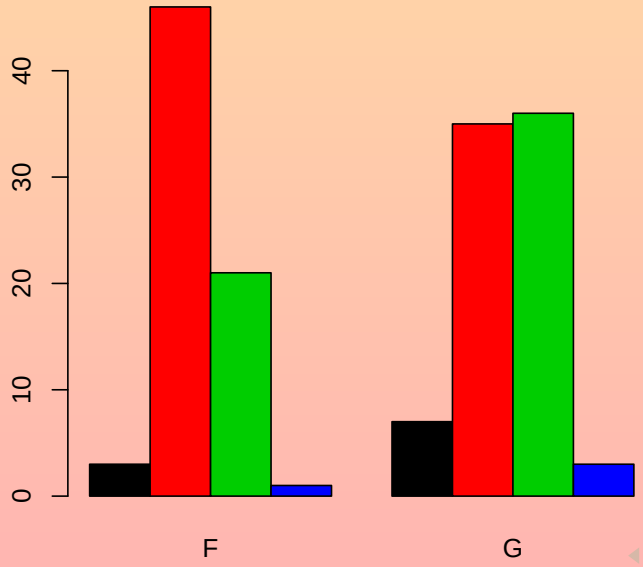


Diagramme en barre groupé

```
> barplot(table(cPoids,sexe),col=1:4)
```




```
> barplot(table(cPoids,sexe),col=1:4,beside=TRUE)
```



Quelques traitements à la volée

```
> range(poids) # étendue
[1] 10.5 22.8
> c(which.min(poids),which.max(poids)) # indices
[1]  41 100
> poids[c(which.min(poids),which.max(poids))] # étendue
[1] 10.5 22.8
> sort(poids) # tri
  [1] 10.5 12.0 12.0 12.0 12.7 13.0 13.0 13.3 13.5 13.5 13.6 13.7 14.0 14.
...
[151] 22.0 22.8
> order(poids) # rangs
  [1]  41  68  91 147  63  47 134  69   3 124  50 145   2  83 126 144 150
...
[145] 109 113 146 142  23  59 101 100
> poids[order(poids)] # tri
  [1] 10.5 12.0 12.0 12.0 12.7 13.0 13.0 13.3 13.5 13.5 13.6 13.7 14.0 14.
...
[151] 22.0 22.8
> # sélection d'indices
> which(poids>20)
[1]  59 100 101
> (1:length(poids))[poids>20] # idem
[1]  59 100 101
> seq(poids)[poids>20]        # idem
[1]  59 100 101
```

Quelques traitements à la volée (Suite)

```
> # sous-matrice de données
> imcEnf[poids>20,]
    sexe zep poids an mois taille      imc
59     G   O  22.0  3   11  107.0 19.21565
100    F   O  22.8  3    9  106.0 20.29192
101    G   O  22.0  4    4  107.5 19.03732
> # médiane, moyenne, variance, écart-type
> c(median(poids),mean(poids),var(poids),sd(poids))
[1] 16.000000 16.280263  3.741727  1.934354
> #variance de la population
> var.pop<- function(x) var(x)*(length(x)-1)/length(x)
> var.pop(poids)
[1] 3.71711
> sd.pop<- function(x) sqrt(var.pop(x))
> sd.pop(poids)
[1] 1.927981
> # quartiles
> quantile(poids,seq(0,1,by=1/4))
  0%  25%  50%  75% 100%
10.5 15.0 16.0 17.5 22.8
> # résumé (par défaut) pour une variable quantitative
> summary(poids)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  10.50   15.00   16.00   16.28   17.50   22.80
```

Tableaux croisés

```
> # tableaux croisés
> (tab_crois<-table(sexe,cPoids))
     cPoids
sexe (10.5,13.6] (13.6,16.6] (16.6,19.7] (19.7,22.8]
   F           3           46           21           1
   G           7           35           36           3
> addmargins(tab_crois,FUN=sum,quiet=TRUE)
      cPoids
sexe   (10.5,13.6] (13.6,16.6] (16.6,19.7] (19.7,22.8] sum
  F              3           46           21           1  71
  G              7           35           36           3  81
  sum           10           81           57           4 152
> # Pourcentage
> (tab_pctT<-round(tab_crois/sum(tab_crois)*100,2))
     cPoids
sexe (10.5,13.6] (13.6,16.6] (16.6,19.7] (19.7,22.8]
   F         1.97        30.26        13.82         0.66
   G         4.61        23.03        23.68         1.97
```

Tableaux croisés (Suite)

```
> # les marges %lignes
> margin.table(tab_pctT,1)
sexe
    F     G
46.71 53.29
> # les marges %colonnes
> margin.table(tab_pctT,2)
cPoids
(10.5,13.6] (13.6,16.6] (16.6,19.7] (19.7,22.8]
       6.58       53.29       37.50        2.63
> # conditionnelles %lignes
> addmargins(prop.table(tab_crois,1),margin=2,FUN=sum)
    cPoids
sexe (10.5,13.6] (13.6,16.6] (16.6,19.7] (19.7,22.8]        sum
   F  0.04225352  0.64788732  0.29577465  0.01408451 1.00000000
   G  0.08641975  0.43209877  0.44444444  0.03703704 1.00000000
> # conditionnelles %colonnes
> addmargins(prop.table(tab_crois,2),margin=1,FUN=sum)
     cPoids
sexe  (10.5,13.6] (13.6,16.6] (16.6,19.7] (19.7,22.8]
  F     0.3000000   0.5679012   0.3684211   0.2500000
  G     0.7000000   0.4320988   0.6315789   0.7500000
  sum   1.0000000   1.0000000   1.0000000   1.0000000
```