Chapitre 9 Aller plus loin avec les objets, crochets et la programmation fonctionnelle

Ce qui a été présenté dans ce module reposer sur les fonctions du package `tidyverse` . Cette approche tend à se généraliser depuis quelques années, mais quand on cherche la réponse à un problème sur Internet, on trouve d'autres façons de programmer en R, qui appellent aux éléments du package `base` et non du `tidyverse` \(\rightarrow\) Cete partie donne quelques clés de compréhension.

9.1 Les objets dans R

Rappel : en informatique, un objet est défini par : ses *attributs* et ses *méthodes* (fonctions). On prend l'exemple du jeu d'échec : chaque pièce peut-être vue comme un objet :

- sa position sur le plateau constitue ses attributs
- sa façon de se déplacer peut-être vue comme une fonction qui ne s'applique qu'à ce type de pièce, donc une méthode

R est un langage orienté objet ; ces objets permettent de structurer les données selon leurs caractéristiques \(\rightarrow\) on retrouve les données dans les attributs. Les méthodes sont en général transparentes pour l'utilisateur (cf utilisation des fonctions `summary` , `plot` …). Les objets les plus courants sont les suivants :

- **Vecteurs** : suite unidimensionnelle de **valeurs** ayant le même type.
- **Facteurs** : vecteur qui prend un nombre limité de modalités (exemple : sexe). Il est défini par les niveaux (*levels*) et les libellés associés (*labels*)
- **Matrice** et **arrays** : suites multidimensionnelles de **valeurs** (matrices=dimension 2 ; array=dimension n)
- **Liste** : ensemble d'**objets** différents. On peut stocker un vecteur alphanumérique avec matrice numérique dans une liste
- **Tableaux** (data.frame) : Objet qui ressemble le plus aux tables Excel, SAS ou SPSS… : description d'individus statistiques (observations, en ligne) par des caractéristiques (variables, en colonnes)
- **Fonctions** : Objets particuliers qui donnent un *résultat* à partir de paramètres en entrée. Cela sera au programme d'un autre module de formation.
- **Autres objets** : Il existe un très grand nombre d'objets *ad hoc* dans R. Par exemple
  - *ts* (time serie) pour les séries temporelles,
  - *lm* (linear model) qui contient tous les résultats d'une régression linéraire…
  - On peut même en définir de nouveaux soi-même !

On utilise `<-` ou `=` pour stocker un objet. La fonction `c` permet de lister simplement les valeurs que l'on veut stocker dans l'objet ; la fonction `seq` génére une suite incrémentée. Il existe aussi la fonction `rep` qui réplique n fois la même valeur.

9.1.1 Créer des vecteurs

```
v1 <- seq(1 : 10)
v2 <- c ("lundi", "mardi", "mercredi", "jeudi",
        "vendredi", "samedi", "dimanche")
```

9.1.2 Créer une matrice

```
m <- matrix (v1, nrow = 10, ncol = 7)
l <- list (v1, v2, m)
```

9.1.3 Visualiser ces objets et leurs attributs

Quelques fonctions simples : `print` , `View` , `names` , `colnames` , `rownames` , `typeof` , `str` , `dim`

```
v1
```

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

```
print (v2)
```

```
## [1] "lundi"    "mardi"    "mercredi" "jeudi"    "vendredi" "samedi"
## [7] "dimanche"
```

```
typeof (v2) # Permet de visualiser le type
```

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

```
dim (m)
```

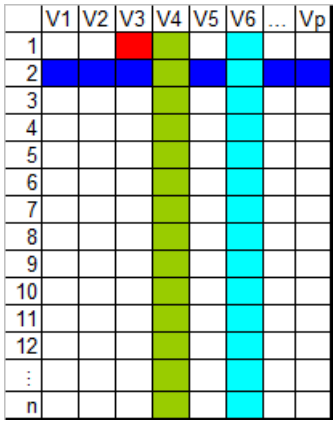
```
## [1] 10  7
```

```
str (l)    # Permet de visualiser les attributs
```

```
## List of 3
##  $ : int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ : chr [1:7] "lundi" "mardi" "mercredi" "jeudi" ...
##  $ : int [1:10, 1:7] 1 2 3 4 5 6 7 8 9 10 ...
```

9.2 Sélectionner des lignes et des colonnes

Il est aussi possible d'accéder aux éléments d'un dataframe à partir du numéro de ligne et de colonne, grâce aux crochets :



- `base[1,3]` \(\rightarrow\) valeur de la première ligne et de la troisième colonne
- `base[2,]` \(\rightarrow\) toutes les variables pour la 2e observation
- `base[,4]` \(\rightarrow\) toutes les observations de la quatrième colonne
- `base[,'V6']` \(\rightarrow\) toutes les observations de la variable V6
- \(\Rightarrow\) Utile pour sélectionner une partie d'une table : `base[1:4, c(3, 6)]`

9.3 Créer une nouvelle fonction en R

La fonction est un objet comme les autres, qu'on crée avec l'opérateur d'affectation. Elle est définie par des paramètres et elle se termine par la fonction `return()` . On reprend l'exemple du calcul de l'IMC

```
calcul_IMC <- function (poids, taille)
{
  ## La taille est exprimée en mètres
  imc <- poids / taille ^ 2
  return (imc)
}
calcul_IMC (poids=80,taille=1.89)
```

```
## [1] 22.39579
```

```
calcul_IMC (poids=60,taille=1.55)
```

```
## [1] 24.97399
```

9.4 Les boucles conditionnelles

Les commandes `if` et `else` sont bien entendues utilisables. Le “then” n’existe pas : il est implicite après les accolades.

```
diag_IMC <- function(poids,taille)
{
  imc <- poids / taille ^ 2
  if (imc < 18.5) {diag <- "maigre"}
  else if (imc < 25) {diag <- "normal"}
       else {diag <- "surpoids"}
  return (diag)
}
diag_IMC (poids=60,taille=1.89)
```

```
## [1] "maigre"
```

```
diag_IMC (poids=80,taille=1.89)
```

```
## [1] "normal"
```

```
diag_IMC (poids=80,taille=1.55)
```

```
## [1] "surpoids"
```

9.5 Les boucles

On peut utiliser les boucles classiques : `repeat` , `while` , `for` :

```
for (pp in seq(from = 50, to = 100, by = 5))
{
  print(paste ("Taille = 1,70m, poids =", pp, "Diagnotic :",
               diag_IMC (poids = pp, taille = 1.70)))
}
```

```
## [1] "Taille = 1,70m, poids = 50 Diagnotic : maigre"
## [1] "Taille = 1,70m, poids = 55 Diagnotic : normal"
## [1] "Taille = 1,70m, poids = 60 Diagnotic : normal"
## [1] "Taille = 1,70m, poids = 65 Diagnotic : normal"
## [1] "Taille = 1,70m, poids = 70 Diagnotic : normal"
## [1] "Taille = 1,70m, poids = 75 Diagnotic : surpoids"
## [1] "Taille = 1,70m, poids = 80 Diagnotic : surpoids"
## [1] "Taille = 1,70m, poids = 85 Diagnotic : surpoids"
## [1] "Taille = 1,70m, poids = 90 Diagnotic : surpoids"
## [1] "Taille = 1,70m, poids = 95 Diagnotic : surpoids"
## [1] "Taille = 1,70m, poids = 100 Diagnotic : surpoids"
```

9.6 Exercices

9.6.1 Vecteurs simples

- Créer trois vecteurs : un numérique, un caractère et un facteur. Vous pouvez vous aider des fonctions `c()`, `rnorm()` (génération d'une variable aléatoire selon une loi normale), `seq()` ou `rep()`
- Regrouper ces variables dans un dataframe, puis dans une liste. Dans les deux configurations, afficher la variable contenant les tailles (vtaille). Pour la liste, utiliser les [] et [[]].

```
rm (list = ls ())

x <- c (1, 160, 2, 9, 60)
x1 <- c("Je", "programme", "en", "R") # Guillemets pour indiquer que c'est une variable textuelle
y <- seq (from = 1, to = 10, by = 1)
z <- rep (x = 1, times = 100)
x <- rnorm (n = 30)

# création de vecteurs avec la fonction c() = combine
v1 <- c( 3, 4, 12, 15, 32, 6, 1, 2, 3, 9)

# avec la fonction seq() = sequence, g?n?ralisation de la syntaxe ci-dessus
v2 <- seq(from = 1 , to = 15 , by = 1.5)

# syntaxe équivalente mais préférable car plus lisible :

v2b <- seq (from=1, to=15, by=1.5)
v3 <- 1:10

# avec la fonction rep() = répétition
v4 <- rep (x = 4, times = 10)

# ces commandes peuvent être combinées. Pratique pour cr?er des variables "facteur"
v5 <- rep (x = c(3, 4, 1.2, 8, 9), times = 2)
v6 <- rep (x = 1:5, times = 2)

# vecteurs de type texte ou factor
vtaille <- rep (x = c ("S", "L"), times = 5)
vtaille <- factor (vtaille)

# concaténation de vecteurs
gtaille <- paste("X", vtaille, sep = "")
gtaille <- factor (gtaille)
toutes_taille <- c (as.character (vtaille), as.character (gtaille))
toutes_taille <- as.factor (toutes_taille)
levels (toutes_taille)


## [1] "L"  "S"  "XL" "XS"
```

9.6.2 Dataframes et listes

```
dataf <- data.frame (vtaille, v1, v2, v3, v4, v5, v6)
liste <- list (vtaille, v1, v2, v3, v4, v5, v6)
names(liste) <- c ("vtaille", "v1", "v2", "v3", "v4", "v5", "v6")

dataf$vtaille
```

```
##  [1] S L S L S L S L S L
## Levels: L S
```

```
liste$vtaille
```

```
##  [1] S L S L S L S L S L
## Levels: L S
```

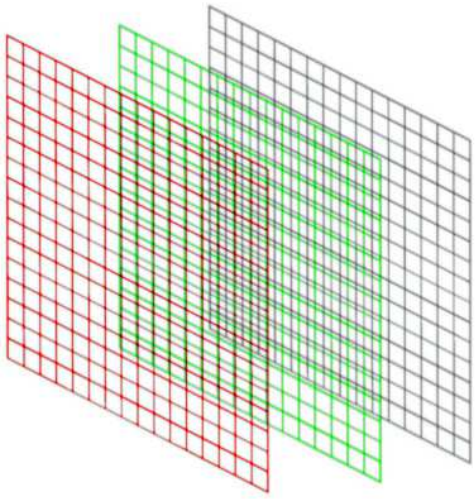
```
rm (dataf, vtaille, v1, v2, v2b, v3, v4, v5, v6)
```

9.6.3 Pour aller plus loin : matrices et arrays

Les matrices et les arrays permettent des calculs rapides et efficaces, et peuvent être très pratiques et optimisent le stockage des données. Ils demandent cependant plus de réflexion en amont quant à leur utilisation, mais . On accède aux éléments avec les [].

- Créer une matrice à 10 lignes et 10 colonnes remplie avec un tirage aléatoire selon une loi normale centrée réduit.
- Créer un hypercube avec la fonction `array()` avec 10 lignes, 5 colonnes et de profondeur 3, toujours avec un tirage aléatoire selon une loi normale

Un hypercube de trois dimensions peut être représenté comme suit :



```r
mat <- matrix(rnorm(50), ncol = 5, nrow = 10)
arr <- array(rnorm(150),dim = c(10,5,3))
mat
```

```
##              [,1]         [,2]       [,3]        [,4]         [,5]
##  [1,]  0.05500996  1.254117912  0.5955068 -1.65628079  1.738884188
##  [2,] -0.71677459 -0.399234727  0.8888565  0.10867942  0.800492054
##  [3,]  1.23079463 -0.549102438 -0.3713453  0.55597325 -0.211004529
##  [4,]  0.85939861  1.112033074 -0.1525016 -0.56735650  0.407098376
##  [5,]  0.06496963  0.215657449  0.5017767  0.93868042  0.293883373
##  [6,] -1.58952322  0.392893763 -0.9344764  0.33907658  1.093055506
##  [7,]  1.27184406  0.005083936 -0.4396057  1.58215357 -0.588753832
##  [8,] -2.23095991  0.901056879 -1.6862659 -0.09344853 -1.600648254
##  [9,]  0.31194736  0.084469756  0.4774278  1.00101060 -0.212207986
## [10,]  0.20492429 -1.327907248  1.2847502 -0.17684064  0.003584898
```

```r
arr
```

```
## , , 1
##
##              [,1]        [,2]        [,3]        [,4]         [,5]
##  [1,]  0.43890465  0.44432399 -1.58331269 -1.13813628  0.058845317
##  [2,] -0.46493797 -0.74429698 -1.07445978  0.45103349 -0.871139700
##  [3,] -0.46418448  1.00192443  0.05210394  0.07843470 -1.178293529
##  [4,]  1.38085887 -0.39282568  0.85057754  1.92707690  0.093523409
##  [5,] -0.90529717 -0.92073612  0.82767741 -0.02031069 -0.344907516
##  [6,] -0.57696846 -0.49390419 -0.15725088 -1.39630054  1.141103387
##  [7,]  0.01700932 -0.29639450  1.34429357  1.59649564  0.152110261
##  [8,]  1.25588043  0.15390705  0.41056938 -1.87823656  0.008117944
##  [9,]  0.69456192 -0.35266530 -0.85908530  1.86436401  2.253453568
## [10,] -0.16509592  0.03991952 -0.19138475 -1.02721045  0.682623439
##
## , , 2
##
##              [,1]        [,2]        [,3]       [,4]       [,5]
##  [1,]  1.25559729 -0.90918793 -0.01515283 -1.5397060  0.2448052
##  [2,]  0.36902576 -0.52903164  1.39672594  1.0713547  0.4581598
##  [3,]  0.28883397  0.02503825 -0.21613752  1.8983348  1.9917220
##  [4,]  2.95375666 -0.48147380 -0.69438584 -0.4668146  1.0037914
##  [5,] -1.06227143 -1.30124657 -0.34438057 -0.1777250 -1.4211725
##  [6,]  0.74880398  1.20661582  0.68044778  0.3414470  0.3006055
##  [7,]  0.86480544 -0.78549062 -0.38128333 -0.3570671  0.6260804
##  [8,]  0.13477370  0.55691250  0.07318192 -0.1516081  0.2921254
##  [9,] -0.70229214  0.41736109 -0.58046588 -0.3230248 -0.1207460
## [10,] -0.07325855 -0.66361813  0.06540128  0.4015813 -1.0585831
##
## , , 3
##
##              [,1]        [,2]        [,3]       [,4]       [,5]
##  [1,]  1.13587484  0.92692655  1.03275070 -0.1597102 -0.8985356
##  [2,] -0.82937649  1.77824902  0.52656880  0.2472573 -0.9022758
##  [3,]  0.35037153  1.30994678 -0.51355945  1.0620587  0.2238343
##  [4,] -0.09508084 -1.05802973 -1.43121922  2.0848509 -1.3469671
##  [5,]  1.72222041  0.84101670  0.53308912  0.8450448 -0.3630200
##  [6,] -0.86148090 -1.06466850  0.96368478 -0.8233341 -0.3082909
##  [7,] -0.07535549  0.31413359 -0.13766063  0.2497568 -0.8528095
##  [8,] -0.31898576  0.55095653 -0.09710754  0.1450302  0.1880352
##  [9,]  1.38858701  0.09404948 -0.18323639 -1.3323557  2.0033563
## [10,] -0.86630702 -0.25890366  0.05107988  2.3429619  0.4859813
```

Pourquoi s'embêter avec ça ? Parce qu'on peut appliquer des fonctions facilement sur les lignes, colonnes et autres dimensions grâce à la fonction `apply()` . Exemple : résultats de validations croisées par bloc, simulations de loi selon différents paramètres. Et on calcule facilement des statistiques "marginales".

Par, exemple, sur une matrice, on peut calculer des statistiques par lignes :

```
apply(mat, MARGIN = 1, FUN=mean)
```

```
##  [1]  0.397447606  0.136403731  0.131063117  0.331734384  0.402993515
##  [6] -0.139794750  0.366144406 -0.942053144  0.332529500 -0.002297707
```

Ou par colonnes :

```
apply(mat, MARGIN = 2, FUN=mean)
```

```
## [1] -0.05383692  0.16890684  0.01641230  0.20316474  0.17243838
```

Sur notre hypercube de type `array` , on peut aussi calculer des stats sur ses différentes dimensions :

```
apply (arr, MARGIN = 3, FUN=mean)
```

```
## [1] 0.03444717 0.10622330 0.17238806
```

```
apply (arr, MARGIN = c(2,3), FUN = mean)
```

```
##             [,1]         [,2]       [,3]
## [1,]  0.12107312  0.477777468  0.1550467
## [2,] -0.15607478 -0.246412103  0.3433677
## [3,] -0.03802716 -0.001604906  0.0744390
## [4,]  0.04572102  0.069677226  0.4661561
## [5,]  0.19954366  0.231678821 -0.1770692
```

Le coin du capitaine [ ]

Le crochet, c'est comme le capitaine du même nom : personne ne l'aime, mais sans lui, pas de Peter Pan, pas de Neverland ! Moralité, on s'en sert beacoup pour pimenter les codes ! On peut utiliser les crochets pour accéder aux éléments des matrices/arrays et dataframe/listes.

- Matrices et arrays

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

```
## [1] 0.05500996
```

```
mat [1,]
```

```
## [1]  0.05500996  1.25411791  0.59550676 -1.65628079  1.73888419
```

```
mat [,1]
```

```
##  [1]  0.05500996 -0.71677459  1.23079463  0.85939861  0.06496963
##  [6] -1.58952322  1.27184406 -2.23095991  0.31194736  0.20492429
```

```
arr [1,1,1]
```

```
## [1] 0.4389046
```

```
arr [1,,]
```

```
##             [,1]        [,2]       [,3]
## [1,]  0.43890465  1.25559729  1.1358748
## [2,]  0.44432399 -0.90918793  0.9269265
## [3,] -1.58331269 -0.01515283  1.0327507
## [4,] -1.13813628 -1.53970600 -0.1597102
## [5,]  0.05884532  0.24480520 -0.8985356
```

```
arr [,,1]
```

```
##              [,1]        [,2]        [,3]        [,4]         [,5]
##  [1,]  0.43890465  0.44432399 -1.58331269 -1.13813628  0.058845317
##  [2,] -0.46493797 -0.74429698 -1.07445978  0.45103349 -0.871139700
##  [3,] -0.46418448  1.00192443  0.05210394  0.07843470 -1.178293529
##  [4,]  1.38085887 -0.39282568  0.85057754  1.92707690  0.093523409
##  [5,] -0.90529717 -0.92073612  0.82767741 -0.02031069 -0.344907516
##  [6,] -0.57696846 -0.49390419 -0.15725088 -1.39630054  1.141103387
##  [7,]  0.01700932 -0.29639450  1.34429357  1.59649564  0.152110261
##  [8,]  1.25588043  0.15390705  0.41056938 -1.87823656  0.008117944
##  [9,]  0.69456192 -0.35266530 -0.85908530  1.86436401  2.253453568
## [10,] -0.16509592  0.03991952 -0.19138475 -1.02721045  0.682623439
```

- dataframes et listes :

Pour les dataframes, le fonctionnement est le même que pour les matrices. Pour les listes, une paire de crochet renvoie un résultat sous forme de liste, un double crochet renvoie le résultat sous sa forme naturelle (ex : vecteur ou matrice).

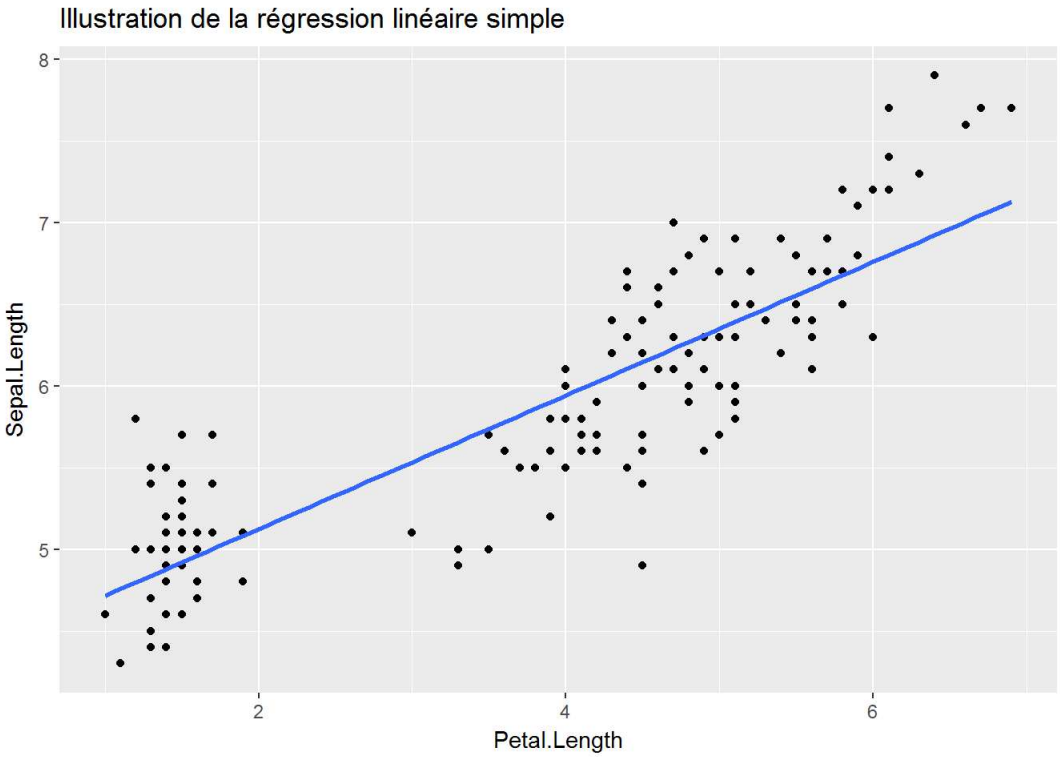
```
str (liste [1])
```

```
## List of 1
##  $ vtaille: Factor w/ 2 levels "L","S": 2 1 2 1 2 1 2 1 2 1
```

```
str (liste [[1]])
```

```
##  Factor w/ 2 levels "L","S": 2 1 2 1 2 1 2 1 2 1
```

9.6.4 Inspection d'un objet : la régression



La régression linéaire consiste à exprimer une variable Y en fonction d'une variable X dans une fonction linéaire. C'est à dire qu'on cherche a et b tels que : \[Y = a \cdot X + b + \epsilon\] où \(\epsilon\) est le résidu de la régression. On utilise dans cet exemple la table des iris de Fisher, existant dans R base qu'il suffit d'appeler avec `data(iris)` (il existe d'autres dataframe inclus dans les packages et qui sont utilisés en exemple dans l'aide).

```
data ("iris")
str (iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Faire la régression de la Sepal.Length sur Petal.length à l'aide de la fonction `lm()`

```
lm (data = iris, formula = Sepal.Length ~ Petal.Length)
```

```
##
## Call:
## lm(formula = Sepal.Length ~ Petal.Length, data = iris)
##
## Coefficients:
##  (Intercept)  Petal.Length
##       4.3066        0.4089
```

On a les paramètres a et b mais on aimerait en savoir plus… Au moins la qualité d'ajustement (le \(R^2\) par exemple), et un graphique des résidus pour détecter une éventuelle structure. Pour cela, stocker le résultat dans un nouvel objet, et explorez-le avec les fonctions `str()` , `summary()` et `plot()`

```
reg <- lm(data = iris, formula = Sepal.Length ~ Petal.Length)
str (reg)
```

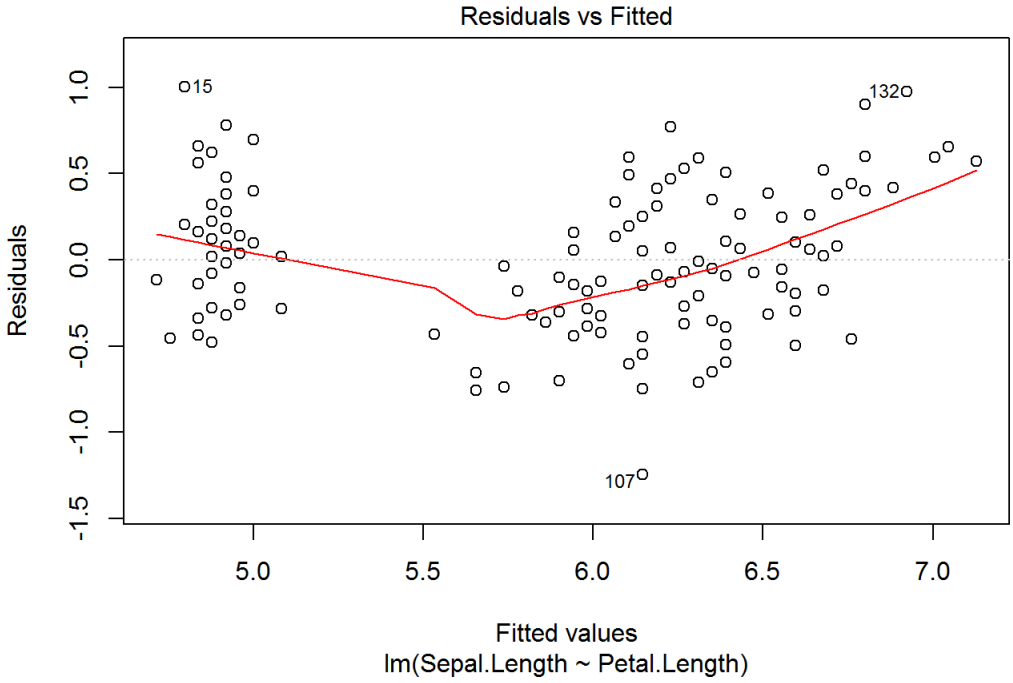
```
## List of 12
##  $ coefficients : Named num [1:2] 4.307 0.409
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "Petal.Length"
##  $ residuals    : Named num [1:150] 0.2209 0.0209 -0.1382 -0.32 0.1209 ...
##   ..- attr(*, "names")= chr [1:150] "1" "2" "3" "4" ...
##  $ effects      : Named num [1:150] -71.566 8.812 -0.155 -0.337 0.104 ...
##   ..- attr(*, "names")= chr [1:150] "(Intercept)" "Petal.Length" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:150] 4.88 4.88 4.84 4.92 4.88 ...
##   ..- attr(*, "names")= chr [1:150] "1" "2" "3" "4" ...
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..$ qr   : num [1:150, 1:2] -12.2474 0.0816 0.0816 0.0816 0.0816 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:150] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:2] "(Intercept)" "Petal.Length"
##   .. ..- attr(*, "assign")= int [1:2] 0 1
##   ..$ qraux: num [1:2] 1.08 1.1
##   ..$ pivot: int [1:2] 1 2
##   ..$ tol  : num 1e-07
##   ..$ rank : int 2
##   ..- attr(*, "class")= chr "qr"
##  $ df.residual  : int 148
##  $ xlevels      : Named list()
##  $ call         : language lm(formula = Sepal.Length ~ Petal.Length, data = iris)
##  $ terms        :Classes 'terms', 'formula'  language Sepal.Length ~ Petal.Length
##   .. ..- attr(*, "variables")= language list(Sepal.Length, Petal.Length)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "Sepal.Length" "Petal.Length"
##   .. .. .. ..$ : chr "Petal.Length"
##   .. ..- attr(*, "term.labels")= chr "Petal.Length"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(Sepal.Length, Petal.Length)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. ..- attr(*, "names")= chr [1:2] "Sepal.Length" "Petal.Length"
##  $ model        :'data.frame':   150 obs. of  2 variables:
##   ..$ Sepal.Length: num [1:150] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##   ..$ Petal.Length: num [1:150] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##   ..- attr(*, "terms")=Classes 'terms', 'formula'  language Sepal.Length ~ Petal.Length
##   .. .. ..- attr(*, "variables")= language list(Sepal.Length, Petal.Length)
##   .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. .. ..- attr(*, "dimnames")=List of 2
```

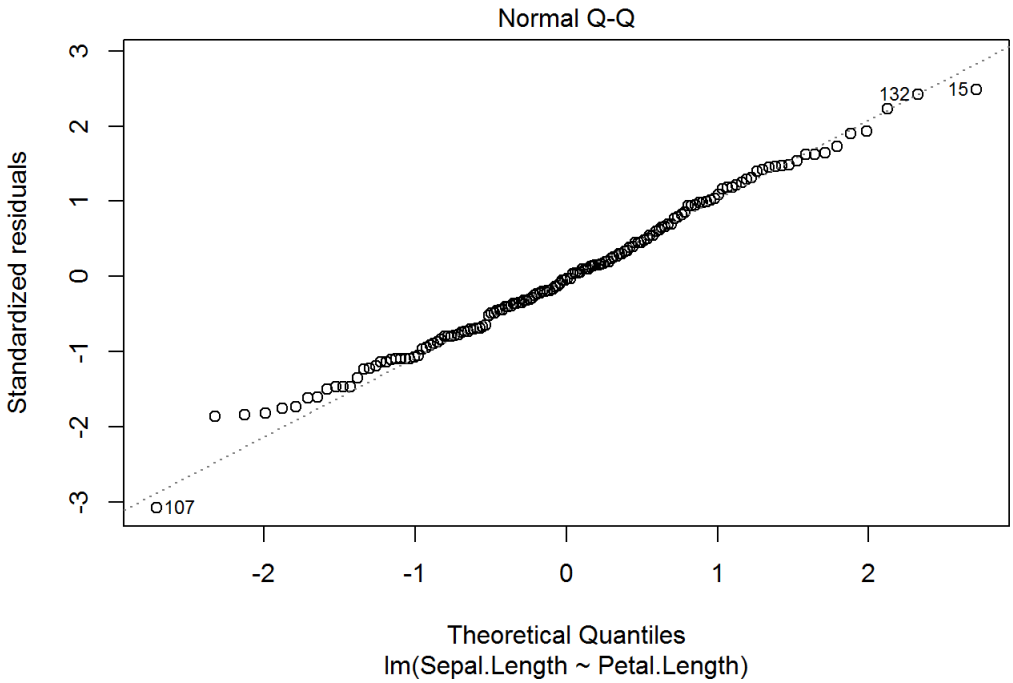
```
##   .. .. .. .. ..$ : chr [1:2] "Sepal.Length" "Petal.Length"
##   .. .. .. .. ..$ : chr "Petal.Length"
##   .. .. ..- attr(*, "term.labels")= chr "Petal.Length"
##   .. .. ..- attr(*, "order")= int 1
##   .. .. ..- attr(*, "intercept")= int 1
##   .. .. ..- attr(*, "response")= int 1
##   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. .. ..- attr(*, "predvars")= language list(Sepal.Length, Petal.Length)
##   .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. .. ..- attr(*, "names")= chr [1:2] "Sepal.Length" "Petal.Length"
##  - attr(*, "class")= chr "lm"
```

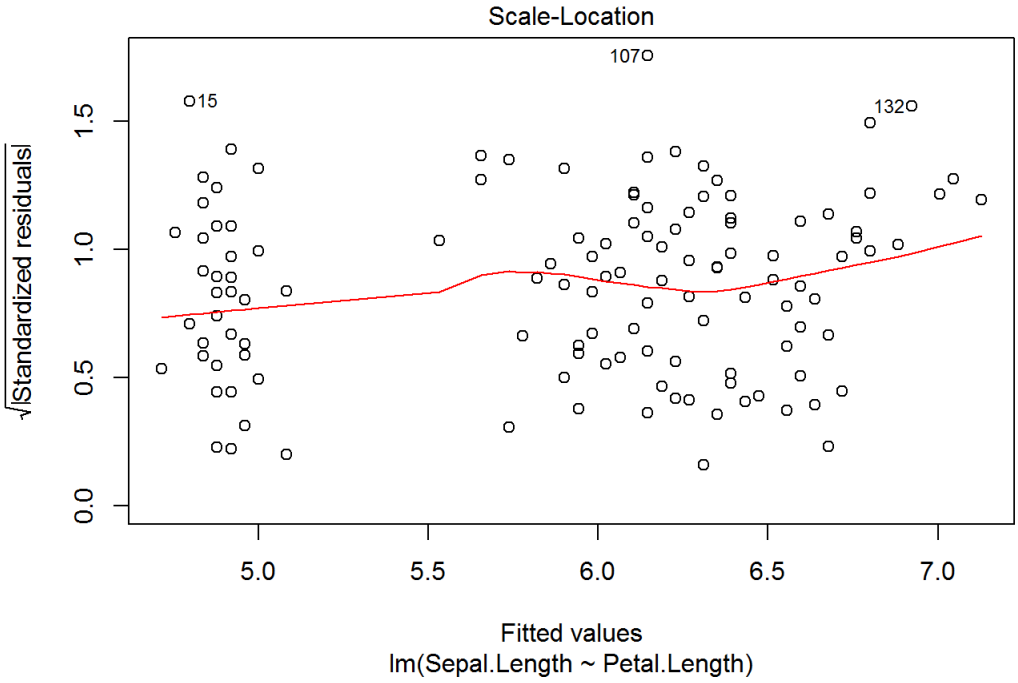
```
summary (reg)
```

```
##
## Call:
## lm(formula = Sepal.Length ~ Petal.Length, data = iris)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.24675 -0.29657 -0.01515  0.27676  1.00269
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   4.30660    0.07839   54.94   <2e-16 ***
## Petal.Length  0.40892    0.01889   21.65   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4071 on 148 degrees of freedom
## Multiple R-squared:   0.76,  Adjusted R-squared:  0.7583
## F-statistic: 468.6 on 1 and 148 DF,  p-value: < 2.2e-16
```

```
plot (reg)
```


Residuals vs Fitted
Residuals
Fitted values
lm(Sepal.Length ~ Petal.Length)
15
132
107
1.0
0.5
0.0
-0.5
-1.0
-1.5
5.0
5.5
6.0
6.5
7.0


Normal Q-Q
Standardized residuals
Theoretical Quantiles
lm(Sepal.Length ~ Petal.Length)
132
15
107
3
2
1
0
-1
-2
-3
-2
-1
0
1
2



Les **méthodes** summary, print et plot sont implémentées pour tous les objets en R, et on peut les utiliser pour avoir un premier aperçu de ce que l'on obtient avec la fonction.