Calculs de base en R

Sophie Baillargeon, Université Laval

2020-01-20

Table des matières

Fonctionnement vectoriel et règle de recyclage **2**

Fonctions et opérateurs pour des calculs mathématiques **6**
Opérateurs mathématiques . . . 6
Opérateurs arithmétiques . . . 6
Opérateurs de comparaison . . . 6
Opérateurs et fonction logiques vectoriels . . . 8
Fonctions mathématiques agissant de façon vectorielle . . . 9
Fonctions mathématiques combinant des éléments . . . 10
Fonctions d'opérations sur des ensembles . . . 11
Mots-clés mathématiques . . . 11

Fonctions pour le calcul de statistiques descriptives **12**
Fonctions retournant une seule statistique . . . 12
Traitement des observations manquantes et argument `na.rm` . . . 13
Fonctions pouvant retourner plusieurs statistiques . . . 14
Fonctions retournant un vecteur de statistiques . . . 16
Fonctions de calcul de fréquences . . . 17
Fonctions pour l'énumération de combinaisons . . . 21
Fonctions pour le traitement des observations dupliquées . . . 22

Fonctions de la famille des `apply` **23**
Fonction `apply` . . . 23
Fonctions raccourcies : `rowSums`, `colSums`, `rowMeans` et `colMeans` . . . 24
Fonctions `lapply`, `sapply` et `mapply` . . . 25
Fonctions `tapply`, `by` et `aggregate` . . . 27
Autres fonctions pour réaliser des calculs par niveaux de facteurs . . . 29
Choix de la fonction de la famille des `apply` à utiliser . . . 30

Conditions logiques **31**
Conditions logiques vectorielles de longueur quelconque . . . 31
Opérateur `%in%` de comparaison à un ensemble de valeurs . . . 32
Fonctions de comparaison pour caractères spéciaux . . . 33
Conditions logiques de longueur 1 . . . 33
Opérateurs et fonctions logiques non vectoriels . . . 33
Fonctions `all` et `any` . . . 33
Fonctions de vérification de type . . . 34

Comparaison de deux objets R **34**

Résumé **36**

R est un environnement spécialisé dans les calculs statistiques. Voyons comment réaliser de tels calculs en R, en se limitant pour l'instant à des calculs simples. Des fonctionnalités de R permettant de réaliser des calculs plus avancés (ex. réaliser des tests statistiques, ajuster des modèles, générer des observations aléatoires, faire de l'algèbre linéaire, etc.) seront vues dans un autre cours. Je présente ici des fonctionnalités utiles pour :

- implanter une formule mathématique ;
- effectuer une transformation mathématique de variables dans un jeu de données ;
- calculer des statistiques descriptives, par exemple dans le cadre d'une analyse exploratoire de données.

Fonctionnement vectoriel et règle de recyclage

Tous les opérateurs et plusieurs des fonctions qui sont présentés dans cette fiche agissent de façon vectorielle. Ils effectuent un traitement élément par élément sur le ou les objets reçus en entrée.

Par exemple, si les deux matrices suivantes sont additionnées avec l'opérateur +,

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

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

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

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

l'élément en position (i,j) dans la première matrice sera additionné à l'élément à la même position dans la deuxième matrice, et ce, pour toutes les positions. Le résultat de cette addition terme à terme est donc le suivant :

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

```
##      [,1] [,2] [,3]
## [1,]    7    7    7
## [2,]    7    7    7
```

Si les deux objets intervenant dans l'opération ne sont pas de mêmes dimensions, la **règle de recyclage** s'applique. Cette règle avait déjà été mentionnée dans les notes sur les structures de données en R. Étant donné son importance, revoyons-là plus en profondeur ici.

```
x <- c(5, 6)
y <- c(2, 5, 3, 1)
x + y
```

```
## [1]  7 11  8  7
```

L'instruction précédente effectue 4 additions, une pour chacun des 4 éléments du plus long des deux vecteurs dans l'opération, soit ici le deuxième. Le premier vecteur est plutôt de longueur 2. R répète donc ses éléments pour créer un vecteur aussi long que le deuxième

```
rep(x, times = length(y)/length(x))
```

```
## [1] 5 6 5 6
```

et effectue en réalité l'opération suivante :

```
c(5, 6, 5, 6) + c(2, 5, 3, 1)
```

```
## [1]  7 11  8  7
```

Cette règle de recyclage est exploitée, souvent sans que l'utilisateur en soit pleinement conscient, lorsque l'un des deux vecteurs impliqués dans une opération est de longueur 1. Par exemple, la commande suivante impliquant un exposant,

```
y ^ 2
```

```
## [1]  4 25  9  1
```

est en fait traduite par R en la commande suivante :

```
y ^ rep(2, times = length(y))
```

```
## [1]  4 25  9  1
```

Règle de recyclage avec des objets à plus d'une dimension

La règle de recyclage s'applique aussi dans des opérations faisant intervenir des objets à plus d'une dimension. Par exemple, pour additionner le même vecteur, disons

```
y <- 3:1
y
```

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

à chacune des colonnes d'une matrice, disons

```
mat <- matrix(1:12 , nrow = 3, ncol = 4)
mat
```

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

il suffit de lancer la commande suivante

```
mat + y
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    4    7   10   13
## [2,]    4    7   10   13
## [3,]    4    7   10   13
```

au lieu de la suivante, qui retourne exactement le même résultat.

```
mat + matrix(rep(y, times = length(mat)/length(y)), nrow = nrow(mat), ncol = ncol(mat))
```

Dans cette dernière commande, les deux arguments fournis à l'opérateur `+` sont réellement de mêmes dimensions, car la deuxième matrice est la suivante

```
matrix(rep(y, length(mat)/length(y)), nrow = nrow(mat), ncol = ncol(mat))
```

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

Une règle de recyclage utilisée pour former une matrice de dimension appropriée va donc remplir la matrice une colonne à la fois, comme le fait la fonction `matrix` par défaut.

Règle de recyclage lorsque la longueur de l'objet le plus long n'est pas multiple de la longueur de l'objet le plus court

Lorsque la longueur de l'objet le plus long n'est pas multiple de la longueur de l'objet le plus court, la règle de recyclage fonctionne quand même. R recycle l'objet le plus court assez de fois pour arriver à un objet de longueur égale ou supérieure à l'objet le plus long. Ensuite, si l'objet recyclé est plus long que l'autre objet, il est tronqué de façon à ce que les deux objets aient la même longueur.

Prenons par exemple les deux vecteurs suivants :

```
x <- 1:12
x
```

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

```
y <- 5:1
y
```

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

Supposons que la commande suivante soit soumise en R.

```
x + y
```

L'objet de gauche dans l'addition est de longueur 12 et l'objet de droite de longueur 5. L'objet de droite sera donc recyclé 3 fois,

```
y_recycle <- rep(5:1, times = ceiling(length(x)/length(y)))
y_recycle
```

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

puis sa longueur sera réduite à la longueur de l'objet de gauche.

```
length(y_recycle) <- length(x)
y_recycle
```

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

Ensuite l'addition terme à terme sera effectuée.

```
x + y_recycle
```

```
##  [1]  6  6  6  6  6 11 11 11 11 11 16 16
```

Cependant, R émettra un avertissement pour nous informer qu'il a dû faire cet ajustement de longueur.

```
x + y
```

```
## Warning in x + y: longer object length is not a multiple of shorter object
## length
```

```
##  [1]  6  6  6  6  6 11 11 11 11 11 16 16
```

Fonctions et opérateurs pour des calculs mathématiques

Opérateurs mathématiques

Opérateurs arithmétiques

Voici une liste d'opérateurs arithmétiques disponibles en R :

- `+` : addition,
- `-` : soustraction,
- `*` : multiplication,
- `/` : division,
- `^` : puissance,
- `%/%` : division entière,
- `%%` : modulo = reste de la division entière.

Les premiers opérateurs sont usuels et ne requièrent aucune explication. Expliquons cependant brièvement les deux derniers opérateurs de cette liste.

Division entière et modulo

L'opérateur `%/%` réalise une division entière. Pour illustrer ce type de division, prenons l'exemple suivant.

```
5 / 2
```

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

L'opérateur de division ordinaire `/` retourne un nombre réel. L'opérateur `%/%` retourne la partie entière du résultat obtenu avec `/`. La partie décimale est tronquée.

```
5 %/% 2
```

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

L'opérateur modulo `%%` retourne le reste de la division entière. Dans l'exemple traité ici, ce reste vaut 1 car 5 - 2*2 = 1.

```
5 %% 2
```

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

Astuces :

- Cet opérateur est pratique pour tester si des nombres sont pairs ou impairs. Les nombres pairs sont des multiples de 2. Alors `x %% 2` retourne 0 pour les nombres pairs et 1 pour les nombres impairs.
- L'opérateur modulo peut aussi servir à tester si un nombre stocké sous le type `double` est en réalité un entier au sens mathématique. S'il s'agit d'un entier, `x %% 1` retournera 0.

Opérateurs de comparaison

Les opérateurs de comparaison permettent de comparer des valeurs. Ils retournent `TRUE` ou `FALSE`. Il s'agit des opérateurs suivants :

- `==` : égalité,
- `!=` : non-égalité,
- `>` : plus grand,
- `>=` : plus grand ou égal,
- `<` : plus petit,
- `<=` : plus petit ou égal.

Supposons `x` et `y` les deux vecteurs numériques suivants.

```
x <- c(2, 5, 7, 3)
y <- c(3, 5, 6, 4)
```

Comparons ces vecteurs à l'aide d'un opérateur de comparaison. Est-ce que les valeurs contenues dans `x` sont supérieures aux valeurs contenues dans `y` ?

```
x > y
```

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

L'opérateur fonctionne de façon vectorielle, donc une comparaison est effectuée pour toutes les paires d'éléments à la même position dans les vecteurs `x` et `y`. Les valeurs dans le résultat retourné sont de type logique.

Les valeurs dans un vecteur peuvent aussi être comparées à une seule valeur, auquel cas la règle de recyclage s'applique.

```
x != 5
```

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

Comparaison de valeurs non numériques

Les opérateurs de comparaison ne fonctionnent pas seulement avec des valeurs numériques. Ils peuvent aussi être utilisés pour comparer des valeurs logiques ou caractères. Dans ce cas, il faut savoir que R considère que `FALSE` est inférieure à `TRUE`.

```
FALSE < TRUE
```

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

Quant aux caractères, les opérateurs de comparaison utilisent l'ordre de classement des caractères pour déterminer, entre deux valeurs, celle qui est inférieure. Cet ordre dépend des paramètres régionaux de la session R. D'une langue à l'autre, cet ordre peut varier.

Pour connaître l'ordre utilisé dans une session R, les instructions suivantes sont utiles :

```
caracteres_speciaux <-
  c("!", "\"", "#", "$", "%", "&", "'", "(", ")", "*", "+", ",", "-", ".", "/", ":", ";",
    "<", "=",">", "?", "@", "[", "\\", "]", "^", "_", "{", "|", "}", "~")
lettres_accentuees <- c("à", "â", "é", "è", "ê", "ë", "ï", "î", "ô", "ù", "ü", "û", "ç")
catacteres_ordonnes <- sort(c(caracteres_speciaux, 0:9, letters, LETTERS,
                              lettres_accentuees, toupper(lettres_accentuees)))
paste(catacteres_ordonnes, collapse = "")
```

J'ai obtenu le résultat suivant, qui sera peut-être différent sur votre ordinateur si vous n'avez pas les mêmes paramètres régionaux que moi.

```
"'-!\"#$%&()*,./:;?@[\\]^_{|}~+<=>0123456789aAàÀâÂbBcCçÇdDeEéÉèÈêÊëËfFgGhHiIîÎïÏjJkKlLmM
nNoOôÔpPqQrRsStTuUùÙûÛüÜvVwWxXyYzZ"
```

Ainsi, dans ma session R :

- les caractères spéciaux sont inférieurs aux chiffres et aux lettres,
- les chiffres sont inférieurs aux lettres,
- les lettres sont classées en ordre alphabétique et
  - les lettres minuscules sont inférieures aux lettres majuscules,
  - les lettres non accentuées sont inférieures aux lettres accentuées.

Pour des chaînes à plus d'un caractère, la comparaison s'effectue caractère par caractère (premiers caractères comparés entre eux, puis deuxièmes en cas d'égalité, puis troisièmes en cas d'égalités aux deux premières positions, etc.).

```
"arborescence" < "arbre"
```

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

Aussi, l'absence de caractères vaut moins que la présence.

```
"a" < "aa"
```

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

Remarque : Afin de correctement ordonner des nombres, il faut s'assurer de les stocker sous un format numérique. S'ils sont stockés sous forme de chaînes de caractères, les résultats obtenus ne seront pas toujours ceux attendus, comme dans cet exemple pour lequel 2 est dit non inférieur à 10 lorsque les nombres sont fournis à l'opérateur de comparaison sous forme de chaînes de caractères.

```
2 < 10
```

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

```
"2" < "10"
```

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

Opérateurs et fonction logiques vectoriels

Un opérateur ou une fonction logique vectoriel prend en entrée un ou deux vecteurs de logiques et retourne un autre vecteur de valeurs logiques. Le R de base comporte les opérateurs et la fonction logiques vectoriels suivants :

- `!` : négation,
- `&` : et,
- `|` : ou,
- `xor` : ou exclusif.

Opérateur de négation `!`

L'opérateur `!` n'a qu'un seul argument, alors que les autres opérateurs logiques en ont deux. Il effectue une négation, donc transforme les `TRUE` en `FALSE` et les `FALSE` en `TRUE`.

```
! c(TRUE, FALSE)
```

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

Opérateurs `&` et `|`, fonction `xor`

Les opérateurs `&` et `|`, ainsi que la fonction `xor`, appliquent de façon vectorielle les tables de vérité des fonctions mathématiques logiques « et », « ou » et « ou exclusif » respectivement.

Rappel : table de vérité de « et », « ou » et « ou exclusif »

```
p <- rep(c(FALSE, TRUE), each = 2)
q <- rep(c(FALSE, TRUE), times = 2)
cbind(p, q, "p et q" = p & q, "p ou q" = p | q, "p xor q" = xor(p, q))
```

```
##          p     q p et q p ou q p xor q
## [1,] FALSE FALSE  FALSE  FALSE   FALSE
## [2,] FALSE  TRUE  FALSE   TRUE    TRUE
## [3,]  TRUE FALSE  FALSE   TRUE    TRUE
## [4,]  TRUE  TRUE   TRUE   TRUE   FALSE
```

Ainsi,

- l'instruction `x & y` retournera un vecteur contenant des `TRUE` aux positions pour lesquelles la valeur en `x` et la valeur en `y` sont toutes les deux `TRUE` et contenant des `FALSE` partout ailleurs ;
- l'instruction `x | y` retournera un vecteur contenant des `FALSE` aux positions pour lesquelles la valeur en `x` et la valeur en `y` sont toutes les deux `FALSE` et contenant des `TRUE` partout ailleurs ;
- l'instruction `xor(x, y)` retournera un vecteur contenant des `TRUE` aux positions pour lesquelles la valeur en `x` ou la valeur en `y` est `TRUE`, mais pas les deux, et contenant des `FALSE` partout ailleurs.

Fonctions mathématiques agissant de façon vectorielle

R offre aussi plusieurs fonctions de calculs mathématiques, travaillant de façon vectorielle, dont les suivantes :

- racine carrée : `sqrt` ;
- exponentielles et logarithmes : `exp`, `log` (= logarithme naturel), `log10`, `log2` ;
- fonctions trigonométriques : `sin`, `cos`, `tan`, `acos`, `asin`, `atan`, `atan2` ;
- fonctions relatives au signe : `abs`, `sign` ;
- fonctions d'arrondissement : `ceiling`, `floor`, `round`, `trunc`, `signif` ;
- fonctions reliées aux fonctions mathématiques bêta et gamma : `beta`, `gamma`, `factorial`, `choose`, etc.

Ces fonctions font un calcul distinct pour tous les éléments de l'objet fourni en entrée et retournent un résultat de même dimension que l'objet en entrée. Voici quelques exemples.

```
# Vecteur de données numériques pour les exemples
x <- seq(from = -1.25, to = 1.5, by = 0.25)
x
```

```
##  [1] -1.25 -1.00 -0.75 -0.50 -0.25  0.00  0.25  0.50  0.75  1.00  1.25  1.50
```

```
# Arrondissement régulier au dixième près
round(x, digits = 1)
```

```
##  [1] -1.2 -1.0 -0.8 -0.5 -0.2  0.0  0.2  0.5  0.8  1.0  1.2  1.5
```

```
# Arrondissement à l'entier supérieur
ceiling(x)
```

```
##  [1] -1 -1  0  0  0  0  1  1  1  1  2  2
```

```
# Arrondissement à la partie entière
trunc(x)
```

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

Ces fonctions arrivent aussi à effectuer des calculs par élément dans un objet atomique de dimension supérieure à un ou dans un data frame.

```
# Matrice de données numériques pour les exemples
x_mat <- matrix(x, nrow = 2)
x_mat
```

```
##       [,1]  [,2]  [,3] [,4] [,5] [,6]
## [1,] -1.25 -0.75 -0.25 0.25 0.75 1.25
## [2,] -1.00 -0.50  0.00 0.50 1.00 1.50
```

```
# Extraction du signe
sign(x_mat)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   -1   -1   -1    1    1    1
## [2,]   -1   -1    0    1    1    1
```

Fonctions mathématiques combinant des éléments

Certaines fonctions mathématiques en R effectuent des calculs faisant intervenir plus d'un élément de l'objet donné en entrée, plutôt que d'effectuer un calcul distinct pour chacun des éléments. C'est le cas des fonctions suivantes :

- somme ou produit de tous les éléments (retourne une seule valeur) : `sum`, `prod` ;
- somme ou produit cumulatif des éléments (retourne un vecteur de même longueur que le vecteur en entrée) : `cummsum`, `cumprod` ;
- différences entre des éléments : `diff`.

Voici quelques exemples.

```
# Matrice de données numériques pour les exemples
mat <- matrix(c(2, 5, 3, 4, 6, 5, 4, 3, 1, 2, 9, 8), nrow = 3, ncol = 4)
mat
```

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

```
# Produit de tous les éléments
prod(mat)
```

```
## [1] 6220800
```

```
# Somme cumulative des éléments (ici 2, 2+5, 2+5+3, 2+5+3+4, ...)
cumsum(mat)
```

```
##  [1]  2  7 10 14 20 25 29 32 33 35 44 52
```

Fonction `diff`

Pour une matrice ou un data frame, `diff` calcule les différences terme à terme des éléments composant les lignes. Par défaut, la fonction calcule pour chaque ligne, à l'exception de la première, la différence entre la ligne et la ligne au-dessous.

```
diff(mat)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    3    2   -1    7
## [2,]   -2   -1   -2   -1
```

La commande suivante retourne donc le même résultat que la précédente.

```
mat[-1, ] - mat[-nrow(mat), ]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    3    2   -1    7
## [2,]   -2   -1   -2   -1
```

Pour un vecteur, la fonction `diff` retourne les différences entre un élément (sauf le premier) et l'élément précédent.

```
diff(c(2, 5, 3, 4))
```

```
## [1]  3 -2  1
```

La fonction `diff` peut calculer des différences entre les éléments séparés par plus d'une position grâce à l'argument `lag`, comme dans cet exemple.

```r
diff(c(2, 5, 3, 4), lag = 2)
```

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

```r
# soustractions effectuées : 3-2 et 4-5
```

Fonctions d'opérations sur des ensembles

Les fonctions R d'opérations sur des ensembles sont les suivantes :

- `union` : union,
- `intersect` : intersection,
- `setdiff` : différence,
- `setequal` : test d'égalité,
- `is.element` : test d'inclusion.

Voici quelques exemples utilisant les deux ensembles suivants, stockés sous forme de vecteur :

```r
A <- c("m", "s", "e", "f", "m")
B <- c("m", "e", "h", "i")
```

Union de tous les éléments des ensembles `A` et `B`, en retirant les doublons :

```r
union(A, B)
```

```
## [1] "m" "s" "e" "f" "h" "i"
```

Identification des éléments communs entre `A` et `B`, en retirant les doublons :

```r
intersect(A, B)
```

```
## [1] "m" "e"
```

Identification des éléments de `A` ne se retrouvant pas dans `B`, en retirant les doublons :

```r
setdiff(A, B)
```

```
## [1] "s" "f"
```

Test sur l'égalité entre les ensembles `A` et `B` :

```r
setequal(A, B)
```

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

Test sur la présence de `"d"` et `"e"` dans l'ensemble `A` :

```r
is.element(el = c("d", "e"), set = A)
```

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

Mots-clés mathématiques

En R, le nombre π est représenté par le mot-clé `pi`.

```r
pi
```

```
## [1] 3.141593
```

`Inf` est le symbole R pour l'infini ∞.

```r
-5 / 0
```

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

`NaN` est un mot-clé signifiant *Not A Number*. Ce mot-clé est retourné par R lorsqu'un utilisateur lui demande d'effectuer une opération mathématique impossible, par exemple :

```
log(-1)
```

```
## Warning in log(-1): NaNs produced
```

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

Rappel : Attention à ne pas confondre le mot-clé `NaN` avec le mot-clé `NA` qui signifie plutôt *Not Available* et qui sert à représenter les données manquantes.

Fonctions pour le calcul de statistiques descriptives

Fonctions retournant une seule statistique

Certaines fonctions de calcul de statistiques descriptives retournent en sortie une seule valeur. C'est le cas des fonctions suivantes :

- mesures de position : `min`, `max` ;
- mesures de tendance centrale : `mean`, `median` ;
- mesure de dispersion : `sd` (écart-type).

Utilisons le jeu de données `cars` du package `datasets` pour présenter quelques exemples. Ce jeu de données contient 50 observations de 2 variables numériques.

```
str(cars)
```

```
## 'data.frame':    50 obs. of  2 variables:
##  $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
##  $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```

```
# Moyenne des observations de la variable dist
mean(cars$dist)
```

```
## [1] 42.98
```

Si l'objet en entrée a plus d'une dimension, la sortie est tout de même de longueur 1. Donc tous les éléments contenus dans l'objet sont mis en commun pour faire le calcul.

```
max(cars)
```

```
## [1] 120
```

Fonctions `which.max` et `which.min`

Les fonctions `min` et `max` retournent respectivement la valeur la plus petite et la valeur la plus grande parmi les éléments d'un objet. Les fonctions `which.max` et `which.min` retournent pour leur part la position dans l'objet du **premier** maximum ou minimum.

```
which.min(cars$speed)
```

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

Dans l'exemple précédent, il y a en fait deux observations qui prennent la valeur minimum de `min(cars$speed)`. La commande suivante permet de trouver la position de toutes les observations prenant la valeur minimale.

```
which(cars$speed == min(cars$speed))
```

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

Traitement des observations manquantes et argument `na.rm`

Les fonctions `min`, `max`, `mean`, `median` et `sd`, ainsi que quelques autres fonctions vues dans ces notes, ont un argument en commun nommé `na.rm`. Cet argument sert à indiquer à la fonction comment agir en présence de données manquantes (`NA`). Par défaut, `na.rm` prend la valeur `FALSE` pour ces fonctions. Cette valeur signifie que les données manquantes ne doivent pas être retirées avant d'effectuer le calcul. Cependant, en présence de données manquantes, ces fonctions ne sont pas en mesure de calculer des statistiques. Par exemple, supposons que nous voulions calculer la médiane des données dans le vecteur suivant.

```
x <- c(3, 6, NA, 8, 11, 15, 23)
```

Si nous ne retirons pas la donnée manquante, nous obtenons le résultat suivant.

```
median(x)
```

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

Ce résultat s'explique par le fait que la valeur de la médiane dépend de toutes les observations, incluant l'observation manquante, qui est inconnue. La valeur de la médiane est donc elle aussi inconnue. Pour calculer plutôt la médiane des observations non manquantes, il faut donner la valeur `TRUE` à l'argument `na.rm` comme suit.

```
median(x, na.rm = TRUE)
```

```
## [1] 9.5
```

Notons que la fonction `na.omit` permet de retirer les observations manquantes d'un objet R. Si l'objet est un vecteur, les éléments contenant `NA` sont retirés.

```
na.omit(x)
```

```
## [1]  3  6  8 11 15 23
## attr(,"na.action")
## [1] 3
## attr(,"class")
## [1] "omit"
```

La fonction `na.omit` ajoute deux attributs à l'objet, dont un pour identifier les observations retirées.

Remarquons que les deux commandes suivantes retournent le même résultat.

```
median(x, na.rm = TRUE)
```

```
## [1] 9.5
```

```
median(na.omit(x))
```

```
## [1] 9.5
```

Si la fonction `na.omit` reçoit en entrée une matrice ou un data frame, elle retire toutes les lignes contenant au moins un `NA`, comme dans cet exemple :

```
exJeu <- data.frame(x, y = c(2, NA, 8, 9, 6, NA, 2));
exJeu
```

```
##    x  y
## 1  3  2
## 2  6 NA
## 3 NA  8
## 4  8  9
## 5 11  6
## 6 15 NA
## 7 23  2
```

```r
na.omit(exJeu)
```

```
##    x y
## 1  3 2
## 4  8 9
## 5 11 6
## 7 23 2
```

Fonctions pouvant retourner plusieurs statistiques

D'autres fonctions peuvent retourner plus d'une statistique, notamment les fonctions suivantes :

- mesures de position : `range`, `quantile` ;
- résumé comprenant plusieurs mesures : `summary` ;
- variances, covariances et corrélations : `var`, `cov`, `cor`.

Fonctions `range` et `quantile`

La fonction `range` retourne à la fois le minimum est le maximum, comme dans cet exemple :

```r
range(cars$speed)
```

```
## [1]  4 25
```

Une façon simple d'obtenir l'étendue d'observations à partir de la sortie de la fonction `range` est de procéder comme suit :

```r
diff(range(cars$speed))
```

```
## [1] 21
```

La fonction `quantile` calcule des quantiles empiriques. Par défaut, elle retourne le minimum, le maximum et les quartiles, comme dans cet exemple :

```r
quantile(cars$speed)
```

```
##   0%  25%  50%  75% 100%
##    4   12   15   19   25
```

L'argument `probs` permet de demander n'importe quels quantiles. Dans l'exemple suivant, les premiers et neuvièmes déciles sont demandés.

```r
quantile(cars$speed, probs = c(0.1, 0.9))
```

```
##  10%  90%
##  8.9 23.1
```

Remarque : Il existe plusieurs façons de calculer des quantiles. La fonction `quantile` implémente 9 algorithmes de calcul de quantiles (voir `help(quantile)`).

Fonction `summary`

La fonction `summary` retourne les statistiques suivantes selon l'entrée qu'elle reçoit :

- vecteur numérique : minimum, premier quartile, médiane, moyenne, troisième quartile, maximum ;
- facteur : fréquences des modalités (comme la fonction `table` vue plus loin) ;
- matrice ou data frame : la fonction `summary` est appliquée séparément à chacune des colonnes.

Utilisons le jeu de données `Puromycin` du package `datasets` pour présenter quelques exemples d'utilisation de la fonction `summary`. Ce jeu de données contient 23 observations de 3 variables, dont deux numériques et une catégorique, stockée sous forme de facteur.

```r
str(Puromycin)
```

```
## 'data.frame':    23 obs. of  3 variables:
##  $ conc : num  0.02 0.02 0.06 0.06 0.11 0.11 0.22 0.22 0.56 0.56 ...
##  $ rate : num  76 47 97 107 123 139 159 152 191 201 ...
##  $ state: Factor w/ 2 levels "treated","untreated": 1 1 1 1 1 1 1 1 1 1 ...
##  - attr(*, "reference")= chr "A1.3, p. 269"
```

```r
# Vecteur numérique en entrée :
summary(Puromycin$rate)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    47.0    91.5   124.0   126.8   158.5   207.0
```

```r
# Facteur en entrée :
summary(Puromycin$state)
```

```
##   treated untreated
##        12        11
```

```r
# Data frame entier en entrée :
summary(Puromycin)
```

```
##       conc             rate             state
##  Min.   :0.0200   Min.   : 47.0   treated  :12
##  1st Qu.:0.0600   1st Qu.: 91.5   untreated:11
##  Median :0.1100   Median :124.0
##  Mean   :0.3122   Mean   :126.8
##  3rd Qu.:0.5600   3rd Qu.:158.5
##  Max.   :1.1000   Max.   :207.0
```

Fonctions var, cov et cor

La fonction `var` peut prendre en entrée un vecteur ou un objet à deux dimensions. Si elle reçoit en entrée un vecteur, elle calcule la variance empirique de toutes les valeurs, comme dans cet exemple :

```r
var(cars$speed)
```

```
## [1] 27.95918
```

Cependant, si elle reçoit en entrée une matrice ou un data frame de valeurs numériques, elle considère que chaque colonne contient les observations d'une variable aléatoire. Elle va calculer une matrice de variances-covariances, comme dans cet exemple :

```r
var(cars)
```

```
##           speed     dist
## speed  27.95918 109.9469
## dist  109.94694 664.0608
```

La fonction `cov` fait exactement le même calcul par défaut.

```r
cov(cars)
```

```
##           speed     dist
## speed  27.95918 109.9469
## dist  109.94694 664.0608
```

Elle peut cependant calculer des covariances de Kendall ou de Spearman (toutes deux des statistiques non paramétriques basées sur les rangs des observations) au lieu de la covariance classique de Pearson. La fonction

`cor` calcule des corrélations plutôt que des covariances. Elle aussi peut utiliser les définitions de Pearson (par défaut), Kendall et Spearman. Voici un exemple de calcul de matrice de corrélations de Spearman.

```
cor(cars, method = "spearman")
```

```
##           speed      dist
## speed 1.0000000 0.8303568
## dist  0.8303568 1.0000000
```

Fonctions retournant un vecteur de statistiques

Certaines fonctions, telles que les suivantes, retournent autant de statistiques qu'il y a d'éléments dans l'objet donné en entrée.

- mesures de position : `cummin`, `cummax`, `pmin`, `pmax` ;
- rangs : `rank`.

Fonctions `cummin` et `cummax`

Les fonctions `cummin` et `cummax` calculent les minimums et les maximums cumulatifs. Comme nous pouvons le constater dans l'exemple suivant, la valeur en position `i` du vecteur retourné par une de ces deux fonctions est la valeur minimale ou maximale dans le sous-vecteur `x[1:i]`.

```
cummin(x = c(-2, 4, -3, 4, 7, -6, 0))
```

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

Fonctions `pmin` et `pmax`

Les fonctions `pmin` et `pmax` calculent le minimum et le maximum par position, entre autant de vecteurs que désiré, comme dans l'exemple suivant.

```
pmax(c(-2, 4, -3, 4, 7, -6, 0),
     c( 1, 2,  3, 4, 5,  6, 7),
     c( 5, 0, -2, 4, 5,  3, 3))
```

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

Ces fonctions sont utiles pour remplacer des valeurs par un seuil. Par exemple, l'instruction suivante permet de remplacer par zéro toute valeur négative contenue dans le vecteur en entrée.

```
pmax(c(-2, 4, -3, 4, 7, -6, 0), 0)
```

```
## [1] 0 4 0 4 7 0 0
```

Fonction `rank`

Certains tests statistiques non paramétriques utilisent des statistiques basées sur les rangs des observations. Voici un exemple d'obtention de ces rangs avec la fonction `rank`.

```
rank(c(-2, 4, -3, 4, 7, -6, 0))
```

```
## [1] 3.0 5.5 2.0 5.5 7.0 1.0 4.0
```

Par défaut, en cas d'égalité, le rang moyen est utilisé. Pour changer cette option, il faut modifier la valeur de l'argument `ties.method`. Dans l'exemple suivant, le rang minimum est retourné en cas d'égalité.

```
rank(c(-2, 4, -3, 4, 7, -6, 0), ties.method = "min")
```

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

Fonctions de calcul de fréquences

Les fonctions `table`, `xtabs` et `ftable` permettent de calculer des fréquences.

Voici un petit jeu de données pour illustrer l'utilisation de ces fonctions. Il contient des observations concernant 7 individus fictifs : la couleur de leurs yeux, la couleur de leurs cheveux et leur genre.

```
sondage <- data.frame(
  yeux    = c("brun",    "brun",     "bleu",    "brun",    "vert",     "brun",    "bleu"),
  cheveux = c("brun",    "noir",     "blond",   "brun",    "brun",     "blond",   "brun"),
  genre   = c("féminin", "masculin", "féminin", "féminin", "masculin", "féminin", "masculin")
)
sondage
```

```
##   yeux cheveux    genre
## 1 brun    brun  féminin
## 2 brun    noir masculin
## 3 bleu   blond  féminin
## 4 brun    brun  féminin
## 5 vert    brun masculin
## 6 brun   blond  féminin
## 7 bleu    brun masculin
```

Fonctions `table`

La fonction `table` permet de compter le nombre d'occurrences de chacune des modalités d'une variable catégorique dans des données. Demandons, par exemple, à `table` de compter le nombre d'individus dans les données `sondage` classés dans chacune des catégories de couleurs de cheveux.

```
table(sondage$cheveux)
```

```
##
## blond  brun  noir
##     2     4     1
```

La fonction `table` produit un tableau de fréquences à une variable si elle reçoit les observations d'une seule variable. Elle peut aussi produire des tableaux de fréquences croisées à deux variables ou plus.

```
# Exemple de tableau de fréquences à deux variables (avec variables nommées)
table(yeux = sondage$yeux, cheveux = sondage$cheveux)
```

```
##       cheveux
## yeux   blond brun noir
##   bleu     1    1    0
##   brun     1    2    1
##   vert     0    1    0
```

```
# Exemple de tableau de fréquences à trois variables (data frame en entrée à table)
t3 <- table(sondage)
t3
```

```
## , , genre = féminin
##
##       cheveux
## yeux   blond brun noir
##   bleu     1    0    0
##   brun     1    2    0
##   vert     0    0    0
##
```

```
## , , genre = masculin
##
##       cheveux
## yeux   blond brun noir
##   bleu     0    1    0
##   brun     0    0    1
##   vert     0    1    0
```

Fonctions `ftable`

La fonction `ftable` retourne un tableau de fréquences sous la forme d'une table « plate » (en anglais *flat*, d'où le `f` dans le nom de la fonction) dans le cas d'un croisement de 3 variables ou plus, plutôt que sous la forme d'un array comme le fait la fonction `table`. Elle accepte les mêmes types d'entrées que `table` (série d'objets atomiques à une dimension ou objet récursif dont les éléments sont interprétables en facteurs) et peut aussi recevoir une sortie de la fonction `table`, comme dans l'exemple suivant.

```
ftable(t3)
```

```
##              genre féminin masculin
## yeux cheveux
## bleu blond               1        0
##      brun                0        1
##      noir                0        0
## brun blond               1        0
##      brun                2        0
##      noir                0        1
## vert blond               0        0
##      brun                0        1
##      noir                0        0
```

Fonctions `xtabs`

La fonction `xtabs` fait le même calcul que les fonctions précédentes, mais elle prend en entrée une formule. Le tableau de fréquences à deux variables créé précédemment peut être réobtenu de la façon suivante avec `xtabs`.

```
xtabs(~ yeux + cheveux, data = sondage)
```

```
##       cheveux
## yeux   blond brun noir
##   bleu     1    1    0
##   brun     1    2    1
##   vert     0    1    0
```

La fonction `xtabs` est utile lorsque les données que nous avons en main contiennent déjà des fréquences, car il est possible d'inclure une variable réponse contenant des dénombrements dans la formule que nous lui fournissons en entrée. Par exemple, `xtabs` permet de facilement retrouver le tableau de fréquences marginales croisées entre les variables `yeux` et `cheveux` à partir du tableau de fréquences à trois variables produit précédemment mis sous forme de data frame, qui a l'allure suivante.

```
t3_df <- as.data.frame(t3)
t3_df
```

```
##    yeux cheveux    genre Freq
## 1  bleu   blond  féminin    1
## 2  brun   blond  féminin    1
## 3  vert   blond  féminin    0
## 4  bleu    brun  féminin    0
```

```
## 5  brun    brun  féminin    2
## 6  vert    brun  féminin    0
## 7  bleu    noir  féminin    0
## 8  brun    noir  féminin    0
## 9  vert    noir  féminin    0
## 10 bleu   blond masculin    0
## 11 brun   blond masculin    0
## 12 vert   blond masculin    0
## 13 bleu    brun masculin    1
## 14 brun    brun masculin    0
## 15 vert    brun masculin    1
## 16 bleu    noir masculin    0
## 17 brun    noir masculin    1
## 18 vert    noir masculin    0
```

Il suffit de procéder comme suit :

```
t2 <- xtabs(Freq ~ yeux + cheveux, data = t3_df)
t2
```

```
##       cheveux
## yeux   blond brun noir
##   bleu     1    1    0
##   brun     1    2    1
##   vert     0    1    0
```

Autres fonctions relatives au calcul de fréquences

Les fonctions `margin.table`, `addmargins` et `prop.table` permettent de calculer des fréquences marginales ou relatives à partir d'un tableau de fréquences. Voici quelques exemples d'utilisation de ces fonctions exploitant le tableau de fréquences à deux variables produit ci-dessus.

```
# Fréquences marginales en colonnes :
margin.table(t2, margin = 2)
```

```
## cheveux
## blond  brun  noir
##     2     4     1
```

```
# Fréquences marginales ajoutées au tableau :
addmargins(t2)
```

```
##       cheveux
## yeux   blond brun noir Sum
##   bleu     1    1    0   2
##   brun     1    2    1   4
##   vert     0    1    0   1
##   Sum      2    4    1   7
```

```
# Fréquences relatives croisées :
prop.table(t2)
```

```
##       cheveux
## yeux       blond      brun      noir
##   bleu 0.1428571 0.1428571 0.0000000
##   brun 0.1428571 0.2857143 0.1428571
##   vert 0.0000000 0.1428571 0.0000000
```

```r
# Fréquences relatives conditionnelles à la variable yeux :
prop.table(t2, margin = 1)
```

```
##       cheveux
## yeux   blond brun noir
##   bleu  0.50 0.50 0.00
##   brun  0.25 0.50 0.25
##   vert  0.00 1.00 0.00
```

Transformation du format d'un objet de classe "table"

Les fonctions `table` et `xtabs` attribuent à l'objet qu'ils retournent en sortie la classe `"table"`, comme nous pouvons le constater en observant l'objet `t2`.

```r
attributes(t2)
```

```
## $dim
## [1] 3 3
##
## $dimnames
## $dimnames$yeux
## [1] "bleu" "brun" "vert"
##
## $dimnames$cheveux
## [1] "blond" "brun"  "noir"
##
##
## $class
## [1] "xtabs" "table"
##
## $call
## xtabs(formula = Freq ~ yeux + cheveux, data = t3_df)
```

```r
str(t2)
```

```
##  'xtabs' int [1:3, 1:3] 1 1 0 1 2 1 0 1 0
##  - attr(*, "dimnames")=List of 2
##   ..$ yeux   : chr [1:3] "bleu" "brun" "vert"
##   ..$ cheveux: chr [1:3] "blond" "brun" "noir"
##  - attr(*, "call")= language xtabs(formula = Freq ~ yeux + cheveux, data = t3_df)
```

Il est parfois utile de transformer un objet de classe `"table"` en un array (matrice si la table croise deux variables) ou un data frame. Pour la transformation en array, il suffit de retirer l'attribut `class` avec la fonction `unclass`, comme dans cet exemple :

```r
str(unclass(t2))
```

```
##  int [1:3, 1:3] 1 1 0 1 2 1 0 1 0
##  - attr(*, "dimnames")=List of 2
##   ..$ yeux   : chr [1:3] "bleu" "brun" "vert"
##   ..$ cheveux: chr [1:3] "blond" "brun" "noir"
##  - attr(*, "call")= language xtabs(formula = Freq ~ yeux + cheveux, data = t3_df)
```

Comme nous avons pu le constater dans un exemple précédent, la transformation en data frame crée pour sa part un jeu de données contenant une ligne par combinaison distincte des niveaux des facteurs croisés dans la table. Le data frame obtenu comporte une colonne par facteur, ainsi qu'une colonne nommée `Freq` contenant les fréquences dans la table. En voici un exemple :

```
str(as.data.frame(t2))
```

```
## 'data.frame':    9 obs. of  3 variables:
##  $ yeux   : Factor w/ 3 levels "bleu","brun",..: 1 2 3 1 2 3 1 2 3
##  $ cheveux: Factor w/ 3 levels "blond","brun",..: 1 1 1 2 2 2 3 3 3
##  $ Freq   : int  1 1 0 1 2 1 0 1 0
```

Fonctions pour l'énumération de combinaisons

Un fonction utile pour énumérer toutes les combinaisons des niveaux d'un facteur est `expand.grid`. Par exemple, retrouvons avec cette fonction toutes les combinaisons présentes dans le data frame `t3_df` créé précédemment.

```
expand.grid(yeux = c("bleu", "brun", "vert"),
            cheveux = c("blond", "brun", "noir"),
            genre = c("féminin", "masculin"))
```

```
##    yeux cheveux    genre
## 1  bleu   blond  féminin
## 2  brun   blond  féminin
## 3  vert   blond  féminin
## 4  bleu    brun  féminin
## 5  brun    brun  féminin
## 6  vert    brun  féminin
## 7  bleu    noir  féminin
## 8  brun    noir  féminin
## 9  vert    noir  féminin
## 10 bleu   blond masculin
## 11 brun   blond masculin
## 12 vert   blond masculin
## 13 bleu    brun masculin
## 14 brun    brun masculin
## 15 vert    brun masculin
## 16 bleu    noir masculin
## 17 brun    noir masculin
## 18 vert    noir masculin
```

Il faut fournir en entrée à `expand.grid` les valeurs à combiner. Si les vecteurs ou facteurs contenant ces valeurs sont fournis avec des noms, comme dans l'exemple précédent (via les assignations), les colonnes du data frame retourné en sortie par `expand.grid` porteront ces noms.

Une autre fonction R permet d'énumérer des combinaisons possibles : la fonction `combn`. Il s'agit de combinaisons au sens mathématique cette fois, donc « de dispositions non ordonnées d'un certain nombre d'éléments d'un ensemble »[1]. Par exemple, voici toutes les combinaisons possibles de 3 éléments parmi l'ensemble `c("Ève", "Jean", "Mia", "Paul")`, trouvées par la fonction `combn`.

```
combn(x = c("Ève", "Jean", "Mia", "Paul"), m = 3)
```

```
##      [,1]   [,2]   [,3]   [,4]
## [1,] "Ève"  "Ève"  "Ève"  "Jean"
## [2,] "Jean" "Jean" "Mia"  "Mia"
## [3,] "Mia"  "Paul" "Paul" "Paul"
```

Contrairement à `expand.grid` qui présente les combinaisons possibles ligne par ligne, chaque colonne de la sortie produite par `combn` représente une combinaison possible.

[1] Cette définition est tirée du site web suivant : http://www.alloprof.qc.ca/BV/pages/m1346.aspx

Notons finalement que la fonction `choose` mentionnée précédemment permet de compter le nombre de combinaisons possibles de `k` éléments d'un ensemble de taille `n`. Elle calcule donc le coefficient binomial $\binom{n}{k}$.

```
# Nombre de combinaisons possibles dans l'exemple précédent :
choose(n = 4, k = 3)
```

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

Fonctions pour le traitement des observations dupliquées

Il est parfois utile de gérer les observations dupliquées dans un jeu de données. En R, les fonctions suivantes sont utiles avec des observations dupliquées :

- pour tester la présence d'observations dupliquées : `duplicated`,
- pour retirer les observations dupliquées : `unique`.

Une observation est ici définie par l'ensemble des valeurs observées de toutes les variables pour un individu (ou une unité) de la population statistique à l'étude. Donc une observation dupliquée est une ligne répétée (donc deux lignes ou plus complètement identiques) dans une matrice ou un data frame. Dans le cas d'une seule variable, stockée dans un vecteur, une observation dupliquée est une valeur présente plus d'une fois dans le vecteur.

Pour illustrer l'emploi des fonctions `duplicated` et `unique`, réutilisons le jeu de données `sondage`. Ce jeu de données contient une observation dupliquée.

```
sondage
```

```
##   yeux cheveux    genre
## 1 brun    brun  féminin
## 2 brun    noir masculin
## 3 bleu   blond  féminin
## 4 brun    brun  féminin
## 5 vert    brun masculin
## 6 brun   blond  féminin
## 7 bleu    brun masculin
```

En effet, les lignes 1 et 4 sont identiques. La fonction `duplicated` identifie la 4e observation comme une duplication d'une autre observation.

```
duplicated(sondage)
```

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

L'observation peut être retirée avec la fonction `unique` comme suit.

```
unique(sondage)
```

```
##   yeux cheveux    genre
## 1 brun    brun  féminin
## 2 brun    noir masculin
## 3 bleu   blond  féminin
## 5 vert    brun masculin
## 6 brun   blond  féminin
## 7 bleu    brun masculin
```

Si elles reçoivent un vecteur en entrée, les fonctions `duplicated` et `unique` réagissent comme suit.

```
duplicated(c(1, 3, 2, 1, 2, 1))
```

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

```r
unique(c(1, 3, 2, 1, 2, 1))
```

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

Fonctions de la famille des `apply`

R propose plusieurs fonctions, dites « de la famille des `apply` », qui ont pour but d'appliquer itérativement une autre fonction sur des sous-sections d'un objet. Les grandes étapes d'un traitement effectué par une de ces fonctions sont les suivantes :

- séparer un objet en sous-objets ;
- répéter la même action pour tous les sous-objets : appeler une fonction en lui donnant comme premier argument le sous-objet ;
- combiner les résultats obtenus.

Ces fonctions cachent en fait des boucles. Les fonctions de la famille des `apply` sont utiles pour :

- obtenir des statistiques marginales à partir d'une matrice ou d'un array,
- appliquer le même traitement à tous les éléments d'une liste,
- calculer des statistiques descriptives selon les niveaux de facteurs,
- effectuer des calculs en parallèle (nous y reviendrons plus tard),
- etc.

Nous verrons ici les fonctions : `apply`, `lapply`, `sapply`, `mapply`, `tapply`, `by` et `aggregate`.

Fonction `apply`

Si elle reçoit comme premier argument une matrice, la fonction `apply` appelle en boucle une fonction en lui donnant en entrée l'une après l'autre chacune des lignes ou des colonnes d'une matrice. Voici un exemple.

```r
mat <- matrix(1:12, nrow = 3, ncol = 4, byrow = TRUE)
mat[2,3] <- NA
mat
```

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

```r
# Calcul sur chaque ligne :
apply(mat, MARGIN = 1, FUN = mean)
```

```
## [1]  2.5   NA 10.5
```

```r
# Calcul sur chaque colonne :
apply(mat, MARGIN = 2, FUN = mean)
```

```
## [1]  5  6 NA  8
```

Pour ajouter un argument à envoyer à la fonction `FUN`, il suffit de l'ajouter à la liste des arguments fournis, préférablement en le nommant. C'est l'argument `...` qui permet ce transfert d'arguments entre une fonction principale et une fonction présente dans le corps de la fonction principale.

```r
apply(mat, MARGIN = 2, FUN = mean, na.rm = TRUE)
```

```
## [1] 5 6 7 8
```

La fonction retourne une liste si `FUN` retourne plus d'une valeur.

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

```
## [[1]]
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    1.75    2.50    2.50    3.25    4.00
##
## [[2]]
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   5.000   5.500   6.000   6.333   7.000   8.000       1
##
## [[3]]
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    9.00    9.75   10.50   10.50   11.25   12.00
```

De façon plus générale, la fonction `apply` peut itérer sur des sous-objets créés à partir d'un array à plus de deux dimensions.

```
arr <- array(1:12, dim = c(2, 3, 2))
arr
```

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

```
apply(arr, MARGIN = c(1, 2), FUN = sum)
```

```
##      [,1] [,2] [,3]
## [1,]    8   12   16
## [2,]   10   14   18
```

Si elle reçoit en entrée un data frame, elle le transformera en matrice avant d'effectuer les calculs.

Fonctions raccourcies : `rowSums`, `colSums`, `rowMeans` et `colMeans`

Pour le calcul de sommes et de moyennes par lignes ou colonnes d'une matrice, il existe des fonctions raccourcies à la fonction `apply` : `rowSums`, `colSums`, `rowMeans`, `colMeans`. Par exemple :

```
colMeans(mat, na.rm = TRUE)
```

```
## [1] 5 6 7 8
```

est équivalent à `apply(mat, MARGIN = 2, FUN = mean, na.rm = TRUE)` et

```
rowSums(mat, na.rm = TRUE)
```

```
## [1] 10 19 42
```

est équivalent à `apply(mat, MARGIN = 1, FUN = sum, na.rm = TRUE)`.

Ces fonctions spécialisées ont été optimisées en termes de temps d'exécution.

Fonctions `lapply`, `sapply` et `mapply`

Les fonctions `lapply`, `sapply` et `mapply` prennent en entrée un vecteur ou une liste (qui peut aussi être un data frame) et appliquent une fonction sur chaque élément de cet objet. Voici une liste qui sera utilisée pour illustrer l'emploi de ces fonctions. Cette liste contient les mots formant trois courtes phrases (ponctuation omise).

```
phrases <- list(
  phrase1 = c("regarde", "la", "belle", "neige"),
  phrase2 = c("allons", "skier"),
  phrase3 = c("non", "il", "fait", "trop", "froid")
)
```

Fonction `sapply` :

Supposons que nous voulons isoler le dernier mot de chaque phrase dans la liste `phrases`. L'action que nous souhaitons réaliser revient à extraire le dernier élément d'un vecteur. Elle doit être réalisée pour tous les vecteurs qui sont des éléments de la liste `phrases`. Nous pourrions réaliser cette tâche avec la commande suivante.

```
derniers_mots <- sapply(phrases, FUN = tail, n = 1)
derniers_mots
```

```
## phrase1 phrase2 phrase3
## "neige" "skier" "froid"
```

Décortiquons maintenant cette commande. L'instruction `sapply(phrases, FUN = tail, n = 1)` permet d'appliquer la fonction `tail` à chaque élément de la liste `phrases`. Ces éléments sont tous des vecteurs. L'argument `n = 1` est passé à la fonction `tail`. Ainsi, seul le dernier élément de chaque vecteur est extrait. C'est comme si nous avions soumis la commande

```
tail(phrases[[i]], n = 1)
```

séparément pour tous les éléments, donc pour `i` = 1, 2 et 3, puis que nous avions rassemblé les résultats.

Utilisation d'un opérateur comme valeur de l'argument `FUN` :

Si nous cherchions plutôt à isoler le deuxième mot de chaque phrase dans la liste `phrases`, nous pourrions réaliser cette extraction avec la commande suivante.

```
sapply(phrases, FUN = '[', 2)
```

```
## phrase1 phrase2 phrase3
##    "la" "skier"    "il"
```

Dans cet exemple, la fonction à appliquer est en fait l'opérateur d'extraction du crochet simple. Rappelons que les opérateurs sont en fait des fonctions. Donc, pour un vecteur quelconque, disons

```
x <- phrases[[1]]
```

les commandes suivantes sont équivalentes.

```
x[2]
```

```
## [1] "la"
```

```
'['(x, 2)
```

```
## [1] "la"
```

Ainsi, l'objet duquel nous voulons extraire est le premier argument à fournir à l'opérateur `[`. L'identifiant de l'élément à extraire (ici un entier représentant une position) est le deuxième argument à fournir à l'opérateur `[`. Si l'objet avait plus d'une dimension, il suffirait d'ajouter des arguments.

Lorsque l'argument `FUN` d'une fonction de la famille des `apply` est un opérateur, il faut toujours l'encadrer de guillemets (simples ou doubles).

Fonction `lapply` :

La fonction `lapply` fait exactement le même calcul que la fonction `sapply`, mais retourne le résultat sous la forme d'une liste plutôt que sous une forme simplifiée. Voici un appel à `lapply` équivalent à l'appel à `sapply` qui a permis de créer `derniers_mots`. Les valeurs en sortie n'ont pas changé, mais elles sont stockées dans une liste plutôt que dans un vecteur.

```
lapply(phrases, FUN = tail, n = 1)
```

```
## $phrase1
## [1] "neige"
##
## $phrase2
## [1] "skier"
##
## $phrase3
## [1] "froid"
```

Fonction `mapply` :

Il aurait aussi été possible de solutionner le problème de l'extraction des derniers mots des phrases en utilisant la fonction `mapply`. La différence entre cette fonction et les fonctions `sapply` et `lapply` est qu'elle peut fournir à la fonction `FUN` plusieurs (ou de multiples, d'où le `m` dans `mapply`) arguments qui sont des vecteurs ou des listes.

Par exemple, nous pourrions extraire les derniers mots en appliquant l'opérateur `[` à chaque élément de la liste `phrases`, mais en spécifiant comme argument pour l'opérateur d'extraction la position du dernier élément. Cette position diffère un peu d'un élément à l'autre. Elle est égale à la longueur de l'élément.

Nous pourrions donc, dans un premier temps, calculer la longueur de chaque élément de `phrases` comme suit :

```
longueurs_phrases <- sapply(phrases, length)
longueurs_phrases
```

```
## phrase1 phrase2 phrase3
##       4       2       5
```

Ayant en main un vecteur contenant les longueurs, nous pouvons utiliser `mapply` pour extraire les derniers éléments des vecteurs dans `phrases` par la commande suivante :

```
mapply(FUN = "[", phrases, longueurs_phrases)
```

```
## phrase1 phrase2 phrase3
## "neige" "skier" "froid"
```

La boucle cachée derrière cet appel à la fonction `mapply` est la suivante : pour `i` allant de 1 à 3, soit le nombre total d'éléments dans la liste `phrases`, l'extraction suivante est effectuée.

```
"["(phrases[[i]], longueurs_phrases[[i]])
```

Voici un autre exemple d'utilisation de la fonction `mapply`. Supposons que nous possédons trois listes contenant des vecteurs numériques, dont la longueur est la même selon la position, telles que les listes suivantes.

```
liste1 <- list(c(1, 2, 3, 4, 5), c(1, 2, 3))
liste2 <- list(c(3, 5, 4, 2, 3), c(3, 4, 2))
liste3 <- list(c(0, 3, 9, 8, 6), c(7, 5, 0))
```

Nous pourrions utiliser `mapply` pour former des matrices en concaténant en lignes tous les vecteurs à la même position dans les listes, comme suit :

```
mapply(FUN = rbind, liste1, liste2, liste3)
```

```
## [[1]]
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    3    5    4    2    3
## [3,]    0    3    9    8    6
##
## [[2]]
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    3    4    2
## [3,]    7    5    0
```

La fonction `mapply` est capable d'itérer sur les éléments d'un nombre indéterminé de vecteurs ou de listes.

Fonctions `tapply`, `by` et `aggregate`

Ces fonctions appliquent elles aussi la même fonction à plusieurs sous-objets. Ce qui les distingue des autres fonctions de la famille des `apply` est la formation des sous-objets, qui se réalise cette fois selon les niveaux de facteurs.

Nous allons reprendre le jeu de données `Puromycin` pour illustrer l'utilisation de ces fonctions.

```
str(Puromycin)
```

```
## 'data.frame':    23 obs. of  3 variables:
##  $ conc : num  0.02 0.02 0.06 0.06 0.11 0.11 0.22 0.22 0.56 0.56 ...
##  $ rate : num  76 47 97 107 123 139 159 152 191 201 ...
##  $ state: Factor w/ 2 levels "treated","untreated": 1 1 1 1 1 1 1 1 1 1 ...
##  - attr(*, "reference")= chr "A1.3, p. 269"
```

Fonction `tapply` :

Nous pourrions par exemple calculer la moyenne de la variable `rate` selon les niveaux du facteur `state` comme suit.

```
tapply(Puromycin$rate, INDEX = Puromycin$state, FUN = mean)
```

```
##   treated untreated
##  141.5833  110.7273
```

L'argument `INDEX` pourrait être une liste de plusieurs facteurs.

```
tapply(Puromycin$rate, INDEX = Puromycin[, c("conc", "state")], FUN = mean)
```

```
##       state
## conc   treated untreated
##   0.02    61.5      59.0
##   0.06   102.0      85.0
```

```
##   0.11   131.0     106.5
##   0.22   155.5     127.5
##   0.56   196.0     151.0
##   1.1    203.5     160.0
```

Dans l'exemple ci-dessous, nous avons fourni à `INDEX` un data frame, mais rappelons-nous que les data frames sont des cas particulier de listes. De plus, un élément de ce data frame n'est pas un facteur. Il s'agit de la variable `conc`. Cela n'a pas posé problème parce que `lapply` est arrivé à transformer l'élément en facteur.

Fonction `by` :

La fonction `by` prend comme objet en entrée un data frame et permet d'effectuer un calcul sur des sous-objets qui sont aussi des data frames. Par exemple, nous pourrions calculer la matrice de corrélations entre les observations des variables `conc` et `rate` selon les niveaux du facteur `state` comme suit.

```
by(Puromycin[, c("conc", "rate")], INDICES = Puromycin$state, FUN = cor)
```

```
## Puromycin$state: treated
##           conc      rate
## conc 1.0000000 0.8310362
## rate 0.8310362 1.0000000
## ------------------------------------------------------------
## Puromycin$state: untreated
##           conc      rate
## conc 1.0000000 0.8207311
## rate 0.8207311 1.0000000
```

Fonction `aggregate` :

Finalement, la fonction `aggregate` prend aussi en entrée un data frame, mais elle applique la fonction séparément pour chaque colonne du data frame.

```
aggregate(Puromycin[, c("conc", "rate")], by = list(state = Puromycin$state), FUN = mean)
```

```
##       state      conc     rate
## 1   treated 0.3450000 141.5833
## 2 untreated 0.2763636 110.7273
```

L'argument `by` doit obligatoirement être une liste. Nommer les éléments de la liste aide à clarifier la sortie.

La fonction `aggregate` accepte aussi des formules en entrée, comme dans les exemples ci-dessous.

```
# Exemple avec deux variables réponses et une variable explicative (de groupement)
aggregate(cbind(conc, rate) ~ state, data = Puromycin, FUN = mean)
```

```
##       state      conc     rate
## 1   treated 0.3450000 141.5833
## 2 untreated 0.2763636 110.7273
```

```
# Exemple avec une variable réponse et deux variables explicatives (de groupement)
aggregate(rate ~ conc + state, data = Puromycin, FUN = median)
```

```
##    conc     state  rate
## 1  0.02   treated  61.5
## 2  0.06   treated 102.0
## 3  0.11   treated 131.0
## 4  0.22   treated 155.5
## 5  0.56   treated 196.0
## 6  1.10   treated 203.5
```

```
## 7  0.02 untreated  59.0
## 8  0.06 untreated  85.0
## 9  0.11 untreated 106.5
## 10 0.22 untreated 127.5
## 11 0.56 untreated 151.0
## 12 1.10 untreated 160.0
```

Autres fonctions pour réaliser des calculs par niveaux de facteurs

Quelques packages R offrent d'autres fonctions permettant de réaliser des calculs par niveaux de facteurs. L'utilisation de deux de ces packages, souvent mentionnés par la communauté R, est illustrée ici en reproduisant les deux exemples précédents.

Package `dplyr`

L'utilisation conjointe des fonctions `group_by` et `summarize` du package `dplyr` du `tidyverse` permet d'agréger, en utilisant une statistique de notre choix, les observations de variables selon les niveaux de facteurs. En voici un exemple.

```
library(dplyr)
```

```
# Exemple avec deux variables réponses et une variable explicative (de groupement)
summarize(group_by(Puromycin, state), conc = mean(conc), rate = mean(rate))
```

```
## # A tibble: 2 x 3
##   state      conc  rate
##   <fct>     <dbl> <dbl>
## 1 treated   0.345  142.
## 2 untreated 0.276  111.
```

```
# Exemple avec une variable réponse et deux variables explicatives (de groupement)
summarize(group_by(Puromycin, conc, state), rate = median(rate))
```

```
## # A tibble: 12 x 3
## # Groups:   conc [6]
##     conc state      rate
##    <dbl> <fct>     <dbl>
##  1  0.02 treated    61.5
##  2  0.02 untreated  59
##  3  0.06 treated   102
##  4  0.06 untreated  85
##  5  0.11 treated   131
##  6  0.11 untreated 106.
##  7  0.22 treated   156.
##  8  0.22 untreated 128.
##  9  0.56 treated   196
## 10  0.56 untreated 151
## 11  1.1  treated   204.
## 12  1.1  untreated 160
```

Pour plus d'informations :

- https://dplyr.tidyverse.org/
- https://stt4230.rbind.io/tutoriels_etudiants/hiver_2016/agreger_donnees_dplyr/

Package `data.table`

Il est aussi possible de réaliser ces agrégations grâce à l'argument `by` de l'opérateur `[` du package `data.table`.

```
library(data.table)
Puromycin_dt <- data.table(Puromycin)
```

```
# Exemple avec deux variables réponses et une variable explicative (de groupement)
Puromycin_dt[, .(conc = mean(conc), rate = mean(rate)), by = state]
```

```
##        state      conc     rate
## 1:   treated 0.3450000 141.5833
## 2: untreated 0.2763636 110.7273
```

```
# Exemple avec une variable réponse et deux variables explicatives (de groupement)
Puromycin_dt[, .(rate = median(rate)), by = .(conc, state)]
```

```
##     conc     state  rate
##  1: 0.02   treated  61.5
##  2: 0.06   treated 102.0
##  3: 0.11   treated 131.0
##  4: 0.22   treated 155.5
##  5: 0.56   treated 196.0
##  6: 1.10   treated 203.5
##  7: 0.02 untreated  59.0
##  8: 0.06 untreated  85.0
##  9: 0.11 untreated 106.5
## 10: 0.22 untreated 127.5
## 11: 0.56 untreated 151.0
## 12: 1.10 untreated 160.0
```

Ce package offre l'avantage de pouvoir effectuer ces opérations rapidement sur de grands jeux de données.

Pour plus d'informations :

- https://rdatatable.gitlab.io/data.table/articles/datatable-intro.html#aggregations
- https://stt4230.rbind.io/tutoriels_etudiants/hiver_2017/data.table/

Choix de la fonction de la famille des `apply` à utiliser

Les fonctions de la famille des `apply` servent à appliquer un même calcul sur différentes parties (sous-objets) d'une structure de données R (objet principal).

La structure de données peut être brisée en sous-objets de différentes façons. Par exemple, s'il s'agit d'une matrice, elle peut être séparée en lignes ou en colonnes. S'il s'agit d'une liste, elle peut être séparée en éléments. Il est aussi possible de briser un vecteur ou un data frame en blocs d'observations référant à différents niveaux de facteurs. Dans ces notes, les fonctions de la famille des `apply` ont été séparées en 3 catégories selon la façon de former les sous-objets.

Le format de la sortie retournée varie aussi d'une fonction à l'autre.

Quand vient le temps de choisir une fonction de la famille des `apply` à utiliser, il faut donc se demander :

- Quel est le type de l'objet sur lequel appliquer les calculs ?
- Comment les sous-objets doivent-ils être formés ?
- Quel format de sortie est le plus approprié ?

Le tableau suivant permet de facilement comparer les fonctions de la famille des `apply` présentées en fournissant les réponses aux questions précédentes.

Fonction	Objet typique en entrée	Formation des sous-objets	Format de la sortie
`apply`	array (matrice)	selon une ou des dimensions	vecteur, array, liste
`lapply`	vecteur, liste (data frame)	éléments de l'objet en entrée	liste
`sapply`	vecteur, liste (data frame)	éléments de l'objet en entrée	simplifié par défaut
`mapply`	vecteurs, listes (data frames)	éléments des objets en entrée	simplifié par défaut
`tapply`	vecteur	selon les niveaux de facteurs	array ou liste
`by`	data frame	selon les niveaux de facteurs	array ou liste
`aggregate`	data frame	selon les niveaux de facteurs et par colonne du data frame	data frame

---

Conditions logiques

Une condition logique est simplement une instruction R qui retourne une ou des valeurs logiques (`TRUE` ou `FALSE`). Ce type d'instruction a différentes utilités, par exemple :

- explorer des données : répondre à des questions du genre combien d'observations respectent une certaine condition ;
- filtrer des données : extraire les observations respectant une certaine condition ;
- définir une condition dans une structure de contrôle conditionnelle `if ... else` ;
- etc.

Conditions logiques vectorielles de longueur quelconque

Les deux premières utilités potentielles des conditions logiques énumérées ci-dessus requièrent la création d'un vecteur de valeurs logiques de la même longueur que l'objet R sur lequel la condition est testée. Nous avons vu au début de cette fiche des outils pour écrire de telles conditions logiques :

- les opérateurs de comparaison : `==`, `!=`, `>`, `>=`, `<` et `<=` ;
- les opérateurs et fonctions logiques vectoriels : `!` (négation), `&` (et), `|` (ou) et `xor` (ou exclusif).

Voici des exemples d'écriture de conditions logiques utilisant le vecteur suivant, que nous avons déjà manipulé dans des notes précédentes.

```
de <- c(2, 3, 4, 1, 2, 3, 5, 6, 5, 4)
```

Supposons que nous voulions connaître le nombre d'éléments dans ce vecteur numérique dont la valeur est supérieure à 3. La condition logique suivante nous permet d'identifier ces valeurs.

```
condition <- de > 3
condition
```

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

Compter le nombre de valeurs supérieures à 3 dans `de` revient à compter le nombre de `TRUE` dans le vecteur précédent. Ce calcul se réalise facilement avec la fonction `sum` comme suit.

```
sum(condition)
```

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

Même si une somme est une opération mathématique sur des valeurs numériques, la commande précédente ne retourne par d'erreur, car R réalise d'abord une conversion implicite de type de données pour transformer les valeurs logiques en nombres (`TRUE` devient 1 et `FALSE` devient 0), puis effectue la somme.

Le vecteur `condition` serait aussi utile pour extraire les éléments de `de` ayant une valeur supérieure à 3. Nous savons que l'opérateur d'indiçage `[` et la fonction d'extraction `subset` acceptent en entrée un vecteur logique. Nous pouvons donc extraire les éléments respectant la condition comme suit.

```
de[condition]
```

```
## [1] 4 5 6 5 4
```

Fonction `which`

Une fonction parfois utile avec un vecteur logique est la fonction `which`, utilisée précédemment dans un exemple. Elle permet de connaître les positions des `TRUE` dans le vecteur, comme l'illustre cet exemple :

```
which(condition)
```

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

L'utilisation de `which` n'est cependant pas nécessaire lors de l'extraction d'éléments à partir d'un vecteur logique. Par exemple, les commandes `de[which(condition)]` et `de[condition]` produisent le même résultat, mais la commande sans appel à la fonction `which` a l'avantage d'être plus succincte.

Conditions combinant des vecteurs logiques

La condition précédente était plutôt simple. Une condition plus complexe requiert souvent de combiner des vecteurs logiques à l'aide d'un opérateur logique. Par exemple, l'instruction suivante identifie les éléments du vecteur `de` dont la valeur se situe dans l'intervalle $[3, 5]$.

```
de >= 3 & de <= 5
```

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

L'instruction suivante identifie pour sa part les éléments du vecteur `de` égaux à 1, 4 ou 6.

```
de == 1 | de == 4 | de == 6
```

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

Pour identifier les éléments du vecteur `de` non-égaux à 1, 4 ou 6, nous pourrions inverser le vecteur logique précédent avec l'opérateur de négation comme suit.

```
!(de == 1 | de == 4 | de == 6)
```

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

Rappelons qu'en logique mathématique, la négation d'une disjonction est équivalente à la conjonction de négations. L'instruction suivante retourne donc le même résultat que la précédente.

```
de != 1 & de != 4 & de != 6
```

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

Opérateur `%in%` de comparaison à un ensemble de valeurs

Pour effectuer une comparaison à un ensemble de valeur, telle que le fait l'instruction `de == 1 | de == 4 | de == 6`, R offre un opérateur raccourcissant la syntaxe : l'opérateur `%in%`. Cet opérateur compare les éléments d'un vecteur (placé avant l'opérateur) aux éléments d'un ensemble présenté sous la forme d'un vecteur (placé après). Il retourne `TRUE` pour un élément égal à n'importe lequel des éléments de l'ensemble, `FALSE` sinon. L'instruction `de == 1 | de == 4 | de == 6` est donc équivalent à la suivante.

```
de %in% c(1, 4, 6)
```

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

Combiné à un opérateur de négation `!`, l'opérateur `%in%` permet de facilement tester si les valeurs dans un vecteur sont différentes des valeurs d'un ensemble, comme dans cet exemple.

```
! de %in% c(1, 4, 6)
```

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

Fonctions de comparaison pour caractères spéciaux

Notons que tester si un ou des éléments sont égaux à `NA`, `NaN` ou `Inf` (constante pour l'infini), ne se fait pas directement avec l'opérateur `==` comme suit.

```
c(1, 2, NA, 4, 5) == NA
```

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

Il faut plutôt utiliser la fonction `is.na`, `is.nan` ou `is.infinite`.

```
is.na(c(1, 2, NA, 4, 5))
```

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

Conditions logiques de longueur 1

Lors de l'écriture d'une condition logique, il faut parfois s'assurer de retourner un vecteur logique de longueur 1. C'est le cas lors de l'écriture d'une condition logique dans une structure de contrôle conditionnelle `if ... else` (que nous verrons plus loin). La condition dans un `if` doit être obligatoirement de longueur 1.

Opérateurs et fonctions logiques non vectoriels

Les opérateurs et fonctions logiques suivants garantissent que le résultat retourné est de longueur 1.

- `&&` : et,
- `||` : ou,
- `isTRUE` et `isFALSE`.

Les opérateurs `&&` et `||` appliquent les mêmes tables de vérité que les opérateurs `&` et `|`, mais ils ne travaillent pas de façon vectorielle. Si, par inadvertance, `&&` ou `||` reçoit en entrée des vecteurs de longueurs supérieures à 1, il effectue une opération seulement sur les premiers éléments de ces vecteurs, comme dans cet exemple.

```
de == 1 || de == 4 || de == 6
```

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

Les fonctions `isTRUE` et `isFALSE`, pour leur part, sont des fonctions raccourcies permettant d'effectuer les tests suivants.

```
is.logical(x) && length(x) == 1 && !is.na(x) && x   # isTRUE
is.logical(x) && length(x) == 1 && !is.na(x) && !x  # isFALSE
```

Elles permettent donc de s'assurer qu'une condition possède toutes les caractéristiques requises pour être fournie à un `if` (contenir des données logiques, être de longueur 1 et ne pas prendre la valeur `NA`).

Fonctions `all` et `any`

Les fonctions `all` et `any` font partie des fonctions R retournant toujours une seule valeur logique. La fonction `all` indique si tous les éléments d'un vecteur logique sont `TRUE`. Par exemple, pour tester si toutes les valeurs dans le vecteur `de` sont entières au sens mathématique, nous pourrions utiliser la commande suivante.

```
all(de %% 1 == 0)
```

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

La fonction `any` indique pour sa part si au moins un élément d'un vecteur logique est `TRUE`. Nous pourrions par exemple vérifier si le vecteur `de` comporte des valeurs négatives comme suit.

```
any(de < 0)
```

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

Fonctions de vérification de type

Finalement, les fonctions `is.numeric`, `is.character`, `is.logical`, `is.vector`, `is.matrix`, `is.data.frame`, `is.factor`, `is.null`, `is.function`, etc., testent une condition et retournent toujours un logique de longueur unitaire. Par exemple, testons si le vecteur `de` contient bien des données numériques.

```
is.numeric(de)
```

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

Comparaison de deux objets R

Les opérateurs de comparaison permettent de comparer les éléments d'objets R. Mais comment comparer des objets entiers ? Cela dépend de ce qui doit être comparé.

- Pour comparer tous les éléments, mais pas les attributs : `all(x == y)`
  - retourne `TRUE` si tous les éléments sont égaux,
  - `FALSE` sinon,
  - `NA` si un des deux objets comparés contient au moins une valeur manquante et que l'argument `na.rm` de la fonction `all` prend la valeur `FALSE`.
- Pour comparer les objets dans leur totalité (éléments, attributs, type de l'objet et de ses éléments) : `identical(x, y)`
  - retourne `TRUE` si les deux objets comparés sont totalement identiques,
  - `FALSE` sinon.
- Pour comparer tous les éléments et les attributs, en acceptant des différences dans les valeurs numériques selon une certaine tolérance : `all.equal(x, y)`
  - retourne `TRUE` en cas d'égalité respectant la tolérance,
  - sinon retourne des informations sur les différences.

Voici quelques exemples.

Éléments identiques, mais attributs différents

```
# Objets comparés
x <- 1:5
y <- 1:5
names(x) <- letters[1:5]
str(x)
```

```
##  Named int [1:5] 1 2 3 4 5
##  - attr(*, "names")= chr [1:5] "a" "b" "c" "d" ...
```

```
str(y)
```

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

```r
# Résultats des différentes comparaisons
all(x == y)
```

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

```r
identical(x, y)
```

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

```r
all.equal(x, y)
```

```
## [1] "names for target but not for current"
```

Éléments équivalents, mais de types différents, attributs identiques

```r
# Objets comparés
x <- as.double(x)
str(x)
```

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

```r
str(y)
```

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

```r
# Résultats des différentes comparaisons
all(x == y)
```

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

```r
identical(x, y)
```

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

```r
all.equal(x, y)
```

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

Éléments numériques pas tout à fait identiques, attributs et types identiques

```r
# Objets comparés
y <- 1:5 + 1e-10
str(x)
```

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

```r
str(y)
```

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

Bien que les valeurs numériques dans `x` et `y` ne soient pas tout à fait identiques, elles semblent identiques à l'affichage de `x` et `y`.

```r
# Résultats des différentes comparaisons
all(x == y)
```

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

```r
identical(x, y)
```

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

```r
all.equal(x, y)
```

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

Rappel : Il est possible de contrôler le nombre de chiffres composant un nombre affichés dans la console R avec l'option `digits` de la session R.

```r
optionsDefaut <- options()
options(digits = 11)
y
```

```
## [1] 1.0000000001 2.0000000001 3.0000000001 4.0000000001 5.0000000001
```

```r
options(digits = optionsDefaut$digits)
```

Résumé

Fonctions et opérateurs mathématiques et statistiques de base en R

- fonctionnement vectoriel et règle de recyclage : calculs élément par élément pour un objet, ou encore terme à terme entre des objets ;

Calcul	opère de façon vectorielle	combine, retourne une valeur	combine, retourne valeur(s)	combine, retourne un vecteur
arithmétique	`+`, `-`, `*`, `/`, `^`, `%%`, `%/%`	`sum`, `prod`		`cumsum`, `cumprod`, `diff`
comparaison	`==`, `!=`, `>`, `>=`, `<`, `<=`			
logique	`!`, `&`, `|`, `xor`	`&&`, `||`		
racine carrée, exponentielle, logarithme, trigonométrie, signe, arrondissement, bêta, gamma	`sqrt`, `exp`, `log`, `log10`, `log2`, `sin`, `cos`, `tan`, `acos`, `asin`, `atan`, `atan2`, `abs`, `sign`, `ceiling`, `floor`, `round`, `trunc`, `signif`, `beta`, `gamma`, `factorial`, `choose`, etc.			
mesure de position	`pmin`, `pmax`	`min`, `max`, `which.min`, `which.max`	`range`, `quantile`, `summary`	`cummin`, `cummax`, `rank`
tendance centrale		`mean`, `median`	`summary`	
dispersion		`sd`	`var`, `cov`, `cor`	
fréquences			`table`, `ftable`, `xtabs`, `summary`	

- opérations sur des ensembles : `union`, `intersect`, `setdiff`, `setequal`, `is.element` ;
- mots-clés mathématiques : `pi`, `Inf`, `NaN`.
- argument `na.rm` : spécifie le comportement de la fonction en présence de valeurs manquantes ;
- calcul de fréquences marginales ou relatives à partir d'un tableau de fréquences : `margin.table`, `addmargins`, `prop.table`.
- énumération de combinaisons : `expand.grid`, `combn` ;
- traitement des observations dupliquées : `duplicated`, `unique`.

Fonctions de la famille des `apply`

Principe de base derrière ces fonctions (qui cachent des boucles) :

- séparer un objet en sous-objets ;
- appeler une fonction en lui donnant comme premier argument tous les sous-objets, un à la fois ;
- combiner les résultats obtenus.

Résumé du fonctionnement des fonctions présentées :

Fonction	Objet typique en entrée	Formation des sous-objets	Format de la sortie
`apply`	array (matrice)	selon une ou des dimensions	vecteur, array, liste
`lapply`	vecteur, liste (data frame)	éléments de l'objet en entrée	liste
`sapply`	vecteur, liste (data frame)	éléments de l'objet en entrée	simplifié par défaut
`mapply`	vecteurs, listes (data frames)	éléments des objets en entrée	simplifié par défaut
`tapply`	vecteur	selon les niveaux de facteurs	array ou liste
`by`	data frame	selon les niveaux de facteurs	array ou liste
`aggregate`	data frame	selon les niveaux de facteurs et par colonne du data frame	data frame

Description des arguments à donner en entrée à ces fonctions :

- 1^{e} argument (sauf pour `mapply`) : objet à séparer et sur lequel appliquer la fonction ;
- argument suivant : information pour spécifier comment séparer l'objet en sous-objets (sauf pour les fonctions prenant en entrée une liste, soit pour `lapply`, `sapply` et `mapply`, car dans ce cas les sous-objets sont les éléments de la liste) ;
- argument suivant (celui nommé `FUN`) : la fonction à appliquer (les sous-objets lui seront fournis comme premier argument) ;
- `...` : il est possible de passer des arguments supplémentaires à la fonction à appliquer (`FUN`) simplement en les donnant en argument à la fonction de la famille des `apply` grâce aux `...` (rappel : il s'agit de la deuxième utilité de l'argument `...` mentionnée dans les notes sur les concepts de base en R).

Note : La fonction `aggregate` accepte aussi une formule en entrée.

Autres fonctions pour réaliser des calculs par niveaux de facteurs (comme `tapply`, `by` et `aggregate`) :

- fonctions `group_by` et `summarize` du package `dplyr` utilisées conjointement ;
- opérateur `[` du package `data.table` utilisé en exploitant son argument `by`.

Écriture de conditions logiques

Fonctions opérant de façon vectorielle :

- Opérateurs de comparaison : `==`, `!=`, `>`, `>=` , `<`, `<=`.
- Opérateurs et fonction logiques : `!` (négation), `&` (et), `|` (ou), `xor` (ou exclusif).
- Opérateur de comparaison à un ensemble de valeurs : `%in%`.
- Fonctions de comparaison pour caractères spéciaux : `is.na`, `is.nan`, `is.infinite`.

Fonctions retournant toujours un logique de longueur 1 :

- Opérateurs logiques non vectoriels : `&&` (et), `||` (ou), `isTRUE`, `isFALSE`.
- Fonctions qui condensent un vecteur logique en une seule valeur logique : `all`, `any`.
- Fonctions de vérification de type : `is.`(`numeric`/`character`/`logical`/`vector`/`matrix`/`array`/`list`/`data.frame`/`factor`/`null`/...) (il en existe beaucoup !).

Comparaison de deux objets R

- Pour comparer uniquement les valeurs, pas les attributs : `all(x == y)`.
- Pour comparer les objets dans leur totalité (valeurs, attributs, type de l'objet ou des éléments) : `identical(x, y)`.
- Pour comparer les valeurs et les attributs, en acceptant des différences dans les valeurs numériques selon une certaine tolérance : `all.equal(x, y)`.

Références

Livres :

- Cotton, R. (2013). Learning R : A Step-by-Step Function Guide to Data Analysis, O'Reilly Media.
- Teetor, P. (2011). R Cookbook. O'Reilly Media. http://www.cookbook-r.com/
- Muenchen, R. A. (2011). R for SAS and SPSS Users. Second edition. Springer.
- Zumel, N. et Mount, J. (2014). Practical Data Science with R. Manning Publications Co.
- Zuur, A. F., Ieno, E. N. et Meesters, E. H.W.G. (2009). A Beginner's Guide to R. Springer.

Ressource web pour mieux comprendre les fonctions de la famille des `apply` :

- https://www.datacamp.com/community/tutorials/r-tutorial-apply-family