Calculs par sections d'un objet R

Sophie Baillargeon, Université Laval

2021-02-01

Table des matières

Note préliminaire : Lors de leur dernière mise à jour, ces notes ont été révisées en utilisant R version 4.0.3, le package ***`dplyr`*** *version 1.0.3 et le package* ***`data.table`*** *version 1.13.6. Pour d'autres versions, les informations peuvent différer.*

Dans le cadre d'une analyse exploratoire de données, certains calculs doivent être répétés sur différentes parties d'un jeu de données, par exemple diverses variables ou des sous-groupes d'observations. Le R de base propose plusieurs fonctions pour arriver à réaliser de tels calculs répétitifs sans écrire de boucle : les fonctions de la famille des `apply`. Ces fonctions sont présentées ici. Deux solutions de rechange pour le cas particulier du calcul de statistiques selon les niveaux de facteurs sont également présentées.

---

1 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 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`.

1.1 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.

```
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
```

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

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

```
# 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.

```
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.

1.1.1 Fonctions raccourcies

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.

1.2 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.

1.3 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 `tapply` 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
```

1.4 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

2 Solutions de rechange pour 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 brièvement ici en reproduisant les deux exemples précédents.

2.1 Package `dplyr`

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

Reproduisons avec `dplyr` les deux derniers exemples présentés pour la fonction `aggregate`.

```
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))
```

```
## `summarise()` has grouped output by 'conc'. You can override using the `.groups` argument.
```

```
## # 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
```

Les statistiques obtenues n'ont pas changé, mais elles sont retournées dans un tibble au lieu d'un data frame.

Il faut fournir comme premier argument à la fonction `summarize` le jeu de données sur lequel effectuer les calculs. De plus, pour réaliser des calculs selon les niveaux d'un facteur, le jeu de données doit être préalablement brisé en sous-groupes avec la fonction `group_by`. Par exemple, l'expression `group_by(Puromycin, conc, state)` permet d'indiquer que les observations du jeu de données `Puromycin` (argument 1 de `group_by` = jeu de données) doivent être groupées selon les niveaux des variables `conc` et `state` (arguments suivants de `group_by` = noms des colonnes du jeu de données contenant les facteurs selon lesquels créer les groupes). L'appel à la fonction `group_by` est donc assigné au premier argument de `summarize`. Les arguments suivants de `summarize` servent à définir les statistiques à calculer. Autant de statistiques que désiré peuvent être demandées. Les statistiques sont définies par des appels à des fonctions R de calcul statistique, en fournissant comme premier argument à la fonction le nom d'une variable dans le jeu de données (p. ex. `mean(conc)`). Si cette expression est précédée d'un opérateur `=` et d'un nom, celui-ci apparaîtra dans la sortie produite.

Il aurait été possible de passer d'autres arguments dans les appels aux fonctions de calcul statistique. Par exemple, reprenons le premier exemple ci-dessus et demandons le calcul de moyennes tronquées.

```
summarize(
  group_by(Puromycin, state),
  conc = mean(conc, trim = 0.1),
  rate = mean(rate, trim = 0.1),
  n_obs = n()
)
```

```
## # A tibble: 2 x 4
##   state      conc  rate n_obs
## * <fct>     <dbl> <dbl> <int>
## 1 treated   0.302  144.    12
## 2 untreated 0.213  112.    11
```

Ici, le calcul du nombre d'observations par sous-groupe a aussi été demandé avec la fonction `n` du package `dplyr`. Cette fonction n'a besoin d'aucun argument.

Notons que les fonctions de `dplyr` sont souvent utilisées avec l'opérateur `%>%`, nommé « *forward-pipe operator* ». En utilisant cet opérateur, la dernière instruction aurait l'allure suivante.

```
Puromycin %>%
  group_by(state) %>%
  summarize(
    conc = mean(conc, trim = 0.1),
    rate = mean(rate, trim = 0.1),
    n_obs = n()
  )
```

```
## # A tibble: 2 x 4
##   state      conc  rate n_obs
## * <fct>     <dbl> <dbl> <int>
## 1 treated   0.302  144.    12
## 2 untreated 0.213  112.    11
```

L'utilisation de l'opérateur `%>%` sera expliquée plus tard, dans les notes sur les bonnes pratiques. Bien que son utilisation avec les fonctions du `tidyverse` soit très populaire, elle n'est pas obligatoire.

Pour obtenir plus d'informations sur les calculs par niveaux de facteurs avec le package `dplyr` :

- https://dplyr.tidyverse.org/articles/grouping.html#summarise-
- https://stt4230.rbind.io/tutoriels_etudiants/hiver_2016/agreger_donnees_dplyr/

2.2 Package `data.table`

Il est aussi possible de réaliser ces agrégations grâce à l'argument `by` de l'opérateur `[` du package `data.table`. Ce package offre l'avantage de pouvoir effectuer ces opérations rapidement sur de grands jeux de données.

Reproduisons encore une fois les deux derniers exemples présentés pour la fonction `aggregate`.

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

Tout d'abord, le jeu de données doit être transformé en data table.

```
# Exemple avec deux variables réponses et une variable explicative (de groupement)
Puromycin_dt[, j = .(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[, j = .(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
```

Ensuite, la clé est de spécifier adéquatement les arguments `j` et `by` de l'opérateur `[` pour le data table. Cet opérateur avait été présenté dans les notes sur les structures de données en R.

Pour obtenir plus d'informations sur les calculs par niveaux de facteurs avec le package `data.table` :

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

---

3 Résumé

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 `...` (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

Solutions de rechange à `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`.

Références

R Core Team (2020). *R : A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/

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

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

Documentation des package mentionnés apportant des solutions de rechange pour des calculs par niveaux de facteurs :

- package `dplyr` :
  - page web du package sur le CRAN : https://CRAN.R-project.org/package=dplyr

  - documentation du package : https://dplyr.tidyverse.org/
  - informations sur les calculs par niveaux de facteurs :
    * https://dplyr.tidyverse.org/articles/grouping.html#summarise-
    * https://stt4230.rbind.io/tutoriels_etudiants/hiver_2016/agreger_donnees_dplyr/
- package `data.table` :
  - page web du package sur le CRAN : https://CRAN.R-project.org/package=data.table
  - documentation du package : https://rdatatable.gitlab.io/data.table/
  - informations sur les calculs par niveaux de facteurs :
    * https://rdatatable.gitlab.io/data.table/articles/datatable-intro.html#aggregations
    * https://stt4230.rbind.io/tutoriels_etudiants/hiver_2017/data.table/