

Introduction à R - partie 3

Aurélie Labbe

Avant de commencer, assurez-vous que vous avez bien téléchargé tous les fichiers nécessaires pour la séance (pdf, code **R** et données) et que tout est sauvé dans un répertoire de votre ordinateur. Assurez-vous aussi que le code `IntroR_part3.R` est bien ouvert dans l'éditeur de RStudio. Finalement, vous devez avoir assimilé les notions de la première et de la deuxième partie de l'introduction à **R** avant de commencer cette séance.

Dans ce document, nous allons couvrir les thèmes suivants:

1. Programmation avancée: boucles et création de fonction
2. Gestion des valeurs manquantes
3. Les objets de type `list` et `array`
4. La commande `apply` et ses variantes

Données

Nous allons utiliser ici les données présentes dans le fichier `salesmen.txt` qui contient les ventes réalisées par les employés d'une compagnie. Chaque ligne du fichier représente un employé et les variables sont les suivantes:

- `EmpID`: ID de l'employé
- `Month`: mois de l'année (de 1 à 12)
- `nSales`: nombre de ventes totales
- `TotSales`: total des ventes (en \$)

Programmation avancée: boucles et fonctions

Il est très facile d'écrire ses propres fonctions dans R. Une fois écrites, elles deviennent des fonctions permanentes dans votre version de R, et disponibles tout comme n'importe quelle autre fonction. Nous allons vous présenter quelques exemples. Voici tout d'abord un exemple facile, où nous allons créer une fonction qui additionne deux nombres. Bien sûr, ce n'est pas un exemple très pertinent ni utile, mais cela va nous permettre de voir comment les fonctions sont construites en général.

```
add.2.numbers <- function(a, b){
  sum <- a+b
  return(sum)
}
add.2.numbers(3, 4)
```

```
[1] 7
```

C'est tout ce dont nous avons besoin pour créer une fonction !

Une fonction peut retourner plus qu'un résultat, via une liste:

```
sommediff <- function(a, b){
  somme <- a + b
  diff <- a - b
  # par défaut, le dernier énoncé est retourné si return() est omis
  list("somme" = somme,
       "diff" = diff)
}
sommediff(3, 4)
```

```
$somme
```

```
[1] 7
```

```
$diff
```

```
[1] -1
```

Voici maintenant un exemple d'utilisation de boucle, permettant d'automatiser la répétition d'une tâche. Comme **R** est un langage vectorisé, les opérations sur les vecteurs sans boucle, à moindre coût de calcul.

```
# Afficher une série de chiffres successifs à l'écran
for (i in seq_len(4)){ # pour 1 à 4
  print(i^2)
}
```

```
[1] 1
```

```
[1] 4
```

```
[1] 9
```

```
[1] 16
```

```
# Autre exemple
# Afficher une série d'éléments d'un ensemble donné à l'écran
for (w in c("red", "blue", "green")){
  print(w)
}
```

```
[1] "red"
[1] "blue"
[1] "green"
```

Il existe un objet en **R** qui est un pointeur objet “vide” (NULL) et qui est très utile lorsqu’on veut réinitialiser un élément d’une liste

Voici un autre exemple, où nous allons créer une fonction qui construit de façon automatisée des graphiques pour les données `salesmen.txt`.

```
#' Créer des graphiques pour un employé
#' @param x base de données
#' @param emp identifiant de l'employé que l'on veut analyser
#' @return NULL; des graphiques
sales <- function(x, emp){
  if(sum(x$EmpID == emp) == 0){
    stop("Cet employé n'existe pas.")
  }
  par(mfrow = c(1, 2))
  plot(x = 1:12,
       y = x$nSales[x$EmpID == emp],
       type = 'l',
       xlab = "mois",
       ylab = "nombre de transactions")
  title(paste("nombre de transactions de l'employé", emp, "par mois"))
  plot(x = 1:12,
       y = (x$TotSales/x$nSales)[x$EmpID==emp],
       type = 'l',
       xlab = "mois",
       ylab = "ventes moyenne")
  title("Montant moyen des ventes par mois")
}
```

Exercice 0.1. Lire les commandes de la fonction et bien comprendre ce qu’elle fait. Exécuter ensuite la fonction avec les données du fichier `salesmen.txt` et `ID=1001`, puis `ID=1006` puis enfin avec `ID=1`.

Valeurs manquantes

Dans la plupart des cas, les données avec lesquelles vous allez travailler vont contenir des valeurs manquantes. Ces valeurs sont représentées par des NA (*not available*) dans **R**. Voici quelques fonctions prédéfinies reliées au traitement des valeurs manquantes:

```
# Création d'un vecteur qui contient une valeur manquante
x <- c(1, NA, 5.6, -1, 0)

# Retourne des indicateurs de valeurs manquantes TRUE/FALSE
is.na(x)
```

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

```
# Que se passe-t-il lorsque on veut prendre le log d'une valeur NA ?
log(x)
```

```
Warning in log(x): NaNs produced
```

```
[1] 0.000000      NA 1.722767      NaN      -Inf
```

```
# Retourne des indicateurs logiques de valeurs manquantes "not a number" (NaN)
is.nan(x)
```

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

```
is.finite(x)
```

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

```
is.finite(x) # vrai si NA, NaN, ou Inf, -Inf
```

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

Plusieurs fonctions ont un traitement spécial des valeurs manquantes. C'est le cas des fonctions `mean`, `var`, `sd`, etc. Voici un exemple:

```
mean(x)
```

```
[1] NA
```

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

[1] 1.4

On peut aussi extraire les valeurs non manquantes d'un vecteur ou bien calculer le nombre de valeurs manquantes tel que ci-dessous:

```
sum(is.na(x)) # nombre de valeurs manquantes
```

[1] 1

```
x[!is.na(x)] # sous-ensemble de valeurs non-manquantes
```

```
[1] 1.0 5.6 -1.0 0.0
```

Voici maintenant un exemple de fonction que nous pouvons créer qui permet de retourner le nombre de valeurs manquantes dans un vecteur. Vous pouvez en même temps y voir un exemple d'utilisation de la commande `if`.

```
#' Nombre de valeurs manquantes
#'#' @param x vecteur
#'#' @param prop proportion de valeurs manquantes, 0.1 par défaut
#'#' @return nombre de valeurs manquantes
nb.na <- function(x, prop = 0.1){
  if(mean(is.na(x)) >= prop){
    warning(paste("Proportion de valeurs manquantes supérieure à", propr))
  }
  return(sum(is.na(x)))
}
```

Exercice 0.2. Écrire une fonction qui prend comme entrée une table de données (`data.frame`) et un vecteur représentant des indices de colonnes. La fonction devrait retourner, pour chaque colonne, le nombre d'observations dont la valeur est supérieure à la moyenne de la colonne.

Objets de classe list et 'array'

Nous allons voir ici deux types d'objets un peu plus complexes.

Un **array** est une généralisation d'une matrice, en plus de deux dimensions. Si un **array** est de dimension trois, on peut accéder aux différents éléments avec l'opérateur `[, ,]`, comme pour les vecteurs et matrices. Voici un exemple de construction de **array** qui va nous permettre de mieux visualiser les données de vente.

```
donnees <- read.table("Data/salesmen.txt", header = TRUE)

# Mettre les données sous la forme d'un cube
# Dimensions: 12 (mois) * 150 (clients) * 4 (variables)
donnees_array <- array(as.matrix(donnees), c(12, 150, 4))

# Données pour le 1er client, 1er mois
donnees_array[1, 1, ]
```

```
[1] 100.00 1.00 16.00 15687.46
```

```
# Données pour le 1er client
donnees_array[, 1, ]
```

	[,1]	[,2]	[,3]	[,4]
[1,]	100	1	16	15687.46
[2,]	100	2	21	21496.56
[3,]	100	3	11	12015.78
[4,]	100	4	18	18537.36
[5,]	100	5	13	13440.72
[6,]	100	6	13	12748.57
[7,]	100	7	17	16931.61
[8,]	100	8	21	21864.75
[9,]	100	9	20	20147.32
[10,]	100	10	13	12928.39
[11,]	100	11	19	19046.06
[12,]	100	12	21	19194.89

```
# Données pour le 1er mois
head(donnees_array[1, , ])
```

	[,1]	[,2]	[,3]	[,4]
[1,]	100	1	16	15687.46
[2,]	101	1	11	11559.90
[3,]	102	1	7	7783.04
[4,]	103	1	13	13323.09
[5,]	104	1	21	20399.71
[6,]	105	1	11	11035.39

```
# Nombre de ventes pour tous les employés en janvier (6 premières lignes)
head(donnees_array[1, , c(1, 3)])
```

	[,1]	[,2]
[1,]	100	16
[2,]	101	11
[3,]	102	7
[4,]	103	13
[5,]	104	21
[6,]	105	11

```
# Vente moyenne mensuelle pour le 3e employé
donnees_array[, 3, 4]/donnees_array[, 3, 3]
```

[1]	1111.8629	904.9420	975.1046	NA	908.8900	985.2789	1051.4989
[8]	1067.3665	996.9230	1000.7414	1021.9857	943.7427		

Travailler avec des (hyper)cubes requiert un niveau d'abstraction plus élevé, mais une fois que vous allez être habitués, vous allez voir que cela peut être une approche très efficace.

Exercice 0.3. Afficher toutes les données du nombre de vente pour le mois de mars pour tous les employés dont l'identifiant commence par 10.

Un autre objet très utilisé en R est le type `list`. Une liste est un vecteur d'éléments qui eux même peuvent être de n'importe quel format (vecteur, matrice, etc.). Voici un exemple de création de liste:

```
donnees <- read.table("Data/salesmen.txt", header = TRUE)

# Creation d'une liste qui va contenir 5 elements:
# year, succursale, ID, nbre de ventes par mois et total vente par mois
donnees_list <- with(donnees,
```

```

list(
  year = 2012,
  succ = "Montreal Centre",
  ID = unique(EmpID),
  nb = t(matrix(nSales, nrow = 12)),
  montant = t(matrix(TotSales, nrow = 12))
)
names(donnees_list)

```

```
[1] "year"      "succ"      "ID"        "nb"        "montant"
```

Pour accéder aux éléments d'une liste, il faut utiliser le symbole `[[]]` avec le numéro de l'élément auquel on veut accéder. On peut aussi utiliser le symbole `$` avec le nom de l'élément, comme pour un dataframe. Voici quelques exemples:

```

# Extraire de la liste le nbre de vente pour tous les employés par mois
# (6 premières lignes)
head(donnees_list$nb)

```

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]	[,9]	[,10]	[,11]	[,12]
[1,]	16	21	11	18	13	13	17	21	20	13	19	21
[2,]	11	6	12	12	15	23	17	20	18	16	15	24
[3,]	7	15	13	NA	15	18	19	20	23	21	28	30
[4,]	13	12	9	21	19	22	22	12	16	25	19	21
[5,]	21	11	20	17	21	25	23	21	28	18	31	21
[6,]	11	19	17	25	20	19	21	27	24	29	32	32

```

# Autre facon de faire la même chose...
head(donnees_list[[4]])

```

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]	[,9]	[,10]	[,11]	[,12]
[1,]	16	21	11	18	13	13	17	21	20	13	19	21
[2,]	11	6	12	12	15	23	17	20	18	16	15	24
[3,]	7	15	13	NA	15	18	19	20	23	21	28	30
[4,]	13	12	9	21	19	22	22	12	16	25	19	21
[5,]	21	11	20	17	21	25	23	21	28	18	31	21
[6,]	11	19	17	25	20	19	21	27	24	29	32	32


```
# Extraire de la liste le nbre de vente pour tous les mois, premier sujet
donnees_list[[4]][1, ]
```

```
[1] 16 21 11 18 13 13 17 21 20 13 19 21
```

Exercice 0.4. Afficher toutes les données du montant de vente pour le mois de mars pour tous les employés dont l'identifiant commence par 10.

Commande `apply`

Comme nous l'avons vu, les boucles peuvent être très utiles pour automatiser certains processus. Cependant, les boucles ne sont pas très efficaces d'un point de vue computationnel. Il existe des fonctions dans R qui permettent d'éviter de faire des boucles tout en permettant une répétition automatique de tâches. C'est ce que nous allons voir avec la commande `apply` et ses variants `lapply`, `tapply` et `sapply`.

Nous allons illustrer l'utilisation de ces fonctions à l'aide de la matrice du nombre de vente pour tous les employés, que nous allons extraire à partir du array que nous avons créé dans la section précédente:

```
# Lignes = mois
# Colonnes = employés
nb_vente <- donnees_array[, , 3]
dim(nb_vente)
```

```
[1] 12 150
```

L'idée de la commande `apply` est d'**appliquer** une fonction (par exemple `mean`) de façon systématique et répétitive à toutes les lignes ou colonne d'une matrice. Cette fonction peut aussi s'appliquer à un hypercube de classe `array`. Le premier argument de la fonction est l'objet (données) sur lequel on veut appliquer la fonction, le second argument prend les indices des dimensions que l'on veut conserver dans l'objet de sortie. Ainsi, `MARGIN = 1` si la fonction est appliquée aux lignes de l'objet, et prend `MARGIN = 2` si elle est appliquée aux colonnes dans le cas d'une matrice. Le dernier argument est le nom de la fonction que l'on veut appliquer (par ex., `mean`). On peut aussi rajouter d'autres options spécifiques à la fonction en question.

Voici des exemples d'utilisation de la fonction:

```
# Applique la fonction "mean" à toutes les lignes (donc mois) de la matrice
# Cela donne la moyenne des ventes par mois
apply(nb_vente, 1, mean)
```

```
[1]      NA      NA      NA      NA      NA      NA      NA      NA
[9]      NA      NA      NA 25.44667
```

```
# Même chose, en spécifiant l'option `na.rm` de la fonction `mean`
apply(nb_vente, 1, mean, na.rm = TRUE)
```

```
[1] 13.92308 14.50350 15.70423 17.00000 18.20946 18.80272 20.69128 20.45638
[9] 21.93243 22.80537 23.65986 25.44667
```

```
# Applique la fonction "mean" à toutes les colonnes (donc employés) de la matrice
# Cela donne la moyenne des ventes par employé (6 premières lignes)
head(apply(nb_vente, 2, mean, na.rm = TRUE))
```

```
[1] 16.91667 15.75000 19.00000 17.58333 21.41667 23.00000
```

```
# Valeurs min et max par mois
apply(nb_vente, 1, range, na.rm = TRUE)
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
[1,]    5    5    6    6    7    6    7    7    10    12    10    11
[2,]   29   29   33   31   33   32   37   32   35   38   40   43
```

```
# Valeurs min et max par employé pour les 10 premiers employés
apply(nb_vente[, 1:10],
      MARGIN = 2, #dimensions à conserver (ici colonnes)
      FUN = range, # fonction à appliquer
      na.rm = TRUE) # paramètres additionnels à passer à la fonction
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]   11    6    7    9   11   11    8   12    9   14
[2,]   21   24   30   25   31   32   20   26   27   30
```

Les variantes `sapply` et `lapply` s'appliquent aux vecteurs et aux objets de type liste. Voici un exemple:

```
fn <- function(i, num = 1:10){
  sum(num^i)
}
sapply(0:5, fn)
```

[1] 10 55 385 3025 25333 220825

Les commandes `apply` peuvent paraître étrange au début, surtout pour les utilisateurs qui sont habitués aux boucles. Cependant, l'utilisation de ces commandes rend le code plus compacte et élimine les erreurs liées aux indices dans les boucles. Notez qu'un bon programmeur **R** utilise rarement des boucles.

Exercice final

Le fichier `salesmen.txt` avec lequel nous avons travaillé contient en réalité les données de vente pour plusieurs branches de la compagnie. Les 2 premiers caractères du ID de l'employé représentent l'identifiant de la branche.

1. Combien de valeurs manquantes sont présentes dans ces données pour chaque variable ?
2. Que pouvez-vous remarquer à propos des employés de la branche 25 ? Pouvez-vous trouver une raison plausible pour les valeurs manquantes ?
3. Déterminez combien d'employés équivalent temps pleins sont présents pour chaque branche. Notez que si un employé a travaillé pendant 3 mois, cela correspond à 3/12 d'équivalent temps plein.
4. Créer une liste où chaque élément est une matrice du nombre total de vente (lignes = employés et col = mois) pour une branche.
5. En utilisant une des commandes `apply` (ou ses dérivés), calculez le nombre total de vente pour chaque branche. Faire un graphique (diagramme à bande, voir ?`barplot`) qui illustre la contribution de chaque branche aux ventes de la compagnie.
6. Calculez une mesure de performance: les ventes de chaque branche par employé équivalent temps plein. Créer un graphique qui permet de comparer l'efficacité des branches.
7. Écrivez une fonction qui prend comme input l'identifiant (ID) d'une branche et qui retourne un résumé de l'évolution des ventes (nombre et total des ventes) au travers du temps.