

2 - Manipulation de données

Séries Temporelles avec R - Initiation

Anna Smyk, Tanguy Barthelemy

Insee - Département des Méthodes Statistiques



Section 1

Sommaire

Sommaire

- Manipulation de tableaux
- Présentation R base
- Présentation tidyverse

Section 2

Paradigmes

Paradigmes

Pour gérer tout les objets présentés à la séquence précédente, il existe plusieurs paradigmes. Nous présentons ici les 2 plus utilisés :

- R base
- tidyverse

Section 3

R base

R base

La logique de R base s'appuie sur la structure matricielle des listes et vecteurs et propose 3 fonctions principales :

- `[` : Création de sous-ensembles
- `[[` : Extraction
- `$` : Extraction

Sous-ensemble

La notion de **sous-ensemble** est bien différente de la notion d'extraction. On part d'un objet d'un certain type et on récupère un autre objet (souvent plus petit) mais de même type !

Exemple - Vecteur

```
v1 <- c(1, 45, 456145)  
v1[1]
```

```
[1] 1
```

Exemple - list()

```
l1 <- list(1, "element 2", 1:10, "element4", 5)  
l1[1:3]
```

```
[[1]]
```

```
[1] 1
```

```
[[2]]
```

```
[1] "element 2"
```

```
[[3]]
```

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

Exemple - data.frame()

```
df1 <- data.frame(  
  col1 = 1:10,  
  col2 = letters[1:10],  
  col3 = TRUE  
)  
df1[2:5, c(1, 3)]
```

	col1	col3
2	2	TRUE
3	3	TRUE
4	4	TRUE
5	5	TRUE

Extraction

L'extraction permet de récupérer le contenu d'une liste, un élément d'une liste. Dans le cas d'un vecteur, on passe à un scalaire.

Par exemple,

```
l1 <- list(a = 1, b = "element 2", c = 1:10, d = "element4", e = 5)
l1[[1]]
```

```
[1] 1
```

```
l1[["d"]]
```

```
[1] "element4"
```

```
l1$b
```

```
[1] "element 2"
```

Extraction

Pour les `data.frame()`, cela revient à récupérer une colonne sous la forme d'un vecteur :

```
df1 <- data.frame(  
  col1 = 1:10,  
  col2 = letters[1:10],  
  col3 = TRUE  
)  
df1[[1]]
```

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

```
df1[["col2"]]
```

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

```
df1$col3
```

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

Argument drop = FALSE

Pour les `data.frame()`, lorsqu'on ne sélectionne qu'une seule colonne, l'opérateur `[` effectue une extraction par défaut.

```
df1 <- data.frame(  
  col1 = 1:10,  
  col2 = letters[1:10],  
  col3 = TRUE  
)  
df1[2:5, 1]
```

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

Argument drop = FALSE

Pour garder la structure de `data.frame`, il faut utiliser l'argument `drop = FALSE` :

```
df1[2:5, 1, drop = FALSE]
```

	col1
2	2
3	3
4	4
5	5

Argument drop = FALSE

C'est évidemment aussi le cas pour les listes :

```
l1 <- list(a = 1, b = "element 2", c = 1:10, d = "element4", e = 5)
l1[[1]]
```

```
[1] 1
```

```
l1[1]
```

```
$a
```

```
[1] 1
```

C'est pourquoi il est plus sûr d'utiliser l'opérateur `[[` lors d'une extraction, même pour un vecteur.

Récupérer des informations

Pour récupérer des informations sur un objet en R, il y a plusieurs fonctions utiles :

- `print()` va afficher l'objet (tel que défini par la classe de l'objet)
- `summary()` donnera plus de détail (notamment pour les modèles statistiques)
- `head()` et `tail()` retourneront respectivement les premiers ou les derniers éléments
- `str()` affichera une description du type de l'objet
- `dput()` retournera l'objet entier réel tel qu'il est enregistré en mémoire

Récupérer des informations - listes et vecteurs

Face à une liste ou un vecteur, il est possible de récupérer la longueur avec `length()` ou les longueurs des sous éléments (pour une liste) avec `lengths()`.

Une liste et un vecteur peuvent être nommés c'est-à-dire indexer les éléments non pas par un numéro mais par un nom.

On peut alors utiliser la fonction `names()` pour récupérer ces noms.

```
l1 <- list(a = 1, b = "element 2", c = 1:10, d = "element4", e = 5)
v1 <- c(a = 1, b = 3, c = 5)
names(l1)
```

```
[1] "a" "b" "c" "d" "e"
```

```
names(v1)
```

```
[1] "a" "b" "c"
```

Récupérer des informations - `data.frame()`

On se rappelle que les `data.frame()` sont des listes, donc toutes les opérations qui fonctionnaient pour les listes fonctionneront avec des `data.frame()`.

Fonctionnalités additionnelles :

- `View()` pour visualiser la table
- `dim()` pour récupérer les dimensions de la table
- `ncol()` pour connaître le nombre de colonnes de la table
- `nrow()` pour connaître le nombre de lignes de la table
- `colnames()` pour récupérer les noms des colonnes
- `rownames()` pour récupérer les noms des lignes

Modifications - Vecteur et liste

Pour ajouter un élément à un vecteur ou une liste, on utilise l'opérateur `c`. Exemple :

```
c(v1, 3)
```

Pour retirer un élément, on utilise l'indexation négative : `v1[-1]`

Pour modifier la valeur d'un élément, on utilise l'assignation sous opérateur :

```
v1[[2]] <- 9  
l1[[2]] <- 9
```

i Subsetting

L'opérateur `[←` propose les même fonctionnalités de modifications.

Modifications - Vecteur et liste

De même pour un `data.frame()`, il est possible :

- d'ajouter une colonne `df[, 7] <- 4` ou `df$serie <- "id_XXX"`
- de modifier une valeur `df[2, "col3"] <- "val3"`
- de supprimer une colonne `df[, -4]` ou `df[, 4] <- NULL` ou `df$serie <- NULL`

Section 4

tidyverse

tidyverse



Le **tidyverse** adopte une logique vectorielle et fonctionnelle. Il est centré autour de la notion de **pipe**.

Le **pipe** `%>%` permet d'enchaîner les opérations (ici via des appels de fonctions). Cela permet de reproduire une chaîne de production en R.

```
library("tidyverse")
```

Le pipe %>%

Ce qu'on aurait écrit avant :

```
donnees_brutes <- read.csv("chemin/vers/mes/donnees")
donnees_traitees1 <- traitement1(donnees_brutes)
donnees_traitees2 <- traitement2(donnees_traitees1)
donnees_traitees3 <- traitement3(donnees_traitees2)
output_formatte <- formatage(donnees_traitees3)
```

Peut s'écrire :

```
output <- read.csv("chemin/vers/mes/donnees") %>%
  traitement1() %>%
  traitement2() %>%
  traitement3() %>%
  formatage()
```


tidyverse



Le tidyverse regroupe un grand ensemble de packages. Ces packages sont complémentaires dans le travail du R datascientist :

- [{readr}](#) : import de données
- [{stringr}](#) : manipulation de chaîne de caractère
- [{ggplot2}](#) : création de graphique
- [{dplyr}](#) : manipulation de table de données
- [{tidyr}](#) : création de structure tidy
- [{tibble}](#) : classe de tables
- [{purrr}](#) : vectorisation et boucles
- [{forcats}](#) : manipulation de variables qualitatives

Section 5

{dplyr}

{dplyr}

Ici nous présenterons surtout les verbes de [{dplyr}](#)

Via la structure de `tibble()` il est possible de modifier facilement une table de donnée.
Les fonctions les plus utiles sont :

- `filter()` : filtrer les observations de la table (supprimer des lignes)
- `mutate()` : ajouter ou modifier les colonnes
- `select()` : sélectionner les colonnes (supprimer des colonnes)
- `summarise()` : calculer des statistiques par groupe d'observations
- `arrange()` : trier la table
- `rename()` : renommer les colonnes
- `case_when()` : répartir les différents cas de figure

Exemple

```
population <- data.frame(  
  code = c("A", "B", "C", "D", "E", "F", "G", "H", "I", "J"),  
  dep = c(42L, 42L, 7L, 91L, 91L, 7L, 7L, 7L, 42L, 42L),  
  nb = c(530, 150542, 24561, 1021312, 552654, 202135,  
         88541, 12021, 2100000, 987661),  
  stringsAsFactors = FALSE  
)  
  
head(population)
```

	code	dep	nb
1	A	42	530
2	B	42	150542
3	C	7	24561
4	D	91	1021312
5	E	91	552654
6	F	7	202135

Exemple - filter()

```
population %>%  
  filter(dep = 42)
```

	code	dep	nb
1	A	42	530
2	B	42	150542
3	I	42	2100000
4	J	42	987661

Exemple - filter() by

```
population %>%  
  filter(nb = max(nb, na.rm = TRUE), .by = dep)
```

	code	dep	nb
1	D	91	1021312
2	F	7	202135
3	I	42	2100000

Exemple - mutate() et case_when()

```
population %>%  
  mutate(taille = case_when(  
    nb > 1500000 ~ "grandes zones métropolitaines",  
    nb > 500000 ~ "zones métropolitaines",  
    nb > 200000 ~ "zones urbaines moyennes",  
    nb > 50000 ~ "petites zones urbaines",  
    nb > 2000 ~ "ville moyenne",  
    nb ≥ 0 & nb ≤ 2000 ~ "village",  
    TRUE ~ "Inconnu !"  
  ))
```

	code	dep	nb	taille
1	A	42	530	village
2	B	42	150542	petites zones urbaines
3	C	7	24561	ville moyenne
4	D	91	1021312	zones métropolitaines
5	E	91	552654	zones métropolitaines

Exemple - mutate() by

```
population %>%  
  mutate(pourcent_pop_metro = 100 * nb / sum(nb, na.rm = TRUE)) %>%  
  mutate(pourcent_pop_dep = 100 * nb / sum(nb, na.rm = TRUE), .by = dep)
```

	code	dep	nb	pourcent_pop_metro	pourcent_pop_dep
1	A	42	530	0.01031137	0.01636442
2	B	42	150542	2.92885719	4.64817569
3	C	7	24561	0.47784446	7.50508773
4	D	91	1021312	19.87004950	64.88780571
5	E	91	552654	10.75211330	35.11219429
6	F	7	202135	3.93262045	61.76625170
7	G	7	88541	1.72260196	27.05541194
8	H	7	12021	0.23387355	3.67324863
9	I	42	2100000	40.85637292	64.84017052
10	J	42	987661	19.21535530	30.49528936

Exemple - rename(), select() et arrange()

```
population %>%  
  rename(nombre_habitants = nb)
```

	code dep	nombre_habitants
1	A 42	530
2	B 42	150542
3	C 7	24561
4	D 91	1021312
5	E 91	552654
6	F 7	202135
7	G 7	88541
8	H 7	12021
9	I 42	2100000
10	J 42	987661

```
population %>%  
  arrange(nb)
```

	code dep	nb
1	A 42	530
2	H 7	12021
3	C 7	24561
4	G 7	88541
5	B 42	150542
6	F 7	202135
7	E 91	552654
8	J 42	987661
9	D 91	1021312
10	I 42	2100000

```
population %>%  
  select(nb, dep)
```

	nb	dep
1	530	42
2	150542	42
3	24561	7
4	1021312	91
5	552654	91
6	202135	7
7	88541	7
8	12021	7
9	2100000	42
10	987661	42

Exemple - mutate() by

```
population %>%  
  summarise(mean_pop = mean(nb, na.rm = TRUE), .by = dep)
```

	dep	mean_pop
1	42	809683.2
2	7	81814.5
3	91	786983.0

Syntaxe

Vous retrouverez souvent des notations de tidyverse ou de R base dans vos codes ou ceux de vos collègues.

Conseil :

- Il est important de maîtriser les 2
- et de choisir un paradigme pour ne pas trop mélanger les syntaxes.

Exercice pratique

Reprendre le jeu de donnée population et :

- ① Retourner la table triée par population et par département
- ② Récupérer la commune qui contient le moins d'habitants de chaque département
- ③ Enfin, calculer le ratio du nombre d'habitants entre la plus petite commune et la plus grande commune de chaque département

Section 6

Pivot et {tidyr}

Pivot et {tidyr}

Le package **tidyr** propose des fonctions de transposition.

Lorsqu'on parle de **transposition**, ce n'est pas la même transposition que pour les matrices.

Format *long* et *wide*

On distingue 2 formats de données :

- Le **format wide** qui est la représentation que l'on se fait d'un tableau :
 - autant de lignes que d'observations
 - autant de colonnes que de variables (+ 1 avec l'identification de l'observation)
- Le **format long** qui est un uniquement composé de 3 colonnes :
 - id : identifiant de l'observation
 - name : identifiant de la variable
 - value : valeur de cette variable pour cet individu

Ainsi les deux tableaux contiennent AUTANT d'information l'un que l'autre mais sont formatés différemment.

Changement de format

Pour passer d'un format à un autre, il faut utiliser les fonctions `pivot_XXX` du package **{tidyr}**.

- `pivot_longer()` pour passer au format long
- `pivot_wider()` pour passer au format wide

Exemple - pivot_longer()

Notre table population est au format wide car il y a bien une ligne par commune.

```
long_pop <- population %>%  
  pivot_longer(cols = c(dep, nb))  
head(long_pop)
```

```
# A tibble: 6 x 3  
  code name  value  
  <chr> <chr>  <dbl>  
1 A     dep      42  
2 A     nb     530  
3 B     dep      42  
4 B     nb    150542  
5 C     dep       7  
6 C     nb     24561
```

Exemple - pivot_longer

Notre table long_pop est au format long.

```
wide_pop <- long_pop %>%  
  pivot_wider()  
head(wide_pop)
```

```
# A tibble: 6 x 3  
  code    dep    nb  
  <chr> <dbl>  <dbl>  
1 A      42    530  
2 B      42 150542  
3 C       7   24561  
4 D      91 1021312  
5 E      91  552654  
6 F       7   202135
```

Exercice pratique - ts

Pour cet exercice, on va partir du jeu de données IPI. Ce jeu de donnée se structure de la manière suivante :

- une colonne date
- n colonnes avec les séries de l'IPI

Ici on traitera les données de l'IPI comme un `data.frame`.

- ① Importer le jeu de données
- ② Filtrer les valeurs entre 2012 et 2020
- ③ Récupérer la série qui contient la plus grande valeur et la plus petite.

Exercice pratique - Pour aller plus loin

Pour cette question, ne garder que les colonnes date et RF0899

- ④ Créer une variable S qui contient la moyenne de chaque série pour chaque mois de l'année
- ⑤ Centrer la variable S (lui retirer sa moyenne)
- ⑥ Calculer une variable SA qui vaut $RF0899 - S$

Pour cette question, prendre les 5 premières colonnes (dont date)

- ⑦ Calculer la variable SA pour chacune les 4 variables (autres que date)
- ⑧ Vérifier qu'aucune séries SA n'a de valeurs négatives.