3_2_cours

October 15, 2018

Table of Contents


```
In [2]: options(repr.matrix.max.cols=8, repr.matrix.max.rows=5)
```

1 Opérations sur les matrices

1.1 apply

Nous avons vu qu'il était possible de faire des opérations sur les matrices en ligne ou en colonne. Toutefois, ce n'est pas toutes les fonctions statistiques qui sont applicables sur des colonnes et/ou lignes comme `colMeans`. Pour appliquer d'utres sortes de fonctions, nous devons utiliser la fonction `apply`.

On peut alors utiliser `apply` lorsqu'on veut appliquer un calcule ou une fonction quelconque (FUN) sur des colonnes ou des lignes d'une matrice (incluant les matrices plus que 2D)

Soit une matrice de 12 premiers entiers;

```
In [2]: m<-matrix(1:12, nrow=3)
        m
```

1 4 7 10
2 5 8 11
3 6 9 12

Calculons le logarithme naturel de chaque élément de cette matrice:

```
In [3]: h<-apply(m, c(1,2), log) #c(1,2) ça veut dire sur ligne et colonne
        h
```

0.0000000 1.386294 1.945910 2.302585
0.6931472 1.609438 2.079442 2.397895
1.0986123 1.791759 2.197225 2.484907

Créons une matrice `3 x 1` qui nous retourne le résultat de la somme de chaque ligne;

```
In [4]: h<-matrix(apply(m, 1, sum))
        h
```

22
26
30

Si nous comparerons à la fonction `rowSums` que nous avons vue;

```
In [7]: rowSums(m)
```

1. 22 2. 26 3. 30

1.2 lapply

La fonction lapply applique une fonction quelconque (FUN) à tous les éléments d'un vecteur ou d'une liste X et retourne le résultat sous forme de liste.

Dans l'exemple suivant, nous avons une liste de trois vecteurs {vecteur_1, vecteur_2, vecteur_3} de taille différente, on voudrait savoir quelle est la taille de chaque élément, on voudrait la réponse dans une **liste**;

```
In [4]: x <- list(vecteur_1 = 1, vecteur_2 = 1:17, vecteur_3 = 55:97)
        lapply(x, FUN = length)
```

$vecteur_1 1

$vecteur_2 17

$vecteur_3 43

Dans le résultat ci-haut, la fonction `lapply` nous a retourné une liste de trois éléments avec la taille de chaque vecteur.

Regardons un autre exemple où nous cherchons à créer quatre échantillons aléatoires de taille {5, 6, 7, 8} tirés du vecteur `x= 1 2 3 4 5 6 7 8 9 10`

```
In [5]: set.seed(123)
        lapply(5:8, sample, x = 1:10)
```

1. (a) 3 (b) 8 (c) 4 (d) 7 (e) 6
2. (a) 1 (b) 5 (c) 8 (d) 4 (e) 3 (f) 9
3. (a) 5 (b) 7 (c) 10 (d) 1 (e) 6 (f) 2 (g) 9
4. (a) 4 (b) 9 (c) 8 (d) 5 (e) 10 (f) 7 (g) 3 (h) 6

1.3 sapply

Dans certains cas, on voudrait appliquer une fonction quelconque sur une liste, mais on ne veut pas que `R` nous retourne une une liste, on désire plutôt que `R` nous retourne un vecteur. La fonction `sapply` fait exactement cela. Le résultat est donc **s**implifiée par rapport à celui de `lapply`, d'où le nom de la fonction.

La taille de chaque élément de notre liste;

```
In [7]: sapply(x, FUN = length)
```

vecteur_1 1 **vecteur_2** 17 **vecteur_3** 43

ou la somme de chaque élément de notre liste `x`

```
In [6]: sapply(x, FUN = sum)
```

vecteur_1 1 **vecteur_2** 153 **vecteur_3** 3268

Si le résultat de chaque application de la fonction est un vecteur et que les vecteurs sont tous de la même longueur, alors sapply retourne une matrice, remplie comme toujours par colonne :

```
In [7]: (x <- lapply(rep(5, 3), sample, x = 1:10))
```

1. (a) 6 (b) 10 (c) 3 (d) 2 (e) 9
2. (a) 10 (b) 7 (c) 9 (d) 1 (e) 3
3. (a) 8 (b) 2 (c) 3 (d) 9 (e) 1

```
In [8]: sapply(x, sort)
```

2	1	1
3	3	2
6	7	3
9	9	8
10	10	9

1.4 autre

Il existe d'autres façons de manipuler les matrices, listes, vecteurs ...etc. Dans ce cours nous avons couvert les trois principaux, toutefois, on peut avoir besoin dans certains cas d'utiliser `vapply`, `mapply`, `Map`, `rapply` ou même `tapply` qui s'apparentent tous aux trois fonctions que nous avons couverts avec plus d'options ou format différent du résultat obtenu.

2 Manipulation avec dplyr

2.1 le package dplyr

Dans ce cours, afin de manipuler les données, nous allons utiliser la librairie `dplyr` qui assure une manipulation plus intuitive des données. Toutefois, vous pouvez utiliser tout autre libraire ou même les fonctions de base de `R`.

```
In [11]: # install.packages("dplyr")
```

D'abord, téléchargeons un petit *df* afin d'illustrer la théorie. Dans ce *df*, nous avons les données d'un cycliste qui est sorti un jour d'été faire un petit tour dans l'île de Montréal. Chaque observation, représente le nombre de km parcourus d'un parcours (*lap*), le temps que ça a pris, la vitesse moyenne en `km/h` de chaque *lap*, la puissance moyenne en `watts` et finalement, les battements de cours par minutes.

```
In [9]: df<-read.csv("https://raw.githubusercontent.com/nmeraihi/data/master/exemple_2.txt")
        df
```

km	temps	vitesseMoyenne	puissanceMoyenne	bpm
1.24	4:01	19.1	160	134
4.84	9:42	30.2	133	146
1.02	1:57	30.8	141	139
17.61	36:11	29.2	125	144
9.27	19:10	29.0	121	143

Il possible d'ordonner les données avec la fonction de base de `R` appelé `order`. Par exemple, on voudrait ordonner notre *df* en ordre croissant sur la variable `puissanceMoyenne`

```
In [10]: df[order(df$puissanceMoyenne),]
```

	km	temps	vitesseMoyenne	puissanceMoyenne	bpm
5	9.27	19:10	29.0	121	143
4	17.61	36:11	29.2	125	144
2	4.84	9:42	30.2	133	146
3	1.02	1:57	30.8	141	139
1	1.24	4:01	19.1	160	134

Par défaut, l'ordre est est croissant, on peut le rendre décroissant en ajoutant l'argument `decreasing = T`

```
In [11]: df[order(df$vitesseMoyenne, decreasing = T),]
```

	km	temps	vitesseMoyenne	puissanceMoyenne	bpm
3	1.02	1:57	30.8	141	139
2	4.84	9:42	30.2	133	146
4	17.61	36:11	29.2	125	144
5	9.27	19:10	29.0	121	143
1	1.24	4:01	19.1	160	134

Toutefois, lorsque nous avons une base de données comportant un nombre plus important de variables, la syntaxe peut devenir plus compliquée et lourde d'écriture. Regardons un autre exemple:

```
In [3]: df_ass <-read.csv("https://raw.githubusercontent.com/nmeraihi/data/master/assurance.csv"
        head(df_ass)
```

numeropol	debut_pol	fin_pol	freq_paiement	⋯	cout6	cout7	nbsin	equipe
4	11-4-1996	10-4-1997	12	⋯	NA	NA	0	3
4	11-4-1997	10-4-1998	12	⋯	NA	NA	0	3
4	11-4-2002	17-7-2002	12	⋯	NA	NA	0	3
⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋮
4	11-4-2003	10-4-2004	12	⋯	NA	NA	0	3
12	3-5-1995	2-5-1996	1	⋯	NA	NA	0	3

Affichons notre base de données en ordre croissant sur le nombre de sinistres et le numéro de police;

```
In [15]: df_ass[order(df_ass$nbsin, df_ass$numeropol, decreasing = T),]
```

	numeropol	debut_pol	fin_pol	freq_paiement	···	cout6	cout7	nbsin	equipe
988	2006	13-6-1996	12-6-1997	12	···	NA	NA	2	3
902	1820	1-10-1996	30-9-1997	1	···	NA	NA	2	3
861	1733	10-5-1998	9-5-1999	1	···	NA	NA	2	3
⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋮
4	4	18-7-2002	10-4-2003	12	···	NA	NA	0	3
5	4	11-4-2003	10-4-2004	12	···	NA	NA	0	3

Dans le tableau affiché ci-haut, on voit bien que cette fonction ne nous permet pas d'appliquer un ordre croissant ou décroissant sur une variable précise

2.2 arrange

Maintenant, utilisons le paquet (*package*) `dplyr` qui nous permet de plus facilement d'appliquer un ordre quelconque sur une variable précise indépendamment des autres variables;

```
In [4]: library(dplyr, warn.conflicts = FALSE)
```

```
In [17]: arrange(df_ass, desc(nbsin), numeropol)
```

numeropol	debut_pol	fin_pol	freq_paiement	···	cout6	cout7	nbsin	equipe
71	15-2-1996	14-2-1997	1	···	NA	NA	2	3
79	20-11-1997	21-6-1998	12	···	NA	NA	2	3
116	4-9-1998	11-6-1999	1	···	NA	NA	2	3
⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋮
2036	14-3-2000	26-2-2001	12	···	NA	NA	0	3
2036	27-2-2001	13-3-2001	12	···	NA	NA	0	3

2.3 select

Ce paquet nous permet aussi de sélectionner des variables d'intérêt. Par exemple, dans notre `df_ass`, on désire seulement sélectionner les variables `numeropol`, `type_territoire` et `nbsin`

```
In [5]: select(df_ass, numeropol,type_territoire, nbsin)
```

numeropol	type_territoire	nbsin
4	Semi-urbain	0
4	Semi-urbain	0
4	Semi-urbain	0
⋮	⋮	⋮
2036	Semi-urbain	0
2036	Semi-urbain	1

2.4 filter

Afin de filtrer des données sur des observations d'intérêt. On peut utiliser la fonction de base de **R** `which`. Par exemple dans les données `Cars93` du package `MASS`, on voudrait extraire les véhicules ayant 8 cylindres. On voudrait également afficher que les deux variables `'Horsepower'` et `'Passengers'`

```
In [19]: library(MASS, warn.conflicts = F)
```

```
In [20]: Cars93[which(Cars93$Cylinders==8), c('Horsepower' , 'Passengers')]
```

	Horsepower	Passengers
10	200	6
11	295	5
18	170	6
⋮	⋮	⋮
48	278	5
52	210	6

Toutefois, la fonction `filter` de la librairie `dyplr` est plus flexible lorsqu'il s'agit d'appliquer des filtres plus complexes. Essayons le même exemple avec cette fonction;

```
In [21]: filter(Cars93, Cylinders==8)[c('Horsepower' , 'Passengers')]
```

Horsepower	Passengers
200	6
295	5
170	6
⋮	⋮
278	5
210	6

Si l'on cherche les médecins qui ont eu deux sinistres dans notre base de données `df_ass`;

```
In [22]: filter(df_ass, nbsin==2, type_prof=="Médecin")
```

numeropol	debut_pol	fin_pol	freq_paiement	⋯	cout6	cout7	nbsin	equipe
71	15-2-1996	14-2-1997	1	⋯	NA	NA	2	3
140	15-4-1995	14-4-1996	12	⋯	NA	NA	2	3
1820	1-10-1996	30-9-1997	1	⋯	NA	NA	2	3

2.5 mutate

Dans ce package, on trouve aussi la fonction `mutate` qui permet d'ajouter de nouvelles variables à notre df

```
In [23]: mutate(df, arrondi=round(df$vitesseMoyenne,0))
```

km	temps	vitesseMoyenne	puissanceMoyenne	bpm	arrondi
1.24	4:01	19.1	160	134	19
4.84	9:42	30.2	133	146	30
1.02	1:57	30.8	141	139	31
17.61	36:11	29.2	125	144	29
9.27	19:10	29.0	121	143	29

Ajoutons maintenant trois nouvlles variables;

```
In [24]: mutate(df, arrondi=round(df$vitesseMoyenne,0), segementStrava=paste("segment",1:5,sep =
```

km	temps	vitesseMoyenne	puissanceMoyenne	bpm	arrondi	segementStrava	arrondi_2
1.24	4:01	19.1	160	134	19	segment_1	9.5
4.84	9:42	30.2	133	146	30	segment_2	15.0
1.02	1:57	30.8	141	139	31	segment_3	15.5
17.61	36:11	29.2	125	144	29	segment_4	14.5
9.27	19:10	29.0	121	143	29	segment_5	14.5

2.6 summarize

La fonction `summarize` est très similaire à la fonction `mutate`. Toutefois, contrairement à `mutate`, la fonction `summarize` ne travaille pas sur une copie du df, mais elle crée un tout nouveau df avec les nouvelles variables.

```
In [25]: summarise(df,TotalKmParcour=sum(km))
```

TotalKmParcour
33.98

```
In [26]: summarise(df,TotalKmParcour=sum(km), vitesseMoyenne= mean(df$vitesseMoyenne), puissance
```

TotalKmParcour	vitesseMoyenne	puissanceMoyenne
33.98	27.66	136

On voit bien que l'écriture du code commence à être un peu plus compliquée lorsque nous avons plusieurs parenthèses dans notre fonction. Pour remédier à ce problème, nous verrons la notion de *piiping*;

2.7 L'opérateur Pipe: %>%

Avant d'aller plus loin, introduisons l'opérateur de *pipe*:%>%. **dplyr** importe cet opérateur d'une autre librairie (`magrittr`). Cet opérateur vous permet de diriger la sortie d'une fonction vers l'entrée d'une autre fonction. Au lieu d'imbriquer des fonctions (lecture de l'intérieur vers l'extérieur), l'idée de *piping* est de lire les fonctions de gauche à droite.

Crédit de l'image Pipes in R Tutorial For Beginners

Lorsque nous avons écrit:

```
In [6]: select(df_ass, numeropol, type_territoire, nbsin)
```

numeropol	type_territoire	nbsin
4	Semi-urbain	0
4	Semi-urbain	0
4	Semi-urbain	0
⋮	⋮	⋮
2036	Semi-urbain	0
2036	Semi-urbain	1

Si on lit ce que nous avons écrit précédemment de l'intérieur vert l'extérieur, en utilisant le *piping*, nous aurons ceci:

```
In [7]: df_ass %>%
            select (numeropol,type_territoire, nbsin)
```

numeropol	type_territoire	nbsin
4	Semi-urbain	0
4	Semi-urbain	0
4	Semi-urbain	0
⋮	⋮	⋮
2036	Semi-urbain	0
2036	Semi-urbain	1

ou;

```
In [8]: df_ass %>%
            select (numeropol,type_territoire, nbsin) %>%
            head
```

numeropol	type_territoire	nbsin
4	Semi-urbain	0
4	Semi-urbain	0
4	Semi-urbain	0
⋮	⋮	⋮
4	Semi-urbain	0
12	Semi-urbain	0

2.8 group_by

Nous pouvons aussi grouper les données comme nous le faisions dans `SAS` avec les `PROC SQL`

```
In [11]: df_ass$coutTot<-rowSums(df_ass[,c(19:25)], na.rm = T, dims = 1)
```

```
In [12]: df_ass
```

numeropol	debut_pol	fin_pol	freq_paiement	⋯	cout7	nbsin	equipe	coutTot
4	11-4-1996	10-4-1997	12	⋯	NA	0	3	0
4	11-4-1997	10-4-1998	12	⋯	NA	0	3	0
4	11-4-2002	17-7-2002	12	⋯	NA	0	3	0
⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋮
2036	27-2-2001	13-3-2001	12	⋯	NA	0	3	0.0
2036	14-3-2001	13-3-2002	12	⋯	NA	1	3	231051.8

```
In [13]: summarise(df_ass,TotalNbSin=sum(nbsin), TotCout= sum((coutTot), na.rm = T))
```

TotalNbSin	TotCout
156	1078791

Cherchons par exemple nombre de sinistres totaux ainsi que leurs coûts par territoire. En utilisant la syntaxe du *piping*, ça devient plus facile d'inclure plus de sous-groupes;

```
In [14]: df_ass %>%
             group_by(type_territoire) %>%
             summarise(TotalNbSin=sum(nbsin),
                       TotCout= sum((coutTot), na.rm = T)
                      )
```

type_territoire	TotalNbSin	TotCout
Rural	51	547105.01
Semi-urbain	80	471157.64
Urbain	25	60528.81

3 Jointure des bases des données

Dans cette section, nous allons joindre deux ou plusieurs df. Mais d'abord importons deux df afin illustrer quelques exemples;

```
In [15]: df_demo <-read.csv("https://raw.githubusercontent.com/nmeraihi/data/master/donnes_demo.
         df_demo
```

name	province	company	langue	date_naissance	agee
Shane Robinson	Nova Scotia	May Ltd	fr	1944-10-20	72
Courtney Nguyen	Saskatchewan	Foley, Moore and Mitchell	en	1985-12-09	31
Lori Washington	Yukon Territory	Robinson-Reyes	fr	1970-01-27	47
⋮	⋮	⋮	⋮	⋮	⋮
Heidi Freeman	Northwest Territories	Singh, Esparza and Santos	en	1951-06-07	65
Morgan Buchanan	Northwest Territories	Rollins Inc	fr	1971-07-31	45

```
In [16]: df_auto <-read.csv("https://raw.githubusercontent.com/nmeraihi/data/master/cars_info.cs
         df_auto
```

numeropol	marque_voiture	couleur_voiture	presence_alarme	license_plate
1	Autres	Autre	0	DW 3168
5	RENAULT	Autre	0	926 1RL
13	RENAULT	Autre	1	SOV 828
⋮	⋮	⋮	⋮	⋮
84	HONDA	Autre	0	CBV 102
91	BMW	Autre	1	UOR-0725

Dans ces deux df, nous avons une colonne en commun `numeropol`

```
In [17]: df_demo$numeropol
```

1. 1 2. 5 3. 13 4. 16 5. 22 6. 28 7. 29 8. 49 9. 53 10. 57 11. 59 12. 65 13. 67 14. 68 15. 69 16. 72 17. 78 18. 83 19. 84 20. 91

```
In [18]: df_auto$numeropol
```

1. 1 2. 5 3. 13 4. 16 5. 22 6. 22 7. 28 8. 29 9. 49 10. 53 11. 53 12. 57 13. 59 14. 65 15. 65 16. 67 17. 68 18. 69 19. 69 20. 72 21. 78 22. 83 23. 84 24. 84 25. 91

On peut voir l'index des lignes qui se trouvent dans les deux df

```
In [19]: match(df_demo$numeropol, df_auto$numeropol)
```

1. 1 2. 2 3. 3 4. 4 5. 5 6. 7 7. 8 8. 9 9. 10 10. 12 11. 13 12. 14 13. 16 14. 17 15. 18 16. 20 17. 21 18. 22 19. 23 20. 25

```
In [20]: df_demo$numeropol[match(df_demo$numeropol, df_auto$numeropol)]
```

1. 1 2. 5 3. 13 4. 16 5. 22 6. 29 7. 49 8. 53 9. 57 10. 65 11. 67 12. 68 13. 72 14. 78 15. 83 16. 91 17. <NA> 18. <NA> 19. <NA> 20. <NA>

On peut aussi faire un test logique sur la présence des observations du df_demo dans df_auto;

```
In [21]: df_demo$numeropol %in% df_auto$numeropol
```

1. TRUE 2. TRUE 3. TRUE 4. TRUE 5. TRUE 6. TRUE 7. TRUE 8. TRUE 9. TRUE 10. TRUE 11. TRUE 12. TRUE 13. TRUE 14. TRUE 15. TRUE 16. TRUE 17. TRUE 18. TRUE 19. TRUE 20. TRUE

ou le contraire maintenant

```
In [22]: df_auto$numeropol %in% df_demo$numeropol
```

1. TRUE 2. TRUE 3. TRUE 4. TRUE 5. TRUE 6. TRUE 7. TRUE 8. TRUE 9. TRUE 10. TRUE 11. TRUE 12. TRUE 13. TRUE 14. TRUE 15. TRUE 16. TRUE 17. TRUE 18. TRUE 19. TRUE 20. TRUE 21. TRUE 22. TRUE 23. TRUE 24. TRUE 25. TRUE

Dans ce cas toutes les variables se trouvent dans les deux df

```
In [23]: merge(df_demo,df_auto, by.x = "numeropol",  by.y = "numeropol") # x est le df_demo et y
```

numeropol	name	province	company	···	marque_voitu
1	Shane Robinson	Nova Scotia	May Ltd	···	Autres
5	Courtney Nguyen	Saskatchewan	Foley, Moore and Mitchell	···	RENAULT
13	Lori Washington	Yukon Territory	Robinson-Reyes	···	RENAULT
⋮	⋮	⋮	⋮	⋱	⋮
84	Heidi Freeman	Northwest Territories	Singh, Esparza and Santos	···	HONDA
91	Morgan Buchanan	Northwest Territories	Rollins Inc	···	BMW

Que serait-il arrivé si l'on n'avait pas spécifié les arguments `by.x = "numeropol",  by.y = "numeropol"`?

```
In [24]: merge(df_demo,df_auto)
```

numeropol	name	province	company	···	marque_voitu
1	Shane Robinson	Nova Scotia	May Ltd	···	Autres
5	Courtney Nguyen	Saskatchewan	Foley, Moore and Mitchell	···	RENAULT
13	Lori Washington	Yukon Territory	Robinson-Reyes	···	RENAULT
⋮	⋮	⋮	⋮	⋱	⋮
84	Heidi Freeman	Northwest Territories	Singh, Esparza and Santos	···	HONDA
91	Morgan Buchanan	Northwest Territories	Rollins Inc	···	BMW

Cela a bien fonctionné, car R a automatiquement trouvé les noms de colonnes communs au deux df;

Maintenant, changeons les noms de colonnes et voyons ce qui arrive

```
In [25]: names(df_auto)[names(df_auto)=="numeropol"] <- "auto_numpol"
```

Bien évidemment, cela crée une jointure croisée comme on l'avait vu dans les cours de SAS

```
In [26]: head(merge(df_demo,df_auto))
```

name	province	company	langue	⋯	marque_voiture	couleu
Shane Robinson	Nova Scotia	May Ltd	fr	⋯	Autres	Autre
Courtney Nguyen	Saskatchewan	Foley, Moore and Mitchell	en	⋯	Autres	Autre
Lori Washington	Yukon Territory	Robinson-Reyes	fr	⋯	Autres	Autre
⋮	⋮	⋮	⋮	⋱	⋮	⋮
Jeffrey Garcia	Nunavut	Berger-Thompson	en	⋯	Autres	Autre
Colleen Coleman	Saskatchewan	Simmons-Smith	en	⋯	Autres	Autre

On vient bien que dans la dernière colonne `license_plate`, nous obtenons la même observation ce qui est clairement une erreur;

Corrigeons le problème;

```
In [27]: head(merge(df_demo,df_auto, by.x = "numeropol",  by.y = "auto_numpol"))
```

numeropol	name	province	company	⋯	marque_voiture	c
1	Shane Robinson	Nova Scotia	May Ltd	⋯	Autres	A
5	Courtney Nguyen	Saskatchewan	Foley, Moore and Mitchell	⋯	RENAULT	A
13	Lori Washington	Yukon Territory	Robinson-Reyes	⋯	RENAULT	A
⋮	⋮	⋮	⋮	⋱	⋮	⋮
22	Jeffrey Garcia	Nunavut	Berger-Thompson	⋯	VOLKSWAGEN	A
22	Jeffrey Garcia	Nunavut	Berger-Thompson	⋯	VOLKSWAGEN	A

left_join

la fonction `left_join` prend toute l'information de gauche et l'information existante de la partie droite qui est basée sur le critère en commun

```
In [28]: x<-data.frame(nom=c("Gabriel", "Adel", "NM", "Mathieu", "Amine", "Mohamed"),
                       bureaux=c("5518", "4538", "5518", "5517", "4538", "4540"))
```

x

nom	bureaux
Gabriel	5518
Adel	4538
NM	5518
⋮	⋮
Amine	4538
Mohamed	4540

```
In [29]: y<-data.frame(nom=c("Gabriel", "Adel", "JP", "Mathieu", "Amine"),
                       diplome=c("M.Sc", "Ph.D", "Ph.D", "Ph.D", "Ph.D"))
```

y

nom	diplome
Gabriel	M.Sc
Adel	Ph.D
JP	Ph.D
Mathieu	Ph.D
Amine	Ph.D

```
In [30]: left_join(x,y,by = "nom")
```

```
Warning message:
‘‘Column `nom` joining factors with different levels, coercing to character vector’’
```

nom	bureaux	diplome
Gabriel	5518	M.Sc
Adel	4538	Ph.D
NM	5518	NA
⋮	⋮	⋮
Amine	4538	Ph.D
Mohamed	4540	NA

3.1 inner_join

Cette fonction permet de retourner **seulement** les éléments en commun des deux df

```
In [31]: inner_join(x,y,by = "nom")
```

```
Warning message:
‘‘Column `nom` joining factors with different levels, coercing to character vector’’
```

nom	bureaux	diplome
Gabriel	5518	M.Sc
Adel	4538	Ph.D
Mathieu	5517	Ph.D
Amine	4538	Ph.D

3.2 semi_join

Cette fonction retourne seulement les éléments du premier df qui se retrouve dans le deuxième df, sans nous retourner les éléments de ce dernier

```
In [ ]: semi_join(x,y,by = "nom")
```

3.3 anti_join

Cette fonction le contraire de la précédente

```
In [ ]: anti_join(x,y,by = "nom")
```