

# Formation-R-perfectionnement

Module Rappels et compléments sur R et RStudio

# Sommaire

- 1 Travailler avec R au ministère
- 2 Traitement de données avec {dplyr} et {tidyr}
- 3 Les différents objets R
- 4 Naviguer à l'intérieur des objets R (indexation)

## 0.1 Avant-propos

Ce diaporama de formation a été rédigé dans le but d'être le support visuel des formations dispensées au **SSM Agriculture**.

Ces formations s'adressent à des agents qui ont suivi [la formation R initiation][<https://ssm-agriculture.github.io/formation-R-initiation/>] ou qui sont familiers des outils de manipulation de données (dplyr).

## 0.2 Avant-propos

Elles sont données en présentiel sur une durée **de 3 jours**, les modules de cette formation sont ajustables suivants le choix des agents.

### Champ couvert par cette formation

**Ce support couvre les rappels et compléments sur R et l'environnement du SSM Agriculture.**

Pour information, les modules de la formation R-perfectionnement sont:

- 01 - Module Rappels
- 02 - Module Fonctions
- 03 - Module Cartes statiques et interactives
- 04 - Module Création de graphiques avec ggplot2
- 05 - Module Quarto
- 06 - Module Parquet
- 07 - Module Initiation à l'écriture d'applications Shiny

Ils sont orientés pour être utile aux agents du SSM Agriculture et se concentrent sur une utilisation de R via [RStudio](#) qui est mise à disposition des agents sur la plateforme interne Cerise basée sur RStudio Workbench.



MINISTÈRE  
DE L'AGRICULTURE  
ET DE LA SOUVERAINETÉ  
ALIMENTAIRE

*Liberté  
Égalité  
Fraternité*

# 1 Travailler avec R au ministère



# 1.1 Présentation

Logiciel de traitement de données et analyse statistique :

- offre un environnement interactif de développement statistique, analytique et graphique ;
- est doté d'un langage de programmation R ;
- permet d'accéder à des données, de les manipuler et les analyser ;
- S'interface avec les bases de données : Oracle, SYBASE, PostgreSQL, SQLITE...

Remplace SAS ou SPSS

Logiciel IDE : Integrated Development Environment



## 1.2 Comment travailler avec R au SSM Agriculture ?

Depuis la plate-forme CERISE :

- ⇒ PISTACHE : Pistache > Traitements statistiques et Diffusion > R > Migration SPSS et SAS vers R > Accès au WIKI Cerise - R > Accéder à Cerise

Cerise permet l'accès aux ressources déposées sur le serveur, la sauvegarde, le partage de code et le travail simultané. C'est l'usage recommandé.

En local sur son poste

Non préconisée

Existence d'une plate-forme CERISE de Préproduction :

<https://rstudio-pprd.agriculture.rie.gouv.fr>

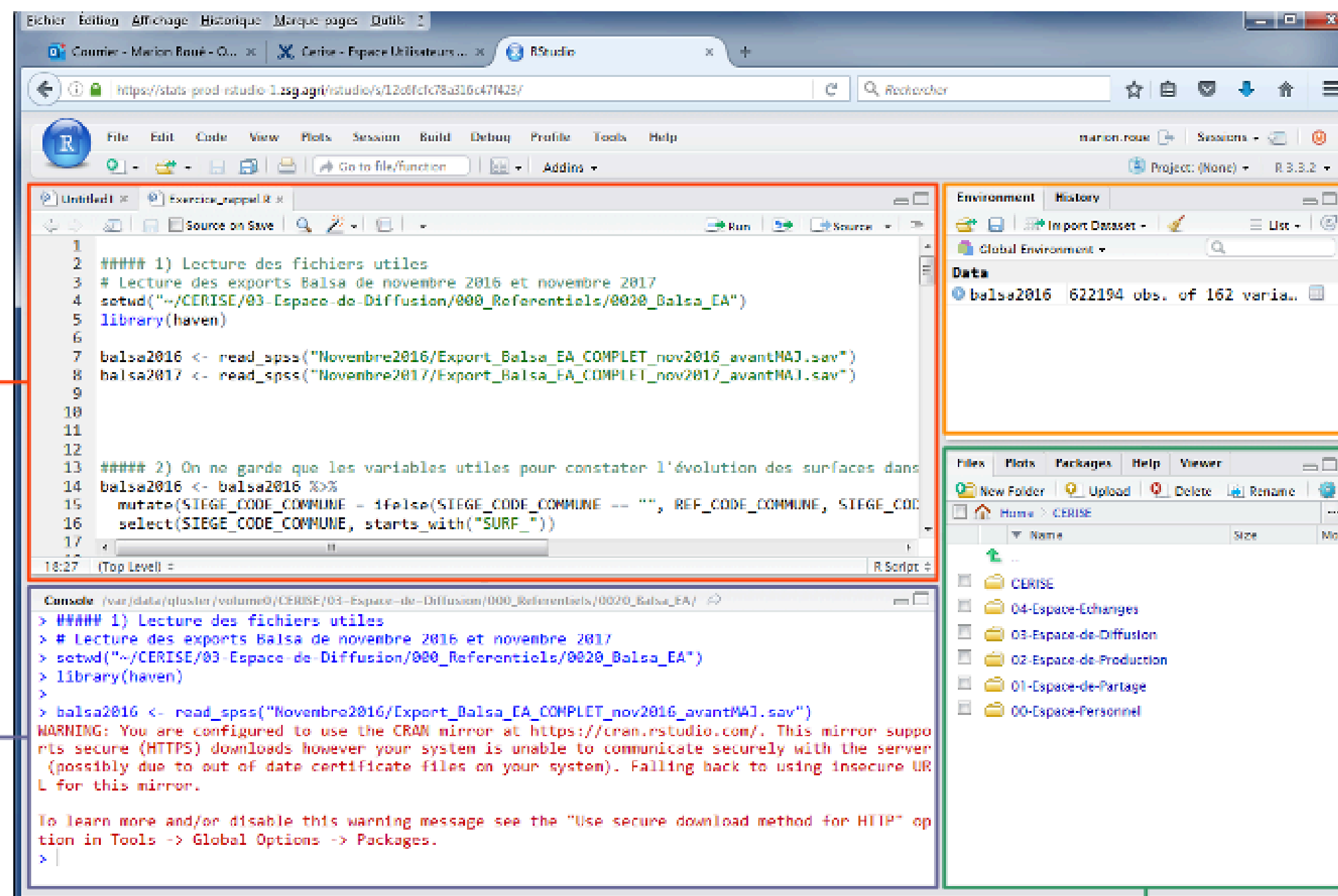
## 1.3 Fenêtres dans RSTUDIO

Vue des scripts et  
des données

Vue des éléments  
en mémoire et de  
l'historique des  
commandes

Console

Vue des  
répertoires, des  
graphiques, des  
packages et de  
l'aide





## 1.4 Organisation sous CERISE

Cerise est organisé en plusieurs répertoires :

- **00-Espace-Personnel** => espaces personnels des agents, **accessible par l'agent uniquement**
- **01-Espace-de-Partage** => lieu de partage général entre les différents acteurs
- **02-Espace-de-Production** => plateforme de stockage des données brutes collectées, ainsi que des fichiers de données et programmes issus des traitements statistiques réalisés par l'équipe projet
- **03-Espace-de-Diffusion** => mise à disposition au sein du SSM des données issues des traitements statistiques réalisés en amont => Ces deux derniers espaces sont découpés par opérations statistiques
- **04-Espace-Echanges** => stockage des fichiers de données à transmettre aux autres applications du SI CASSIS (par exemple Agreste) ainsi qu'aux SI des partenaires extérieurs

## 1.5 Généralités sur R avec RStudio

On utilise des packages (ensemble de fonctions) qu'il faut installer puis charger (sinon erreur!). Par exemple:

- `install.packages("rio")`
- `library(rio)` ou `require(rio)`

Pour obtenir de l'aide sur une fonction :

- `?maFonction`
- double-clic sur le nom de la fonction et F1

Quelques raccourcis claviers usuels:

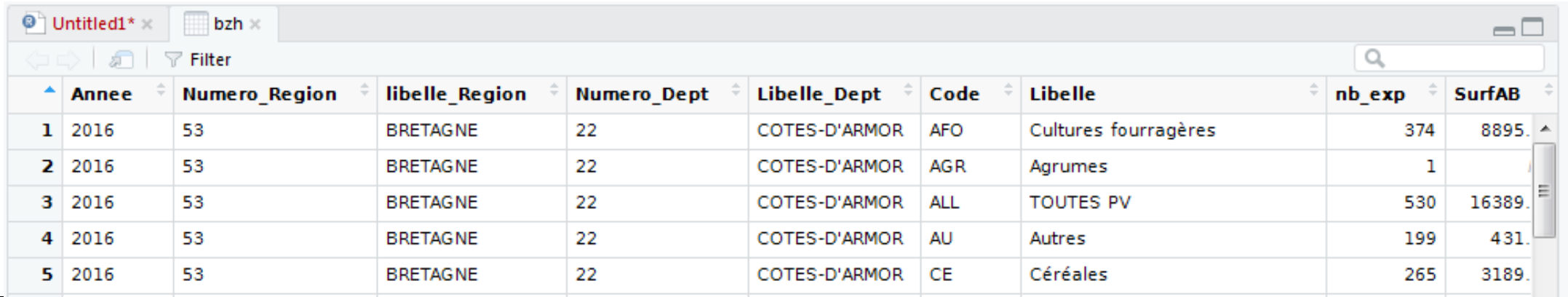
- Ctrl + Entrée : exécuter la ligne de code actuelle (curseur) ou une sélection
- Ctrl + Shift + C : mettre une ligne en commentaires
- Ctrl + Shift + M : Écrire un pipe `%>%` ou avec la nouvelle écriture `|>`

## 2 Traitement de données avec {dplyr} et {tidyr}

# 2.1 Les tables de données

Dans une table de données, on peut vouloir :

- sélectionner des lignes ou des colonnes particulières
- créer de nouvelles variables
- réaliser des calculs ou des opérations sur tout ou partie des observations
- faire une jointure avec une autre table
- transposer ou faire pivoter la table
- etc.



	Annee	Numero_Region	libelle_Region	Numero_Dept	Libelle_Dept	Code	Libelle	nb_exp	SurfAB
1	2016	53	BRETAGNE	22	COTES-D'ARMOR	AFO	Cultures fourragères	374	8895.
2	2016	53	BRETAGNE	22	COTES-D'ARMOR	AGR	Agrumes	1	
3	2016	53	BRETAGNE	22	COTES-D'ARMOR	ALL	TOUTES PV	530	16389.
4	2016	53	BRETAGNE	22	COTES-D'ARMOR	AU	Autres	199	431.
5	2016	53	BRETAGNE	22	COTES-D'ARMOR	CE	Céréales	265	3189.

## 2.2 Packages *dplyr* et *tidyr*

- Objectif de *dplyr* : rassembler dans un seul package les outils de manipulation de données les plus importants pour l'analyse des données  
⇒ ensemble de fonctions correspondant à un ensemble d'opérations élémentaires
- *dplyr* est un package, issu de la collection *tidyverse*, qui a pour objectif de faciliter la manipulation de tableaux.
- Deux principes pour les packages *dplyr* et *tidyr* :
  - **Usage de fonctions « verbe »** toutes construites sur le même principe : le 1er paramètre est la table de données sur laquelle on travaille. Les noms de variable n'ont pas besoin d'être en guillemets
  - **Usage de l'opérateur pipe** (`%>%` ou `|>`). Avec RStudio, le raccourci clavier pour cet opérateur est : **Ctrl + Shift + M**. Permet d'enchaîner les fonctions sans les emboîter:

```
1 maTable |> select(noms_des_colonnes) |> filter(conditions) |> summarise(total=sum(maVariable))
2 # Au lieu de :
3 summarise(filter(select(maTable,noms_des_colonnes),conditions),total=sum(maVariable))
```

## 2.3 Filtre et sélection dans une table

- Pour filtrer les lignes dans une table à l'aide de conditions logiques, on utilise la fonction `filter()`  
`maTable %>% filter(conditions)`
- Pour sélectionner des colonnes dans une table, on utilise la fonction `select()`  
`maTable %>% select(noms_des_colonnes)`

```
ma_table %>% filter(Profession == "Militaire")
```

```
ma_table %>% filter(Age < 30) %>% select(Nom_prenom)
```

row.names	Nom_prenom	Profession	Age	Code_postal_residence
1	Millot Nicolas	Manutentionnaire	26	29280
2	Roué Benjamin	Ingénieur	27	92130
3	Marzin Raymond	Militaire	56	83000

## 2.4 Sélection des colonnes dans une table

### Différentes façons de sélectionner :

- En nommant chaque variable une à une : `select(Nom_prenom, Age)`
- En indiquant un vecteur de variables consécutives : `select(Nom_prenom:Code_postal_residence)`
- En donnant les indices des variables : `select(1:3)`
- Grâce à de la sélection avec des opérateurs “helpers” (contient, commence par, finit par... :  
`select(contains("a"), starts_with("P"), ends_with("E"), where(is.numeric), last_col())`
- En enlevant celles dont on ne veut pas : `select(-Age)` ou `select(!(Age:Code_postal_residence))`

## 2.5 Création ou modification de variables dans une table (1/2)

Avec le package **dplyr**, on utilise la fonction `mutate()` qui permet de créer plusieurs variables à la fois :

- `table <- table %>% mutate(new_var1 = var1, newvar2 = var2)` → formule générique
- `table <- table %>% mutate(new_var1 = constante)` → création d'une constante
- `table <- table %>% mutate(new_var2 = fonction(var1))` → création à partir d'une formule
- `table <- table %>% mutate(new_var3 = var1 + var2)` → création à partir d'opérations arithmétiques
- `table <- table %>% mutate(new_var4 = vecteur1)` → création à partir de variables externes

Pour modifier une variable, on affecte la nouvelle valeur à une variable existante.



## 2.6 Création ou modification de variables dans une table (2/2)

- Paramètres utiles pour aller plus loin :
  - .after ou .before : pour placer la variable où on le souhaite (dernière colonne par défaut) (pour éviter un `relocate()` après)
  - .keep : pour choisir si on garde les variables qui servent à construire une nouvelle variable (pour éviter un `select(-var1, -var2)` après)

```
table ← table %>% mutate(new_var1 = var1+var2, .after=ident, .keep="unused")
```

Dans cet exemple, la nouvelle variable `new_var1` est placée après `ident`, `var1` et `var2` sont supprimés

## 2.7 La fonction `reframe` (1/4)

Avez-vous déjà rencontré ce message d'avertissement lorsque vous utilisez la fonction `summarise()` ?

Warning message:

Returning more (or less) than 1 row per ``summarise()`` group was deprecated in dplyr 1.1.0.

**i** Please use ``reframe()`` instead.

**i** When switching from ``summarise()`` to ``reframe()``, remember that ``reframe()`` always returns an ungrouped data frame and adjust accordingly.

Si oui, vous pouvez vous en débarrasser en utilisant la fonction `reframe()`.

## 2.8 La fonction `reframe` (2/4)

Par rapport à la fonction `summarise()`, la fonction `reframe()` (dplyr >= 1.1.0) est utile lorsqu'on souhaite **retourner plusieurs lignes par groupe** de manière explicite et propre.

Cela arrive par exemple lorsqu'on calcule des quantiles, des top N ...

La fonction `summarise()` est toujours valable et utilisable lorsqu'on souhaite **retourner une seule ligne par groupe** mais vous affichera un message d'avertissement lors des regroupements avec plusieurs lignes.

## 2.9 La fonction reframe (3/4)

Exemple avec `summarise()` :

```
1 # Cette utilisation de summarise déclenche un avertissement
2 iris %>%
3   group_by(Species) %>%
4   summarise(q = quantile(Petal.Length, probs = c(0.25, 0.5, 0.75)))
```

# Groups: Species [3]  
 Species q  
 <fct> <dbl>  
1 setosa 1.4  
2 setosa 1.5  
3 setosa 1.58  
4 versicolor 4  
5 versicolor 4.35  
6 versicolor 4.6  
7 virginica 5.1  
8 virginica 5.55  
9 virginica 5.88  
Warning message:  
Returning more (or less) than 1 row per `summarise()` group was

Exemple avec `reframe()` :

```
1 # Pas d'avertissement avec reframe
2 iris %>%
3   group_by(Species) %>%
4   reframe(q = quantile(Petal.Length, probs = c(0.25, 0.5, 0.75)))
```

# A tibble: 9 × 2  
 Species q  
 <fct> <dbl>  
1 setosa 1.4  
2 setosa 1.5  
3 setosa 1.58  
4 versicolor 4  
5 versicolor 4.35  
6 versicolor 4.6  
7 virginica 5.1  
8 virginica 5.55  
9 virginica 5.88

# 2.10 La fonction `reframe` (4/4)

Quelques remarques supplémentaires entre `summarise()` et `reframe()` :

- `reframe()` renvoie toujours un `data.frame` **non groupé** tandis que `summarise()` peut renvoyer un `data.frame` groupé ou non en fonction du choix fait dans l'argument `.groups`.

Utilisation de `.groups` avec `summarise()` :

Valeur	Comportement
"drop_last" (défaut)	Supprime le dernier niveau de regroupement
"drop"	Supprime tous les groupements (df non groupée)
"keep"	Conserve tous les niveaux de groupement
"rowwise"	Groupe ligne à ligne

- Comme la fonction `summarise()`, la fonction `reframe()` **accepte l'argument `.by`** pour éviter d'écrire un `group_by()` et **est compatible avec la fonction `across()`** (voir plus loin).

## 2.11 Fonctions utiles au traitement des chaînes de caractères

Problématique	Fonction R de base	Fonction du package stringr
Mettre x en majuscules	<code>toupper(x)</code>	<code>str_to_upper(x)</code>
Mettre x en minuscules	<code>tolower(x)</code>	<code>str_to_lower(x)</code>
Concaténer deux vecteurs	<code>paste(x, y, sep = " ")</code>	<code>str_c(x, y, sep = "")</code>
Connaître le nombre de caractères dans une chaîne	<code>nchar(x)</code> <i>Attention ! ≠ de length(x)</i>	<code>str_length(x)</code>
Extraire les caractères deb à fin dans une chaîne	<code>substr(x, deb, fin)</code>	<code>str_sub(x, deb, fin)</code>
Extraire les caractères à partir du deb <sup>ème</sup> dans une chaîne	<code>substring(x, deb)</code>	<code>str_sub(x, deb)</code>
Supprimer les espaces de début et de fin	<code>trimws(x)</code>	<code>str_trim(x)</code>
Rajouter des caractères en début ou en fin de chaîne pour respecter un format (utile code dep)	-	<code>str_pad(x, width = longueur_format, side = "left", pad = caractère_aajouter)</code>
Détecter la présence d'un motif dans une chaîne	<code>grepl(motif, x, ignore.case = F)</code>	<code>str_detect(x, motif)</code>
Remplacer un motif dans une chaîne	<code>gsub(motif, remplacement, x)</code>	<code>str_replace(x, motif, remplacement)</code>
Et bien d'autres encore !..		

## 2.12 Tri dans une table

- Pour trier une table selon une ou plusieurs variables, on utilise la fonction *arrange()*

```
maTable %>% arrange(variables_de_tri)
```

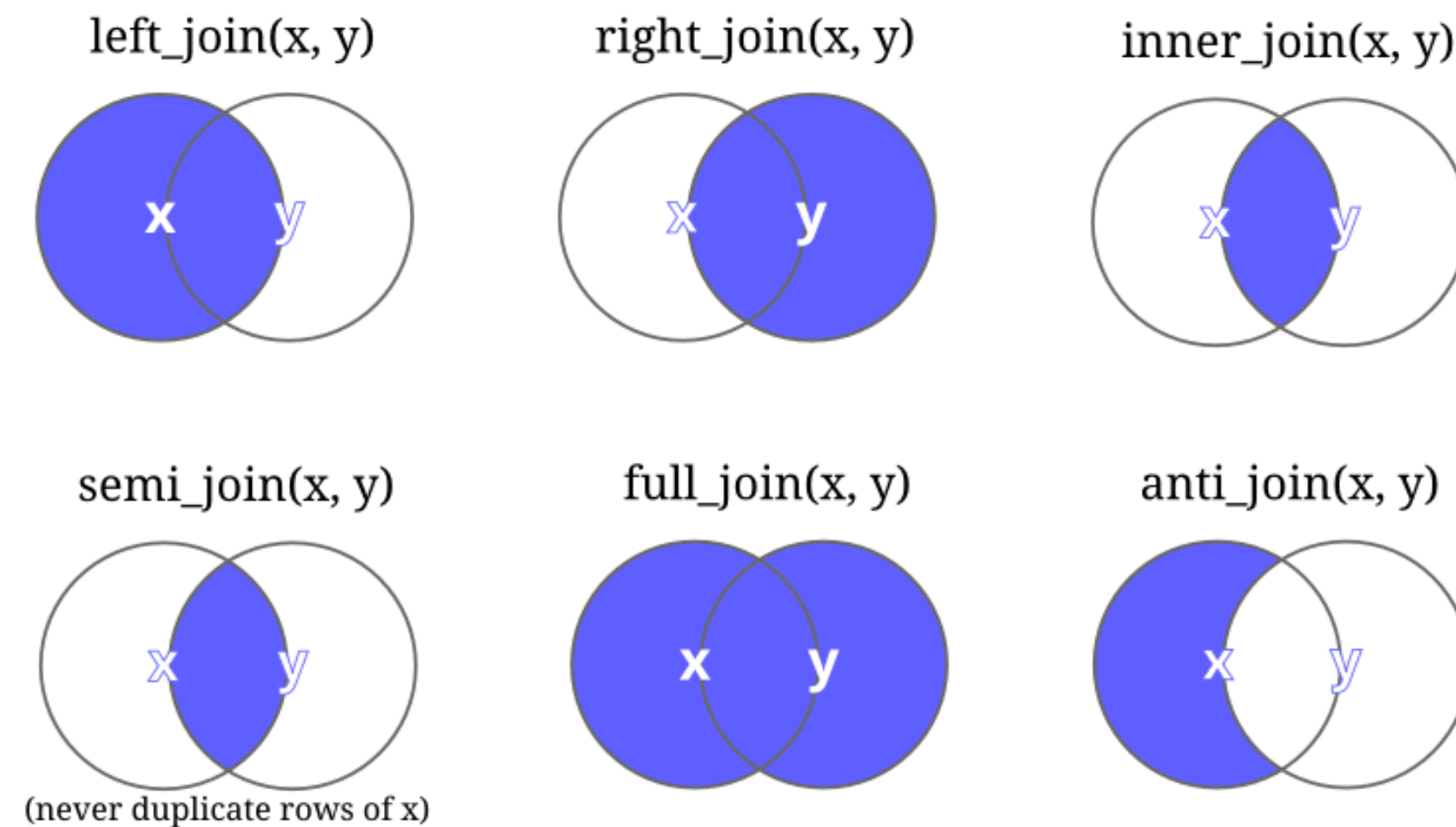
→ possibilité de trier selon plusieurs colonnes : les variables de tri doivent être séparées par une virgule.

→ on encadre les variables qu'il faut trier de façon décroissante par la fonction *desc()*.

```
maTable %>% arrange(var1, desc(var2))
```

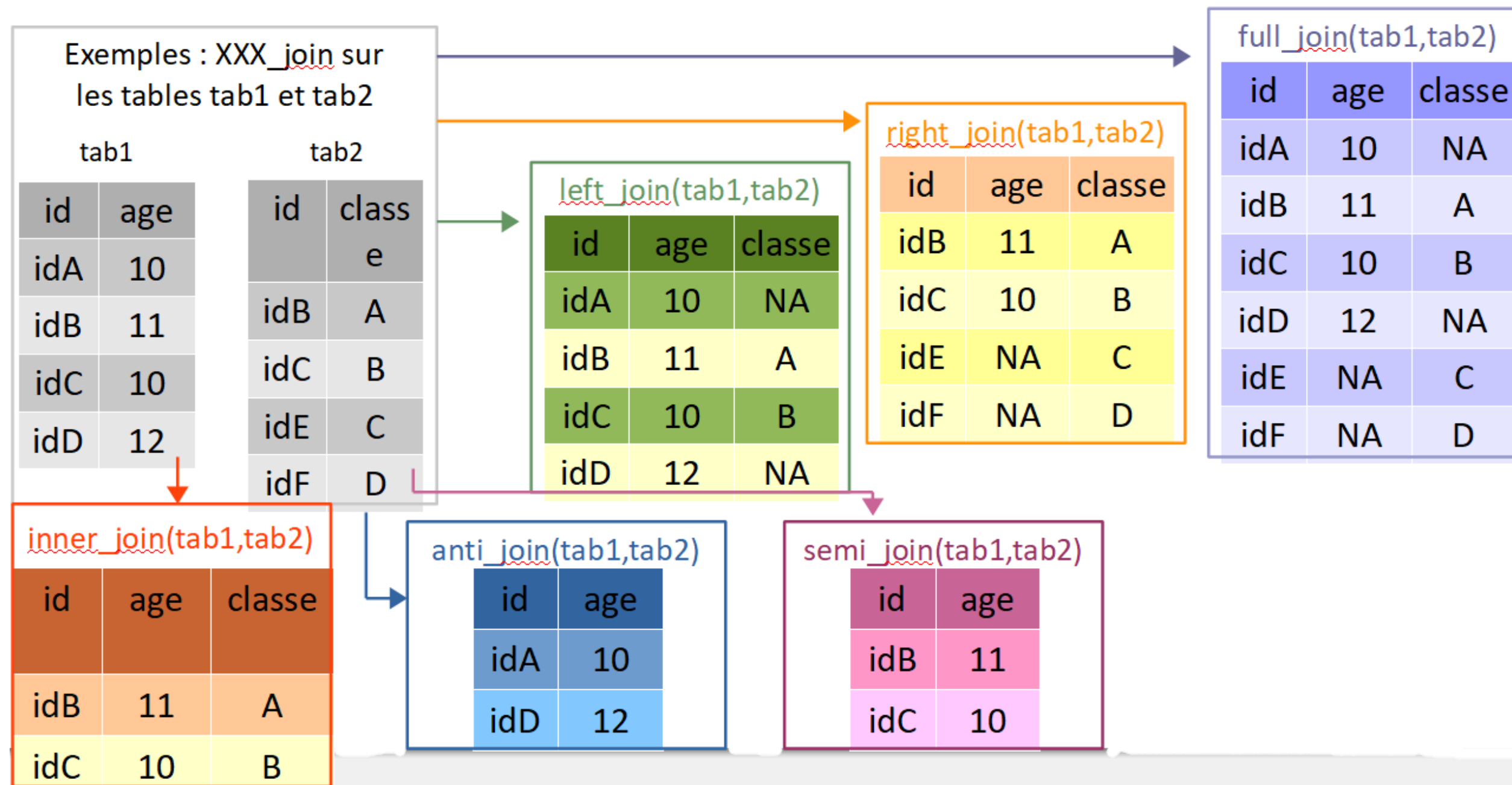
## 2.13 Fusion de tables

- Pour fusionner deux tables en utilisant une ou plusieurs variables de jointure, on utilise les fonctions *XXX\_join()*





## 2.14 Joindre des tables avec une variable de jointure



## 2.15 Fusion de tables

- Syntaxe :

```
maTable1 %>% full_join(maTable2, by = "variable_de_jointure")
```

```
maTable1 %>% full_join(maTable2, by = join_by(id1_tab1==id1_tab2))
```

- Il est possible de réaliser une jointure à l'aide de plusieurs variables identifiantes : l'argument `by` s'écrit alors `c("id1","id2")` ou `join_by(id1,id2)`
- Lorsque les variables de jointure ont des noms différents dans les deux tables, l'argument `by` prend comme paramètre un vecteur du type `c("id1_tab1" = "id1_tab2","id2_tab1" = "id2_tab2",...)`
- Si rien n'est précisé, la fusion se fait sur l'ensemble des variables portant le même nom dans les deux tables

## 2.16 Agrégation dans une table

- Pour résumer les données d'une table en une seule statistique, on utilise la fonction `summarise()`

```
maTable %>% summarise(fonctions_stat(variable))
```

- Pour agréger les données d'une table par groupe d'une ou plusieurs variables catégorielles on utilise la fonction `group_by()` avant la fonction `summarise()`

```
maTable %>% group_by(var_groupe)%>% summarise(fonctions_stat(variable))
```

```
maTable %>% summarise(fonctions_stat(variable), .by=var_groupe)
```

→ possibilité de calculer plusieurs statistiques en séparant les instructions par une virgule

→ on peut utiliser les fonctions statistiques de base telles que max, min, mean, median, sqrt, sd, n,...

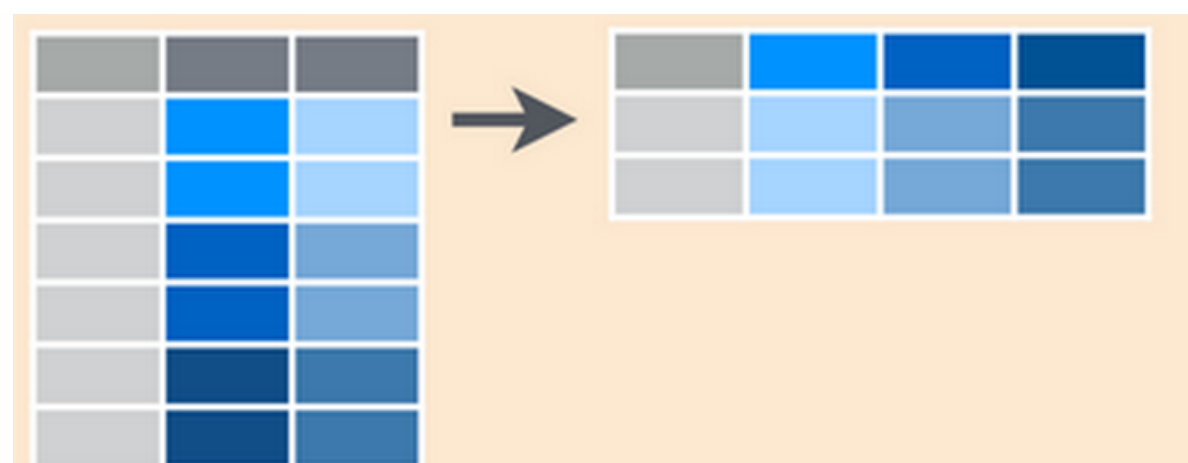
→ La fonction `ungroup()` permet de ne plus tenir compte du regroupement par la suite.

→ Voir également l'argument `.groups` de la fonction `summarise()` qui permet de gérer les niveaux de regroupement d'une table. Voir [cet exemple](#).

## 2.17 Transposition ou réorganisation de table (tidyr)

Pour réorganiser une table, c'est à dire passer des lignes en colonnes ou inversement, on utilise les fonctions `pivot_wider()` et `pivot_longer()` du package tidyr.

## 2.18 pivot\_wider()

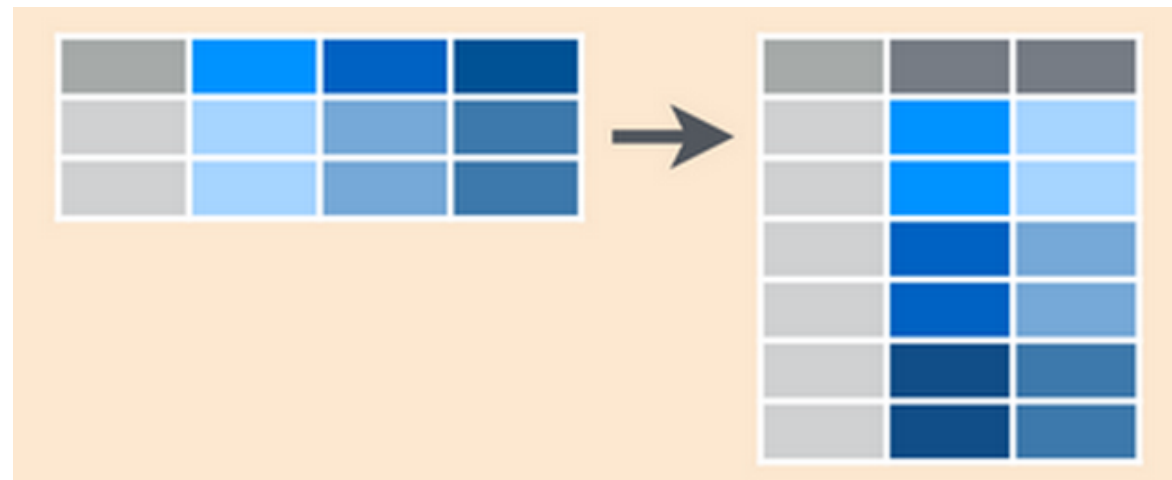


`maTable %>% pivot_wider(id_cols = col_restant, names_from = col_de_noms, values_from = col_de_valeurs)` les intitulés de colonne

- **col\_restant** = nom des variables qui seront conservées telles quelles dans la table pivotée (par défaut toutes celles qui ne sont pas indiquées dans col\_de\_noms ou col\_de\_valeurs)
- **col\_de\_noms** = nom de la variable dont les modalités deviendront les intitulés de colonne
- **col\_de\_valeurs** = nom de la variable à utiliser pour remplir les colonnes

`maTable %>% pivot_wider(col_de_noms, col_de_valeurs)`

## 2.19 pivot\_longer()



```
maTable %>% pivot_longer( cols = variables_apivoter , names_to = "indicateur", values_to = "valeur")
```

- **variables\_a\_pivoter** = nom de l'ensemble des variables à représenter
- **names\_to** = nom de la première colonne à créer
- **values\_to** = nom de la deuxième colonne à créer

## 2.20 Exemples sur les pivots

Ann	Mois	mesure	temperature
2020	janvier	min	- 10
2020	janvier	max	12
2020	janvier	moy	3
2020	février	min	- 6
2020	février	max	14
2020	février	moy	5

*pivot\_wider()*

*pivot\_longer()*

Ann	Mois	min	max	moy
2020	janvier	- 10	12	3
2020	février	- 6	14	5

```
tab2 <- pivot_wider( tab1, names_from = mesure, values_from = temperature)
tab1 <- pivot_longer(tab2, cols = c(min, max, moy), names_to = "mesure", values_to =
"temperature")
```

## 2.21 Autres fonctions utiles de dplyr et tidyr

- `case_when()` : en complément d'un mutate, pour modifier des valeurs ou modalités selon certaines conditions
- `slice()` et dérivés: une autre façon de filtrer les lignes (ex : n plus grandes/petites valeurs d'une variable)
- `distinct()` : gérer des doublons ou connaître les modalités d'une ou d'un groupe de variables
- `rename()` : renommer une variable (nouveau\_nom=ancien\_nom)
- `relocate()` : déplacer une variable (avec .after et .before)
- `count()` : compter le nombre d'observation pour chaque/un croisement de modalités
- `pull()` : pour extraire un vecteur à partir d'un data.frame
- `replace_na()` : remplacer toutes les valeurs manquantes NA par une valeur donnée ({tidyr})
- `drop_na()` : supprimer des lignes qui comportent des NA ({tidyr})



## 2.22 Lecture ou écriture de fichiers (1/2)

Version courte: le package {rio} permet avec une seule fonction d'importer ou exporter quasiment tout type de fichier (.csv, .xlsx, .ods, .rds, .sas7bdat, .parquet, etc.)

Syntaxe de base :

- Import simple : `maTable <- import("V:/adresseMaTable/monFichierAImporter.csv")`
- Export simple : `export(maTable, "V:/adresseMaTable/monFichierAExporter.xlsx")`

{rio} est un package qui mobilise d'autres packages dédiés à certains types de fichier, (cf. [aide de rio](#) pour les connaître et spécifier certains paramètres) Ex: importer un seul onglet.

- `MaTable <- import("V:/adresseMaTable/monFichierAImporter.xlsx", sheet="monOnglet")`  
(paramètres de `readxl::read_excel()`)

## 2.23 Lecture ou écriture de fichiers (2/2)

L'alternative est de passer par les fonctions des packages dédiés à chaque type de fichier:

Type	Lecture	Ecriture
Texte	<code>reader::read_delim()</code>	<code>reader::write_delim()</code>
RDS	<code>readr::read_rds()</code>	<code>readr::write_rds()</code>
XLS	les fonctions du package {readxl}	les fonctions du package {openxlsx}
SAS ou SPSS	<code>haven::read_sas()</code> et <code>haven::read_spss()</code>	Exporter au format texte puis lecture dans SAS ou SPSS (sans objet désormais)
ODS	<code>readODS::read_ods()</code>	<code>readODS::write_ods()</code>
Parquet	Les packages <code>arrow</code> et <code>duckdb</code>	Le package <code>arrow</code>

## 2.24 Base de données avec R (avancé)

- R offre la possibilité d'effectuer des requêtes SQL sur des bases de données externes (il faut donc connaître la syntaxe SQL).
- Des informations pour la connexion sont nécessaires :

```
library(RPostgres)
```

```
cnx <- dbConnect(Postgres(), dbname = "sirene", port = 5432, host = "00.11.22.33", user =  
"nom_user", password = "mdp")
```

- Il suffit ensuite d'écrire les requêtes normalement à l'aide de la fonction dbGetQuery :

```
dbGetQuery(cnx, "SELECT * FROM TABLE")
```

## 2.25 Exercices

Exercices



MINISTÈRE  
DE L'AGRICULTURE  
ET DE LA SOUVERAINETÉ  
ALIMENTAIRE

*Liberté  
Égalité  
Fraternité*

# 3 Les différents objets R



## 3.1 Pourquoi faire du R de base ?

*NB : on peut s'en sortir en manipulation de données sans connaître R de base grâce aux packages du tidyverse (mais ça aide quand même).*

- Aide à mieux se servir de dplyr
- Déboguer des erreurs
- Solutions alternatives à dplyr dans certains cas
- **dplyr** moins adapté à la programmation et aux fonctions (problème de la *tidy evaluation*)
- Mieux comprendre le fonctionnement de R

## 3.2 Les vecteurs : la brique de base

Objet élémentaire de R, une dimension (beaucoup de fonctions sont vectorisées)

- *NB : les colonnes (variables) de nos tableaux de données sont des vecteurs qui ont un nom (le nom de la variable)*
- Ensemble de valeurs, appelées éléments, **de même nature** (nombres entiers ou décimaux, chaînes de caractères, booléens...). Si ce n'est pas le cas, R va convertir. On peut donner un nom à chaque élément.
- Création de vecteurs avec la fonction `c()` ou `:` pour une suite de valeurs numériques :
  - `x <- c(TRUE, TRUE, FALSE) ; is.logical(x)`
  - `x <- c(a=8.04, b=18.01, c=11.05, d=14.05) ; is.numeric(x)`
  - `x <- c("Chaîne1", "804", "Chaîne3") ; is.character(x)`
  - `x <- 1:5`
  - `x <- rep(NA, 10)`
  - `x <- seq(0, 1, 0.1)` (par exemple pour calculer des déciles)
- Fonctions utiles sur les vecteurs : `length()`, `class()`, `sort()`, `unique()`, `which()`, `as.<class>()`...

## 3.3 Les vecteurs : exemples de code (1/2)

1er exemple :

```
1 # Création du vecteur y
2 y <- c(rep("a",5),rep("b",10),rep("c",5));
3 y
4 > [1] "a" "a" "a" "a" "a" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "b" "c" "c" "c" "c" "c"
5 as.factor(y)
6 > [1] a a a a a b b b b b b b b b b c c c c c
7 Levels: a b c
```

Les fonctions `which()` et `unique()` :

```
1 which(y=="a") # indices des éléments qui remplissent la condition
2 > [1] 1 2 3 4 5
3 unique(y) # enlève les doublons
4 > [1] "a" "b" "c"
```

L'opérateur `:` et la fonction `seq()` :

```
1 # Création d'un vecteur de 1 à 8
2 1:8
3 > [1] 1 2 3 4 5 6 7 8
4 # Création d'un vecteur de 0 à 1 avec un pas de 0.1
5 seq(0,1,0.1)
6 > [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```



## 3.4 Les vecteurs : exemples de code (2/2)

- Exemple sur le type des vecteurs :

```
1 x <- c(1, "two", 3.0, TRUE, "29") # un vecteur ne peut être que d'une sorte
2 > [1] "1"      "two"   "3"      "TRUE"  "29"
3 class(x)
4 > [1] "character"
```

- Tri et renversement des éléments d'un vecteur :

```
1 # Création du vecteur x
2 x <- sample(letters[1:10]);
3 # Tri des éléments du vecteur x
4 sort(x)
5 > [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
6 # Renverse l'ordre des éléments du vecteur x
7 rev(sort(x))
8 > [1] "j" "i" "h" "g" "f" "e" "d" "c" "b" "a"
```

## 3.5 Les data.frame : tables de données

Le **data.frame** est l'objet que l'on manipule le plus habituellement (tables du RA, etc.). C'est un tableau de données avec des lignes et des colonnes

- Formellement, un **data.frame** est composé de vecteurs nommés de tailles égales mais pouvant être de types différents (caractère, facteurs, numérique ...).
- Création avec les fonctions `data.frame()`:

```
1 x <- data.frame(nom=c("Yaël", "Luan", "Jade"), # 1er élément : un vecteur « nom » de trois chaînes de caractères`
2               sexe = factor(c("H", "F", "F")), # 2e élément : un vecteur « sexe » de trois chaînes de caractères (en facteurs)
3               age = c(5, 1, 3))                # 3e élément : un vecteur « age » de trois numériques
4 >
5   nom sexe age
6 1 Yaël   H   5
7 2 Luan   F   1
8 3 Jade   F   3
```

## 3.6 Fonctions utiles pour explorer des data.frame

- `str()` donne la structure de la table (ce que l'on voit dans l'environnement)
- `names()` le vecteur des noms de colonnes, `rownames()` le vecteur des noms de lignes (indices par défaut)
- `head()` et `tail()` affichent respectivement, par défaut, les 6 premières et les 6 dernières lignes de la table
- `dim()` renvoie un vecteur de 2 éléments composé du nombre de lignes et du nombre de colonnes de la table
- `summary()` renvoie un résumé du data.frame (utile pour explorer un jeu de données)

# 3.7 Data.frame : exemples de code (1/3)

```
1 maTable <- data.frame(  
2   cat=rep(letters[1:2],5),           # 5 Répétitions des lettres 'a' et 'b'  
3   var1= runif(10,max = 10),         # 10 valeurs aléatoires uniformément distribuées entre 0 et 10  
4   var2= rnorm(10,sd = 100),         # 10 valeurs tirées d'une distrib; normale avec une moy; de 0 et un sd de 100  
5   var3=c(rep(NA,3),sample(1:10,7,T)), # 3 valeurs NA suivies de 7 valeurs aléatoires entre 1 et 10  
6   fact1=factor(rep(LETTERS[1:5],2)), # un facteur composé de 2 répétitions des lettres 'A', 'B', 'C', 'D' et 'E'  
7   majuscule = LETTERS[1:10],        # les 10 premières lettres majuscules de l'alphabet  
8   logique=sample(c(T,F),10,T))      # 10 valeurs booléennes aléatoires
```

```
1 > maTable  
2   cat      var1      var2 var3 fact1 majuscule logique  
3 1    a 8.625657  40.51943  NA    A      A    FALSE  
4 2    b 6.394318 110.42277  NA    B      B     TRUE  
5 3    a 7.351903  57.75785  NA    C      C     TRUE  
6 4    b 7.222472 -39.86580   4    D      D     TRUE  
7 5    a 7.297473 -186.24480   7    E      E    FALSE  
8 6    b 2.516919 -39.21729  10    A      F    FALSE  
9 7    a 0.143648 101.97522   9    B      G    FALSE  
10 8    b 4.199952 -73.90209   8    C      H     TRUE  
11 9    a 4.149988  15.82927  10    D      I    FALSE  
12 10   b 8.856342  56.02326   1    E      J    FALSE
```

# 3.8 Data.frame : exemples de code (2/3)

```
1 str(maTable)
2 > 'data.frame': 10 obs. of 7 variables:
3 $ cat      : chr  "a" "b" "a" "b" ...
4 $ var1     : num  8.43 8.77 8.71 9.88 3.02 ...
5 $ var2     : num -205.9 -87.3 -18.5 -261.6 -32 ...
6 $ var3     : int   NA NA NA 2 6 1 1 9 10 2
7 $ fact1    : Factor w/ 5 levels "A","B","C","D",...: 1 2 3 4 5 1 2 3 4 5
8 $ majuscule: chr   "A" "B" "C" "D" ...
9 $ logique  : logi  FALSE TRUE TRUE TRUE FALSE TRUE ...
```

```
1 summary(maTable) # noter la différence entre les vecteurs caractères et facteurs
2
3 > cat      var1      var2      var3      fact1 majuscule      logique
4 Length:10    Min.    :3.020    Min.    :-261.589    Min.    : 1.000    A:2    Length:10    Mode :logical
5 Class :character 1st Qu.:5.386    1st Qu.: -73.500    1st Qu.: 1.500    B:2    Class :character FALSE:5
6 Mode  :character Median :7.979    Median :  -9.719    Median : 2.000    C:2    Mode  :character  TRUE :5
7              Mean  :7.012    Mean  : -38.531    Mean   : 4.429    D:2
8              3rd Qu.:8.651    3rd Qu.: 44.160    3rd Qu.: 7.500    E:2
9              Max.   :9.875    Max.   : 75.879    Max.   :10.000
10             NA's    :3
```

## 3.9 Data.frame : exemples de code (3/3)

```
1 length(maTable) # pourquoi 7 ?
2 > [1] 7
3 dim(maTable)
4 > [1] 10 7
5 names(maTable)
6 [1] "cat"      "var1"      "var2"      "var3"      "fact1"      "majuscule" "logique"
```



## 3.10 Les listes (un objet fourre-tout)

La **liste** est un objet apparemment moins usuel quand on débute avec `dplyr`, mais qui peut être très utile notamment à l'intérieur de fonctions (across), pour faire des boucles (purrr) ou bien quand on doit ranger dans un unique objet, des objets de toute sorte.

- Une **liste** est composée d'objets de types et tailles différents (peut-être très utile en sortie de fonction).
- Création avec la fonction `list()` :

```
1 x <- list(c(TRUE, TRUE, FALSE),      # 1er élément : un vecteur de trois booléens
2          c(8.04, 18.01, 11.05, 14.05), # 2e élément : un vecteur de quatre numériques
3          c("804"))                   # 3e élément : un vecteur d'une chaîne de caractères
```

*NB : un data.frame est une liste de vecteurs de même taille.*

## 3.11 Listes : exemples de code (1/2)

```

1 # En nommant chaque élément de la liste
2 list(mesLettres=letters,
3       monDF = data_frame(letters,LETTERS),
4       ma2eListe=list("toto",1:10))
5
6 # > $mesLettres
7 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
8
9 $monDF
10 # A tibble: 26 × 2
11   letters LETTERS
12   <chr>    <chr>
13 1 a      A
14 2 b      B
15 3 c      C
16 # i 24 more rows
17 # i Use `print(n = ...)` to see more rows

```

```

1 $ma2eListe
2 $ma2eListe[[1]]
3 [1] "toto"
4
5 $ma2eListe[[2]]
6 [1] 1 2 3 4 5 6 7 8 9 10

```



## 3.12 Listes : exemples de code (2/2)

```
1 # Sans donner de noms
2 liste1 <- list(1:10)
3
4 liste2 <- list(letters)
5
6 # Concaténation des 2 listes
7 c(liste1, liste2)
8
9 >
10 [[1]]
11  [1]  1  2  3  4  5  6  7  8  9 10
12
13 [[2]]
14  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

## 3.13 Les matrices

Objets moins utilisés en traitement de données usuel, plus en statistiques.

- Une matrice est composée d'un ou de plusieurs **éléments de mêmes types**, organisés en **deux dimensions**.
- Création de matrice avec la fonction `matrix()` :

```
1 # Création d'une matrice de numériques avec deux lignes et trois colonnes, remplissage par colonne.`
2 exemple_matrice <- matrix(c(1,2,3,4,5,6), nrow = 2, ncol = 3)
3
4 #      [,1] [,2] [,3]
5 # [1,]    1    3    5
6 # [2,]    2    4    6
7
8 #Transposée
9 t(exemple_matrice)
10 #      [,1] [,2]
11 # [1,]    1    2
12 # [2,]    3    4
13 # [3,]    5    6
```

- Fonctions utiles : `dim()`, `nrow()`, `ncol()`, `t()`...



# 4 Naviguer à l'intérieur des objets R (indexation)



## 4.1 Pour quoi faire?

- Extraire un élément d'un objet (ex : tous les nombres supérieurs à 10)
- Remplacer un élément par un autre (ex : remplacer toutes les chaînes "Bov." par "Bovins")
- Ajouter un élément (ex : une nouvelle variable à un data.frame)

```
1 nombres <- 1:20
2 nombres[nombres>10]
3
4 # > [1] 11 12 13 14 15 16 17 18 19 20
5
6 mes_lettres <- letters[1:5]
7 mes_lettres[3] <- "g"
8 mes_lettres
9
10 # > [1] "a" "b" "g" "d" "e"
11
12 mon_df <- data.frame(letters, LETTERS)
13 mon_df["concat"] <- paste0(letters, LETTERS)
14 mon_df[1,]
15
16 # >   letters LETTERS concat
17 1      a      A      aA
```

## 4.2 Vecteur, indexation avec les crochets:

- Indexation par position :

```
x <- c("Marion", "Benjamin", "Hélène", "Raymond"))
```

```
x[1] renvoie "Marion"
```

```
x[c(1, 2)] renvoie le vecteur c("Marion", "Benjamin")
```

- Indexation par condition logique :

```
x <- c(8.04, 18.01, 11.05, 14.05)
```

```
x[x < 10] renvoie 8.04
```

```
x[x > 10] renvoie le vecteur c(18.01, 11.05, 14.05)
```

```
x[x > 10 & x < 15] renvoie le vecteur c(11.05, 14.05)
```

## 4.3 Data.frame (1/3) : principes de navigation

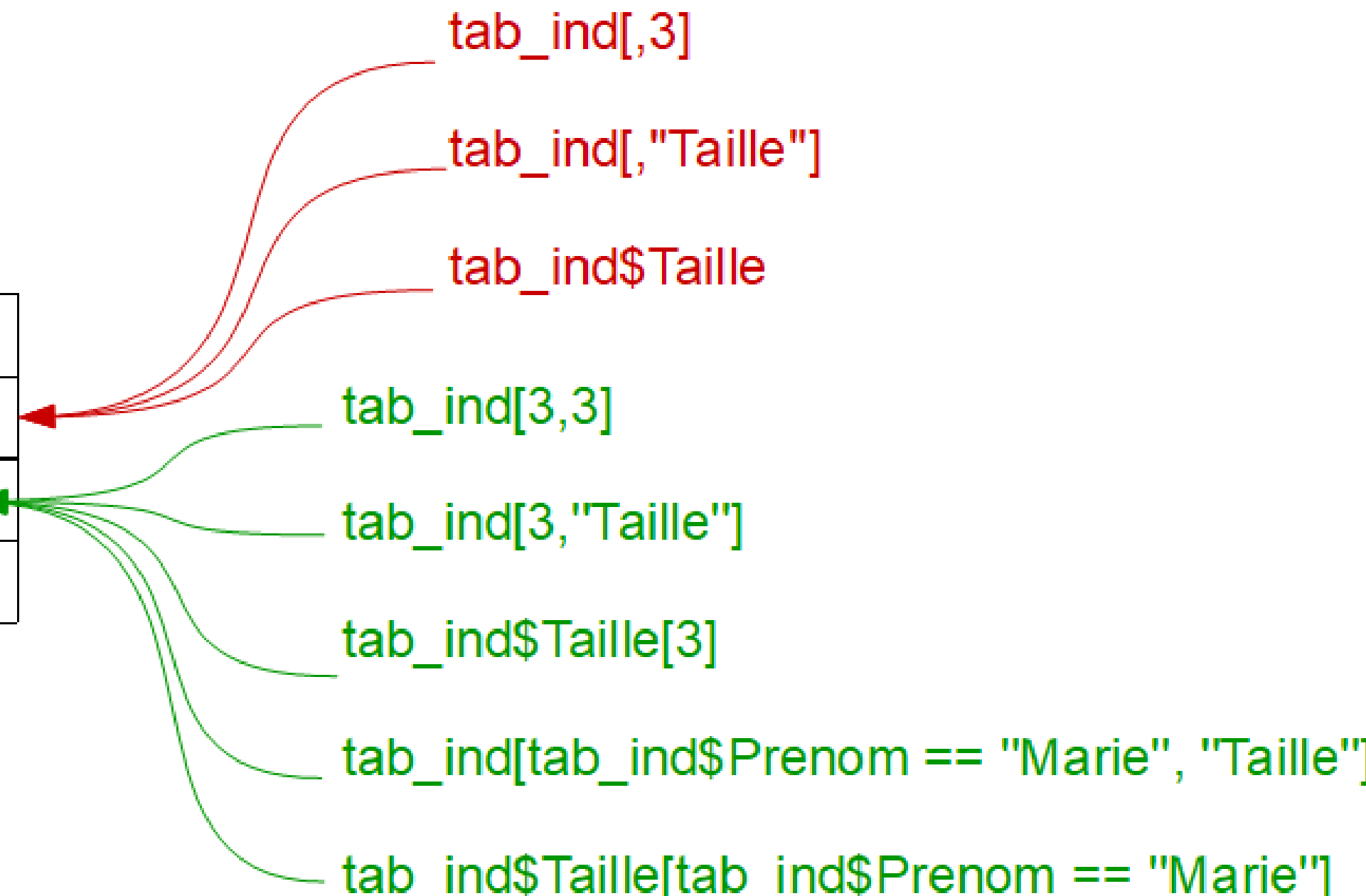
- Indexation **par position** : [numéro de ligne, numéro de colonne]
- Indexation **par nom** : les vecteurs d'une data.frame sont nommés et accessibles directement dans les crochets ou via l'opérateur `$`
- Les moyens d'indexation peuvent se combiner et permettre de filtrer les colonnes et les lignes (équivalent de `select()` et `filter()`)

→ Il y a donc une multitude de façons d'accéder à une valeur dans un data.frame !

## 4.4 Data.frame (2/3) : illustration

Tab\_ind

Prenom	Sexe	Taille
Pierre	H	185
Anne	F	167
Marie	F	171
Charles	H	175



tab\_ind[,3]

tab\_ind[,\"Taille\"]

tab\_ind\$Taille

tab\_ind[3,3]

tab\_ind[3,\"Taille\"]

tab\_ind\$Taille[3]

tab\_ind[tabs\_ind\$Prenom == \"Marie\", \"Taille\"]

tab\_ind\$Taille[tabs\_ind\$Prenom == \"Marie\"]

## 4.5 Data.frame (3/3) : exemple de code

Quelques lignes de code utiles en R de base :

```
1
2 mon_df <- data.frame(letters,LETTERS) # création du data.frame avec les lettres de l'alphabet
3 mon_df[1,] # un data.frame avec uniquement la 1ère ligne
4 mon_df[,1] # un vecteur avec la 1ère colonne,
5 mon_df["letters"] # attention si pas de virgule, renvoie un data.frame d'une seule colonne
6 mon_df[["letters"]] # pour extraire le vecteur, il faut mettre deux crochets
7 mon_df$letters[1:4] # avec le symbole $ on a aussi un vecteur qu'on peut indexer aussi (4 premières valeurs)
```



## 4.6 Création ou modification de variables dans un data.frame en R de base

Avec R de base, on utilise l'opérateur d'affectation :

- `table$new_var <- variable` → formule générique
- `table$new_var <- valeur_constante` → création d'une constante
- `table$new_var <- fonction(table$var1)` → création à partir d'une formule
- `table$new_var <- table$var1 + table$var2` → création à partir d'opérations
- `table$new_var <- vecteur1` OU `table1$new_var <- table2$var1` → création à partir de variables externes
- `table$new_var <- ifelse(table$varref < x, val1, val2)` OU `table$new_var <- case_when(table$varref < x ~ val1, TRUE ~ val2)` → traitement conditionnel

## 4.7 Listes (1/2)

Indexation par position :

```
x <- list(c(TRUE, TRUE, FALSE), c(8.04, 18.01, 11.05, 14.05), "804")
```

=> `x[[2]]` renvoie le deuxième vecteur de la liste x soit `c(8.04, 18.01, 11.05, 14.05)`

=> `x[[2]][1]` renvoie le premier élément du deuxième vecteur de la liste x soit `8.04`

### Note

Attention :

`x[2]` avec de simples crochets, renvoie une liste composée d'un élément : le 2e vecteur de la liste x

## 4.8 Listes (2/2)

Indexation par nom :

```
x <- list(grp = c("Auguste", "Justine", "Sébastien", "Anne-Marie"), prt = c("Hélène",  
"Raymond"), enf = c("Marion", "Benjamin"))
```

=> `x$grp` renvoie le vecteur nommé “grp” de la liste x soit `c("Auguste", "Justine", "Sébastien", "Anne-Marie")`

=> `x$enf[1]` renvoie le premier élément du vecteur nommé “enf” de la liste x soit `"Marion"`

Possibilité d'utiliser la fonction **`pluck()`** du package {purrr} (+ pratique pour éviter de s'y perdre)

```
x <- list(c(TRUE, TRUE, FALSE), c(8.04, 18.01, 11.05, 14.05), "804")
```

=> `pluck(x, 2)` renvoie le deuxième vecteur de la liste x soit `c(8.04, 18.01, 11.05, 14.05)`

=> `pluck(x, 2, 1)` renvoie le premier élément du deuxième vecteur de la liste x soit `8.04`

## 4.9 Matrices, indexation avec les crochets:

Création d'une matrice :

```
x <- matrix(c("Marion", "Benjamin", "Hélène", "Raymond"), nrow = 2, ncol = 2)
```

```
1      [,1]      [,2]  
2 [1,] "Marion"  "Hélène"  
3 [2,] "Benjamin" "Raymond"
```

Indexation **par position** :

=> `x[1, ]` renvoie le vecteur `c("Marion", "Hélène")`

=> `x[, 2]` renvoie le vecteur `c("Hélène", "Raymond")`

=> `x[2,1]` renvoie `"Benjamin"`