

Formation-R-perfectionnement

Module Écrire des fonctions avec R

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 initialisation**.

Champ couvert par cette formation

Ce support couvre **Écrire des fonctions avec R**, pour en finir avec les copiés-collés.

Ce module se décompose en plusieurs chapitres :

- 01 - Écrire ses propres fonctions
- 02 - Itération de fonctions
- 03 - Annexes

Ce support est orienté pour être utile aux agents du SSM Agriculture et se concentre sur une utilisation de R via **RStudio** qui est mise à disposition des agents sur la plateforme interne Cerise basée sur RStudio Workbench.

Chapitre 01 : Écrire ses propres fonctions

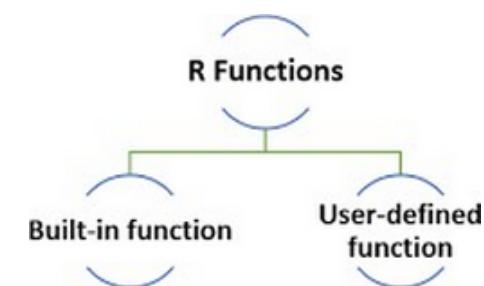
- Quand écrire une fonction ?
- Comment écrire une fonction ?
- Comment utiliser une fonction ?
- Valeurs par défaut
- Exécution conditionnelle
- Portée des variables
- Stocker ses fonctions
- Documenter ses fonctions
- Programmer avec le tidyverse
- Passer des colonnes en paramètres
- Nommer de nouvelles colonnes
- Utiliser des chaînes de caractères
- Récapitulatif

Quand écrire une fonction ?

•
•
•

Quand écrire une fonction ?

R dispose de nombreuses fonctions de base et le chargement de packages permet d'en importer d'autres.



R permet également à l'utilisateur d'écrire ses propres fonctions.

=> C'est l'équivalent des macro dans SAS



Quand écrire une fonction ?

Règle courante : dès qu'on a répété le même code plus de deux fois, ça vaut le coup de l'encapsuler dans une fonction

- **Avantages :**

- Évite les erreurs de copié-collé
- Facilite la lecture du code si la fonction a un nom adapté
- Facilite la mise à jour du code et la correction des erreurs (on corrige une seule fois dans la fonction et non à plein d'endroits du code)
- Permet de réutiliser le même code dans plusieurs scripts

Quand écrire une fonction ?

Plutôt que de recopier le même code pour chaque département, on peut écrire une fonction

On souhaite calculer la composition des ménages dans chaque département

```
base_RP %>%
  mutate(DEP = ifelse(substr(CODEGEO, 1, 2) == "97",
                     substr(CODEGEO, 1, 3),
                     substr(CODEGEO, 1, 2))) %>%
  filter(DEP == "31") %>%
  summarise(PMENPSEUL = 100*sum(PMENPSEUL)/sum(PMEN),
            PMENSFAM = 100*sum(PMENSFAM)/sum(PMEN),
            PMENCOUPSENF = 100*sum(PMENCOUPSENF)/sum(PMEN),
            PMENCOUPAENF = 100*sum(PMENCOUPAENF)/sum(PMEN),
            PMENFAMMONO = 100*sum(PMENFAMMONO)/sum(PMEN))

base_RP %>%
  mutate(DEP = ifelse(substr(CODEGEO, 1, 2) == "97",
                     substr(CODEGEO, 1, 3),
                     substr(CODEGEO, 1, 2))) %>%
  filter(DEP == "34") %>%
  summarise(PMENPSEUL = 100*sum(PMENPSEUL)/sum(PMEN),
            PMENSFAM = 100*sum(PMENSFAM)/sum(PMEN),
            PMENCOUPSENF = 100*sum(PMENCOUPSENF)/sum(PMEN),
            PMENCOUPAENF = 100*sum(PMENCOUPAENF)/sum(PMEN),
            PMENFAMMONO = 100*sum(PMENFAMMONO)/sum(PMEN))

# Autres départements #
```



sans fonction :
 - code long
 - complexe à comprendre
 - difficile à maintenir (modifications, corrections...)



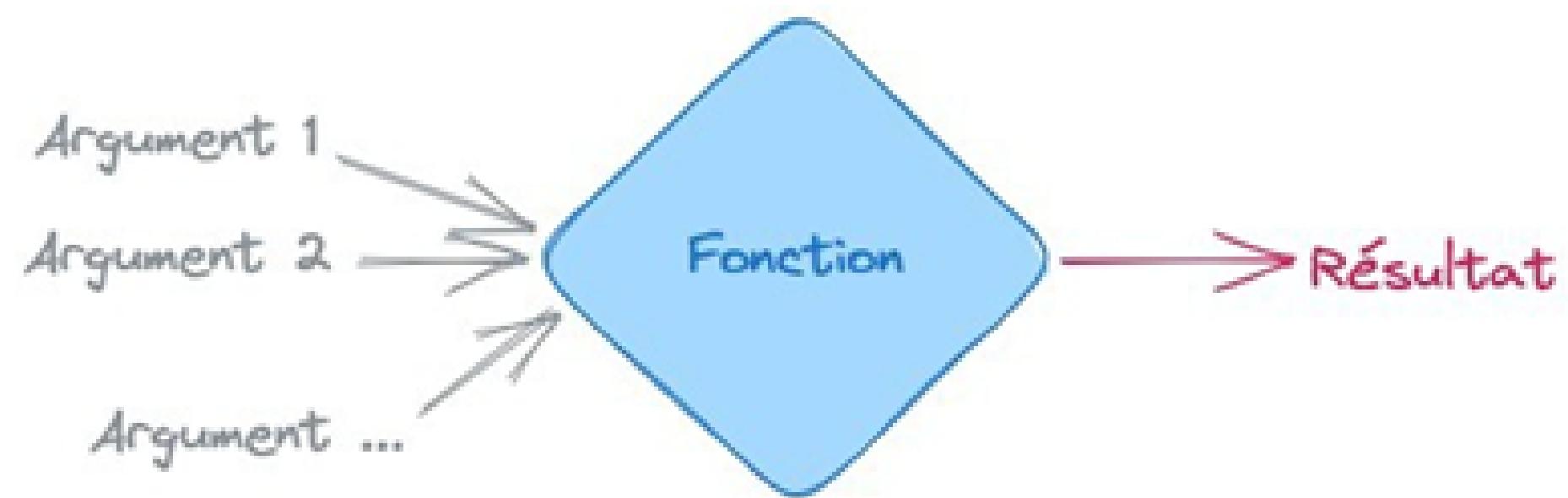
compoMenage_DEP("31")
 compoMenage_DEP("34")

avec une fonction :
 - code plus court
 - plus explicite
 - plus facile à maintenir

Comment écrire une fonction ?

Comment écrire une fonction ?

Une fonction exécute une série d'instructions, à partir d'éventuels objets donnés en entrée.



On crée une fonction en utilisant l'instruction ***function***.

Elle doit être suivie :

- d'une paire de parenthèses ⇒ pour indiquer les arguments
- d'une paire d'accolades ⇒ pour indiquer les instructions

```
ma_fonction <- function(...){ ... }
```

Comment écrire une fonction ?

Quatre éléments principaux pour créer une fonction :

- Le nom de la fonction
- Les arguments éventuels de la fonction
- Un bloc d'instructions pour constituer le corps de la fonction
- Le résultat éventuel de la fonction

```

nom de la fonction
ma_fonction <- function(argument1, argument2, arguments
                         ... ) {
  resultat <- argument1 + argument2 corps de la fonction
  return(resultat) résultat
}

```

Comment appeler une fonction ?

Les parenthèses servent à indiquer les paramètres de la fonction (ou arguments)

Ce sont ceux qui seront passés lors de l'appel de la fonction

- On peut prévoir autant d'arguments que l'on souhaite
- Les arguments doivent être nommés pour être ensuite utilisés dans la fonction

Exemple : `eff_milliers <- function(x, digits)`

Comment écrire une fonction ?

Les accolades comprennent une série d'instructions R

c'est le corps de la fonction, le code qui sera exécuté à l'appel de la fonction

- On utilise dans le corps de la fonction les arguments renseignés dans les parenthèses
- La fonction renvoie un résultat via l'instruction `return()`

Comment écrire une fonction ?

L'instruction `return()` arrête l'exécution de la fonction

⇒ tout ce qui est après le `return()` ne sera pas exécuté

```
eff_milliers <- function(x, digits){
  x_milliers <- round(x / 1000, digits = digits)
  return(x_milliers)
  print("Effectifs en milliers calculés !")
}

> eff_milliers(245631, digits = 0)
[1] 246
```

le message compte-rendu n'a pas été affiché car la commande `print` est située APRES l'instruction `return()`

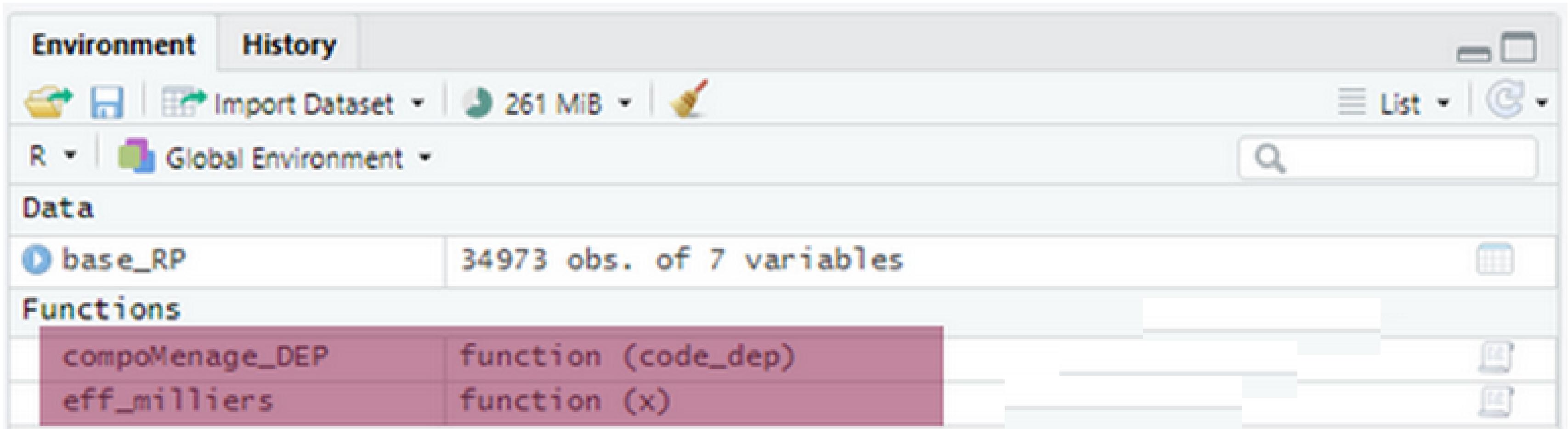
Comment écrire une fonction ?

return() est obligatoire

- une fonction **peut ne rien renvoyer** : certaines fonctions accomplissent une action mais ne renvoient rien (affichage de graphique, export de fichier par exemple)
- sans précision, la fonction **renvoie le dernier objet appelé** dans le bloc d'instructions
- pour un code plus sûr et plus clair, il est recommandé d'utiliser **systématiquement** la fonction `return()` afin de préciser l'élément à renvoyer, quitte à ce que ce soit l'élément `NULL`

Comment écrire une fonction ?

Une fonction est un objet de type *function* – elle apparaît dans l'environnement global dans la catégorie **Functions**



| Functions | |
|-----------------|---------------------|
| compoMenage_DEP | function (code_dep) |
| eff_milliers | function (x) |

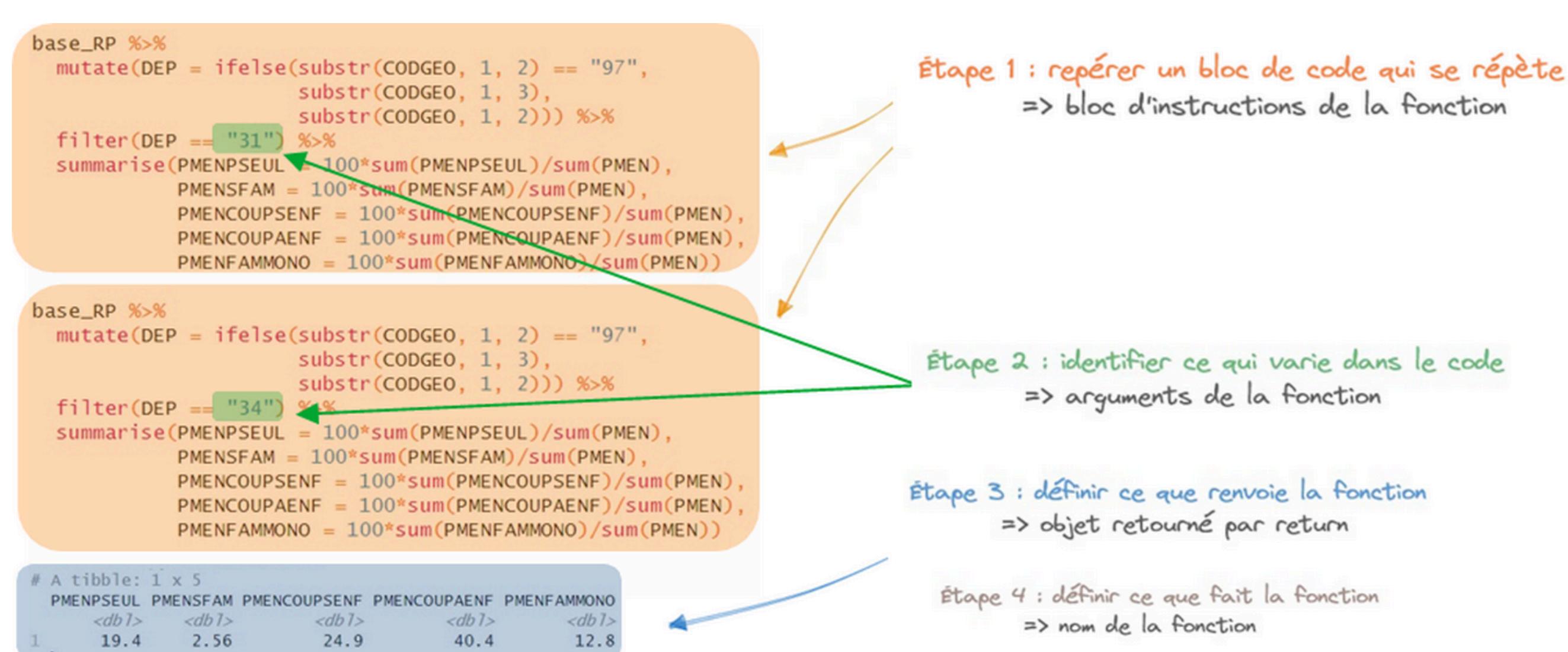
R traite les fonctions définies par l'utilisateur de la même façon que les autres.

Attention à ne pas nommer une fonction personnelle de la même façon qu'une fonction existante, pour ne pas l'écraser.

Comment écrire une fonction ?

Identifier et écrire une fonction dans la pratique :

Code initial :



The diagram illustrates the process of identifying and writing a function from initial R code. It consists of three main sections: two identical blocks of code at the top, a third block below them, and a summary table at the bottom.

Étape 1 : repérer un bloc de code qui se répète
 => bloc d'instructions de la fonction

Étape 2 : identifier ce qui varie dans le code
 => arguments de la fonction

Étape 3 : définir ce que renvoie la fonction
 => objet retourné par return

Étape 4 : définir ce que fait la fonction
 => nom de la fonction

```
base_RP %>%
  mutate(DEP = ifelse(substr(CODEO, 1, 2) == "97",
                     substr(CODEO, 1, 3),
                     substr(CODEO, 1, 2))) %>%
  filter(DEP == "31") %>%
  summarise(PMENPSEUL = 100*sum(PMENPSEUL)/sum(PMEN),
            PMENSFAM = 100*sum(PMENSFAM)/sum(PMEN),
            PMENCOUPSENF = 100*sum(PMENCOUPSENF)/sum(PMEN),
            PMENCOUPAENF = 100*sum(PMENCOUPAENF)/sum(PMEN),
            PMENFAMMONO = 100*sum(PMENFAMMONO)/sum(PMEN))
```

```
base_RP %>%
  mutate(DEP = ifelse(substr(CODEO, 1, 2) == "97",
                     substr(CODEO, 1, 3),
                     substr(CODEO, 1, 2))) %>%
  filter(DEP == "34") %>%
  summarise(PMENPSEUL = 100*sum(PMENPSEUL)/sum(PMEN),
            PMENSFAM = 100*sum(PMENSFAM)/sum(PMEN),
            PMENCOUPSENF = 100*sum(PMENCOUPSENF)/sum(PMEN),
            PMENCOUPAENF = 100*sum(PMENCOUPAENF)/sum(PMEN),
            PMENFAMMONO = 100*sum(PMENFAMMONO)/sum(PMEN))
```

| # A tibble: 1 x 5 | PMENPSEUL | PMENSFAM | PMENCOUPSENF | PMENCOUPAENF | PMENFAMMONO |
|-------------------|-----------|----------|--------------|--------------|-------------|
| 1 | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> |
| 1 | 19.4 | 2.56 | 24.9 | 40.4 | 12.8 |

Comment écrire une fonction ?

Identifier et écrire une fonction dans la pratique :

La fonction correspondante :

```
compomenage_DEP <- function(code_dep){  
  tab_DEP <- base_RP %>%  
    mutate(DEP = ifelse(substr(CODGEO, 1, 2) == "97",  
          substr(CODGEO, 1, 3),  
          substr(CODGEO, 1, 2))) %>%  
    filter(DEP == code_dep) %>%  
    summarise(PMENPSEUL = 100*sum(PMENPSEUL)/sum(PMEN),  
              PMENSFAM = 100*sum(PMENSFAM)/sum(PMEN),  
              PMENCOUPSENF = 100*sum(PMENCOUPSENF)/sum(PMEN),  
              PMENCOUPAENF = 100*sum(PMENCOUPAENF)/sum(PMEN),  
              PMENFAMMONO = 100*sum(PMENFAMMONO)/sum(PMEN))  
  return(tab_DEP)  
}
```

Utilisation / test dans la console:

```
compomenage_DEP("31")  
compomenage_DEP("34")
```

Comment utiliser une fonction ?

:

Comment utiliser une fonction ?

Une fonction personnelle s'utilise comme n'importe quelle autre fonction de R

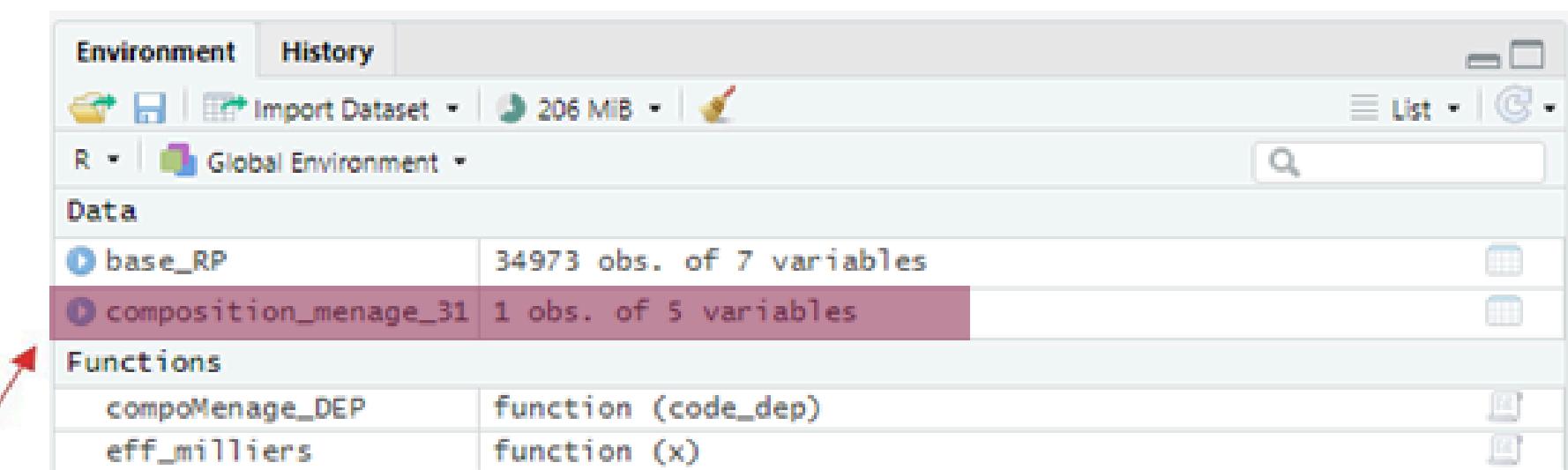
Notamment, on récupère la sortie de la fonction en **affectant le résultat à un objet**

⇒ sinon il n'y aura que l'affichage dans la console

Dans la console :

```
> compoMenage_DEP("31")
# A tibble: 1 x 5
  PMENPSEUL PMENSFAM PMENCOUPSENF PMENCOUPAENF PMENFAMMONO
    <dbl>     <dbl>      <dbl>       <dbl>      <dbl>
1     19.5      3.11     23.3       42.2      11.8
> composition_menage_31 <- compoMenage_DEP("31")
```

On affecte le résultat à un objet
pour avoir le résultat dans l'environnement global



| Data | base_RP | 34973 obs. of 7 variables |
|-----------|-----------------------|---------------------------|
| | composition_menage_31 | 1 obs. of 5 variables |
| Functions | compoMenage_DEP | function (code_dep) |
| | eff_milliers | function (x) |

Exercice 1-a) à d)

Valeurs par défaut

•
•
•

Valeurs par défaut

Au moment de la définition d'une fonction, on peut indiquer une valeur par défaut qui sera prise par l'argument si l'utilisateur n'en fournit pas.

```
eff_milliers <- function(x, digits = 3){
  x_milliers <- round(x / 1000, digits = digits)
  return(x_milliers)
}
```



Dans l'aide d'une fonction existante,
on voit les paramètres qui ont une valeur
par défaut dans la section *Usage*

sum {base}

Sum of Vector Elements

Description

sum returns the sum of all the values present in its arguments.

Usage

sum(..., na.rm = FALSE)

Valeurs par défaut

⇒ lorsqu'un paramètre a une valeur par défaut, il devient facultatif
on peut utiliser la fonction sans préciser de valeur pour ce paramètre.

```
> eff_milliers(252623)  
[1] 252.623
```

Par défaut, digits = 3

```
> eff_milliers(252623, digits = 3) On a précisé l'argument malgré la valeur par défaut  
[1] 252.623
```

```
> eff_milliers(252623, digits = 0) On a choisi une autre valeur que celle par défaut  
[1] 253
```

Valeurs par défaut

L'ordre et le nommage des paramètres ont une importance :

- Si on les nomme, on peut les appeler dans l'ordre que l'on veut

```
> eff_milliens(x = 252623, digits = 3)
[1] 252.623
> eff_milliens(digits = 3, x = 252623)
[1] 252.623
```

⋮

- Sinon, il faut les appeler dans l'ordre prévu lors de la définition de la fonction

```
> eff_milliens(252623, 3)
[1] 252.623
> eff_milliens(3, 252623)
[1] 0.003
```

Lors de la définition d'une fonction, il vaut donc mieux placer les arguments avec une valeur par défaut en dernier : de cette façon, il est plus facile de ne pas les nommer.

Exercice 1 – e)

Exécution conditionnelle



Exécution conditionnelle

Dans une fonction, on liste une série d'instructions que la fonction va effectuer

Pour des actions un peu plus complexes, on peut utiliser des conditions, en utilisant les mots-clés `if` et `else`

Par exemple, on va pouvoir exécuter certaines instructions selon la valeur prise par les **arguments de la fonction**.

```
que_faire_ajd ?  
    si aujourd'hui = jour de semaine  
        ➔ aller au travail  
    sinon, si aujourd'hui = samedi  
        ➔ activités  
    sinon  
        ➔ profiter !
```

Exécution conditionnelle

- ⇒ `if` permet de n'exécuter du code que lorsqu'une condition est remplie
- ⇒ `else` permet d'exécuter des instructions lorsque la condition donnée au `if` n'est pas remplie
- ⇒ on peut enchaîner plusieurs blocs d'instructions `if / else if / else`
- ⇒ `else` s'utilise après `if`, mais n'est pas obligatoire

Portée des variables

:

Portée des variables

Dans une fonction, l'environnement est temporaire

⇒ les éléments créés dans la fonction n'existent pas dans l'environnement global

```

a <- 3
b <- 5

fonction_plus <- function(){
  somme <- a + b
  a <- 12
  b <- a + 24
  return(b)
}

> fonction_plus()
[1] 36
> somme
Error: object 'somme' not found
> a
[1] 3
> b
[1] 5
  
```

Une fonction peut accéder à des objets extérieurs
(calcul de somme en faisant appel à a et b à l'extérieur de la fonction)

Si des objets extérieurs ont le même nom que des objets à l'intérieur de la fonction, ces derniers sont prioritaires
(a vaut 12 et non 5 pour calculer b <- a + 24)

Un objet créé dans une fonction n'existe que dans cette fonction
(somme n'existe pas en dehors de la fonction)

Un objet modifié dans une fonction ne modifie pas l'objet extérieur portant le même nom
(a vaut toujours 3 et b vaut toujours 5 à l'issue de la fonction)

Stocker ses fonctions

:

Stocker ses fonctions

Pour conserver ses fonctions, on peut les enregistrer dans un script à part.

⇒ on utilise ensuite la commande `source()` pour faire appel à ce script depuis un autre script :

```
source("repertoire/mon_script_de_fonctions.R")
```

Pour les experts, il est possible de créer un **package** pour encapsuler ses fonctions.

Exercices 2 et 3

Documenter ses fonctions

:

Documenter ses fonctions

Pour réutiliser ou partager ses fonctions, il est important de les documenter

→ le package `{roxygen}` aide à générer une documentation efficace dans le corps de la fonction :

cliquer sur -> Code -> Insert Roxygen Skeleton pour faire apparaître un bloc de documentation à remplir

On peut renseigner :

```
#' Title
#'
#' @param x
#' @param digits
#'
#' @return
#' @export
#'
#' @examples
eff_milliers <- function(x, digits = 3){
  x_milliers <- round(x / 1000, digits = digits)
  return(x_milliers)
}
```

title : une description rapide de ce que fait la fonction

params : les noms des paramètres et le format attendu

return : le résultat de la fonction

examples : quelques exemples d'utilisation

```
## Conversion des effectifs en milliers
##
## @param x numérique à convertir en milliers
## @param digits nombre de chiffres après la virgule
## à retenir pour l'arrondi
##
## @return chiffre converti en millier
##
## @examples
## eff_milliers(1000)
## eff_milliers(236980, digits = -1)
eff_milliers <- function(x, digits = 3){
  x_milliers <- round(x / 1000, digits = digits)
  return(x_milliers)
}
```

Programmer avec le Tidyverse

Passer des noms de colonnes en paramètres

Les fonctions du {tidyverse} sont compliquées à utiliser dans nos propres fonctions

⇒ lorsque les variables des tableaux de données utilisées dans le corps de la fonction ne sont pas saisies directement mais proviennent d'un paramètre, cela génère des erreurs

création d'une fonction pour calculer des moyennes par groupes

```
moyenne_par_groupe <- function(df, var_groupe, var_moyenne) {
  moyennes <- df %>%
    group_by(var_groupe) %>%
    summarise(moyenne = mean(var_moyenne, na.rm = TRUE))

  return(moyennes)
}
```

```
iris %>% moyenne_par_groupe(var_groupe = Species, var_moyenne = Sepal.Length)
```

Error in `group_by()`:
! Must group by variables found in `.data`.
x column `var_groupe` is not found.
Run `rlang::last_trace()` to see where the error occurred.
> |



l'appel de la fonction génère un message d'erreur :
la fonction cherche une variable intitulée "var_groupe" dans la table iris
et n'en trouve aucune

Passer des noms de colonnes en paramètres

l'opérateur {{ }} (*curly curly*) permet de forcer l'évaluation du paramètre

⇒ lorsque le paramètre de la fonction est une variable à laquelle on souhaite accéder, on entoure les utilisations du paramètre par des accolades

utilisation de l'opérateur {{ }} pour forcer l'évaluation

```
moyenne_par_groupe <- function(df, var_groupe, var_moyenne){

  moyennes <- df %>%
    group_by({{ var_groupe }}) %>%
    summarise(moyenne = mean({{ var_moyenne }}), na.rm = TRUE))

  return(moyennes)
}
```

```
iris %>% moyenne_par_groupe(var_groupe = Species, var_moyenne = Sepal.Length)
```

| # A tibble: 3 x 2 | Species | moyenne |
|-------------------|------------|---------|
| 1 | setosa | 5.01 |
| 2 | versicolor | 5.94 |
| 3 | virginica | 6.59 |

✓ l'appel de la fonction renvoie bien le résultat souhaité :
une table contenant la valeur moyenne de la variable "Sepal.Length"
pour chaque valeur de la variable "Species"

Nommer de nouvelles colonnes

Pour créer de nouvelles colonnes à partir d'un argument,

on utilise les deux opérateurs {{ }} et := (*walrus operator*)

⇒ on peut placer le texte que l'on souhaite comme nouveau nom de colonne

(ex : "{{var_moyenne}}_moy")

utilisation de l'opérateur := pour créer une nouvelle variable selon un paramètre de la fonction

```
moyenne_par_groupe <- function(df, var_groupe, var_moyenne){

  moyennes <- df %>%
    group_by({{ var_groupe }}) %>%
    summarise("{{var_moyenne}}_moy" := mean('{{var_moyenne}}', na.rm = TRUE))

  return(moyennes)
}
```

```
iris %>%
  moyenne_par_groupe(var_groupe = Species,
                      var_moyenne = Sepal.Length)

# A tibble: 3 x 2
#>   Species      Sepal.Length_moy
#>   <fct>          <dbl>
#> 1 setosa           5.01
#> 2 versicolor       5.94
#> 3 virginica        6.59
```

✓ le nom de la variable créée dépend de la variable passée en paramètre de la fonction

```
iris %>%
  moyenne_par_groupe(var_groupe = Species,
                      var_moyenne = Petal.Width)

# A tibble: 3 x 2
#>   Species      Petal.Width_moy
#>   <fct>          <dbl>
#> 1 setosa           0.246
#> 2 versicolor       1.33
#> 3 virginica        2.03
```

Utiliser des chaînes de caractères

Lorsque l'argument est passé sous forme de chaînes de caractères, on ne peut pas utiliser l'opérateur {{ }}

⇒ le prénom **.data** permet d'accéder aux colonnes du tableau à partir de leur nom sous forme de chaîne de caractères

utilisation de l'opérateur **.data** permet d'accéder aux colonnes lorsque les paramètres sont des chaînes de caractères

```
moyenne_par_groupe <- function(df, var_groupe, var_moyenne){

  moyennes <- df %>%
    group_by(.data[[var_groupe]]) %>%
    summarise(moyenne = mean(.data[[var_moyenne]]), na.rm = TRUE))

  return(moyennes)
}
```

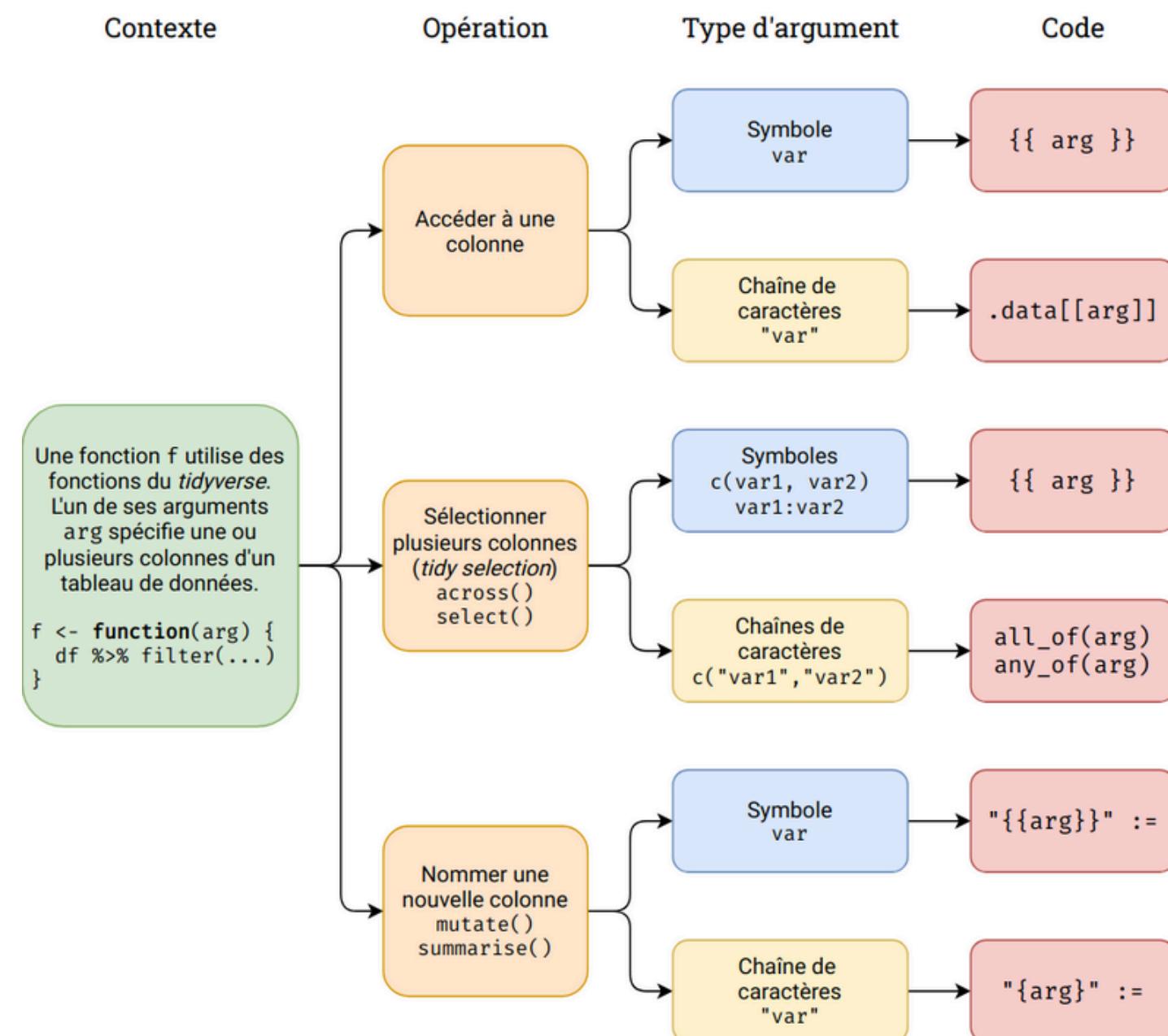
```
iris %>% moyenne_par_groupe(var_groupe = "Species", var_moyenne = "Sepal.Length")
```



| Species | moyenne |
|--------------|---------|
| <fct> | <dbl> |
| 1 setosa | 5.01 |
| 2 versicolor | 5.94 |
| 3 virginica | 6.59 |

✓ l'appel de la fonction donne bien le résultat attendu

Récapitulatif (source [ici](#))



Chapitre 02 : Itération de fonctions

- Itération de fonctions : présentation
- Utilisation de across
- Utilisation de across : Exemple avec mutate
- Utilisation de across : Exemple avec summarise
- Package purrr
- {purrr} : Itérer sur un vecteur ou une liste
- {purrr} : Itérer sur deux éléments
- {purrr} : Généralisation
- {purrr} : Manipuler des listes
- {purrr} : Fonctions à effets de bord

⋮
⋮
⋮
⋮
⋮

Itération de fonctions

Présentation

Intérêt d'une fonction = pouvoir être utilisée plusieurs fois

Utilisation de la fonction `across()`

- Permet d'appliquer, dans un environnement dplyr, un même traitement à plusieurs variables d'une même table

Utilisation du package `{purrr}`

- Regroupe des outils pour appliquer la même fonction à plusieurs éléments d'une liste ou d'un vecteur



Utilisation de `across()`

dplyr::`across()`

EXAMPLE:

```
df %>%
  group_by(species) %>%
  summarise(
    across(where(is.numeric), mean)
  )
```

use within `mutate()`
or `summarize()` to
apply function(s) to
a selection of columns!



| species | mass_g | age_yr | range_sqmi |
|---------|--------|--------|------------|
| pika | 163 | 2.4 | 0.46 |
| marmot | 1509 | 3.0 | 0.87 |
| marmot | 2417 | 5.6 | 0.102 |

Utilisation de `across()`

A l'aide des fonctions
on sait passer de la
syntaxe 1 à la syntaxe
2

Avec `across`, on va
pouvoir réduire la
longueur du code et
passer à la syntaxe 3

```
# Conversion en milliers des variables de population :
base_RP <- base_RP %>%
  mutate(PMEN = round(PMEN / 1000, digits = 3),
        PMENPSEUL = round(PMENPSEUL / 1000, digits = 3),
        PMENSFAM = round(PMENSFAM / 1000, digits = 3),
        PMENCOUPSENF = round(PMENCOUPSENF / 1000, digits = 3),
        PMENCOUPAENF = round(PMENCOUPAENF / 1000, digits = 3),
        PMENFAMMONO = round(PMENFAMMONO / 1000, digits = 3))
```

```
# Avec une fonction
base_RP <- base_RP %>%
  mutate(PMEN = eff_milliers(PMEN),
        PMENPSEUL = eff_milliers(PMENPSEUL),
        PMENSFAM = eff_milliers(PMENSFAM),
        PMENCOUPSENF = eff_milliers(PMENCOUPSENF),
        PMENCOUPAENF = eff_milliers(PMENCOUPAENF),
        PMENFAMMONO = eff_milliers(PMENFAMMONO))
```

```
# Avec across
base_RP <- base_RP %>% mutate(across(starts_with("PMEN"), eff_milliers))
```

Utilisation de `across()`

Objectif de `across()` ⇒ appliquer une même fonction à un ensemble de variables d'une table de données

`across()` s'utilise à l'intérieur des fonctions `dplyr` comme `mutate()` et `summarise()`.

La syntaxe générale est la suivante :

```
across( variables à traiter ,  
       traitement ,  
       règle de renommage )
```



usage avancé, on peut s'en passer dans un premier temps
(Voir annexe)

Utilisation de `across()`

Pour désigner les variables à traiter, on utilise les mêmes « select helpers » que ceux utilisés dans la fonction `select` par exemple :

| Fonction | Sélection | Exemple |
|-------------------------------------|---|---|
| <code>c()</code> | Les colonnes citées dans le vecteur | <code>c(POP_MUN, DENSITE)</code> |
| <code>starts_with()</code> | Les colonnes dont le nom commence par un certain pattern | <code>starts_with("POP")</code> |
| <code>ends_with()</code> | Les colonnes dont le nom termine par un certain pattern | <code>ends_with("20")</code> |
| <code>contains()</code> | Les colonnes dont le nom contient un certain pattern | <code>contains("REG")</code> |
| : | Une étendue de colonnes dans la table | <code>REG:POP_MUN</code> |
| <code>where()</code> | Les colonnes vérifiant une condition | <code>where(is.numeric)</code> |
| <code>any_of()/ all_of()</code> | Les colonnes dont les noms appartiennent éventuellement/ absolument à un vecteur de chaînes de caractère | <code>any_of(c("POP_MUN", "DENSITE", "REG"))</code> |

Utilisation de `across()`

Le traitement à effectuer correspond à une fonction ⇒ on peut la renseigner de différentes façons :

- avec le nom de la fonction (sans parenthèse) = lorsque la fonction n'a qu'un seul paramètre, qui est la variable à traiter

`across(..., mean)`

la fonction `mean()` est appliquée
à l'ensemble des variables sélectionnées

- avec ~ et .x :

.x représente la place
de la variable à traiter

`across(... , ~ round(.x, digits = 1))`

chaque variable est arrondie
à un chiffre après la virgule



*Liberté
Égalité
Fraternité*

Utilisation de `across()`

- avec une fonction anonyme:

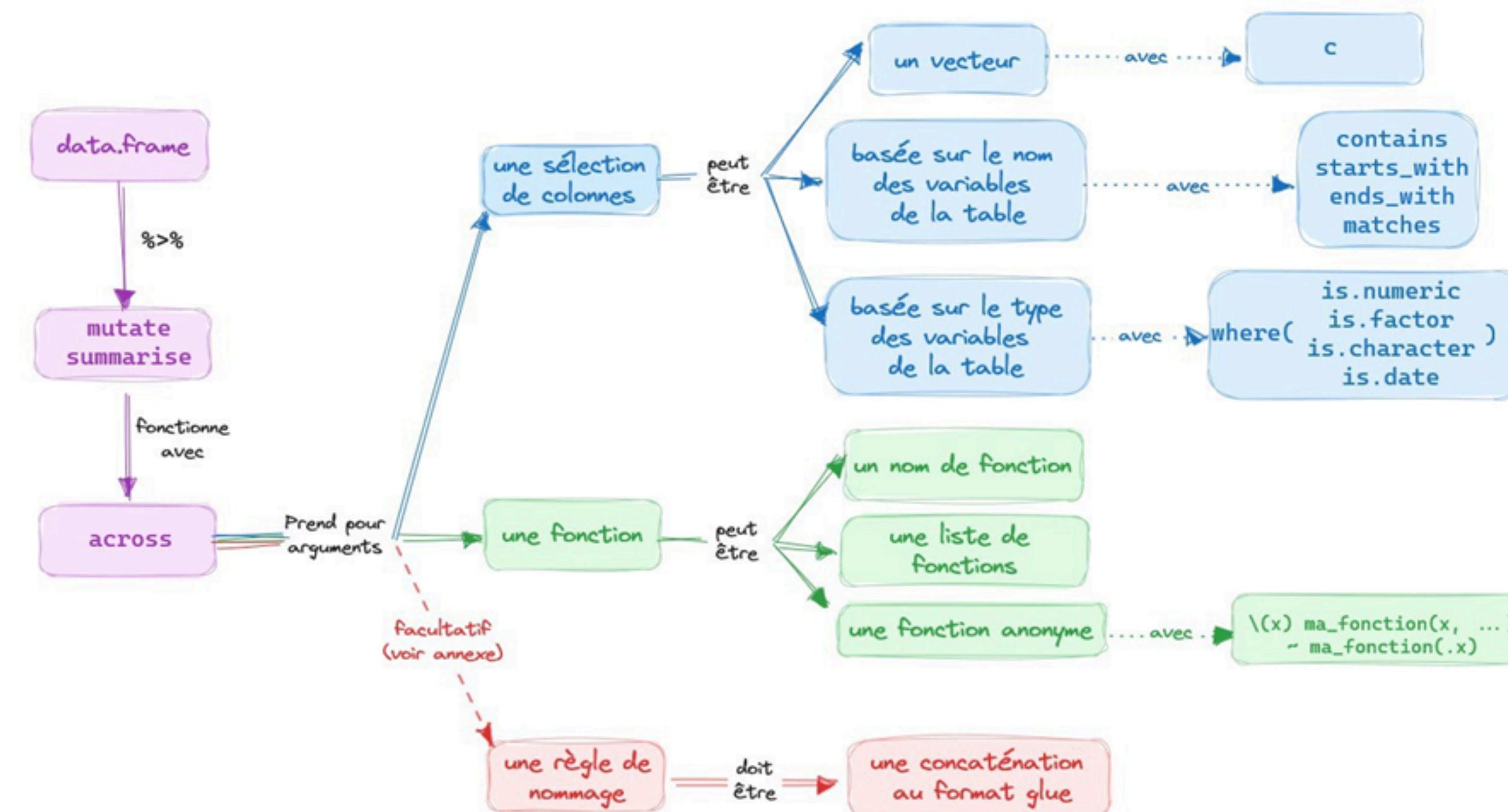
- définie avec (x)

- définie avec des paramètres

`across(... , function(nb) { nb/100 })` chaque variable est divisée par 100



Utilisation de `across()`



Utilisation de `across()` : Exemples avec `mutate`

```
# Mutate avec across
base_RP <- base_RP %>% mutate(
  # Avec une fonction sans autre paramètre que la variable à traiter
  across(starts_with("PMEN"), eff_milliers),
  # Avec une fonction qui a d'autres paramètres
  across(PMEN:PMENFAMMONO, \((x) eff_milliers(x, digits = 1)),
  # Avec une fonction à la carte
  across(where(is.numeric), function(x){round(x / 1000, digits = 1)}))
```

Les variables d'origine
sont écrasées dans la
table

| CODGEO | PMEN | PMENPSEUL | PMENSFAM | PMENCOUPSENF | PMENCOUPAENF | PMENFAMMONO |
|--------|--------|-----------|----------|--------------|--------------|-------------|
| 01001 | 0.821 | 0.068 | 0.018 | 0.221 | 0.463 | 0.051 |
| 01002 | 0.263 | 0.034 | 0.000 | 0.063 | 0.152 | 0.014 |
| 01004 | 13.795 | 2.987 | 0.172 | 3.280 | 5.622 | 1.733 |

Exercice 5 – a) et b)

Utilisation de `across()` : Exemples avec `summarise`

```
# Calculer le même agrégat pour plusieurs variables
base_RP %>% summarise(across(where(is.numeric), sum))

# A tibble: 1 x 6
  PMEN PMENPSEUL PMENSFAM PMENCOUPSENF PMENCOUPAENF PMENFAMMONO
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 69155448. 12218557. 1382257. 16737125. 30424385. 8393124.
```

```
# Calculer plusieurs agrégats sur une seule variable
base_RP %>% summarise(
  across(PMENFAMMONO,
    list(tot = sum, min = min, q1 = \((x) quantile(x, probs = 0.25),
         med = median, q3 = \((x) quantile(x, probs = 0.75), max = max)))

# A tibble: 1 x 6
  PMENFAMMONO_tot PMENFAMMONO_min PMENFAMMONO_q1 PMENFAMMONO_med PMENFAMMONO_q3
  <dbl> <dbl> <dbl> <dbl> <dbl>
1 8393124. 0 10.2 35.2 101.
# i 1 more variable: PMENFAMMONO_max <dbl>
```

Utilisation de `across()` : Exemples avec `summarise`

```
# Calculer plusieurs agrégats sur plusieurs variables
agregats <- base_RP %>% summarise(
  across(PMEN:PMENFAMMONO,
    list(tot = sum, min = min, q1 = \((x) quantile(x, probs = 0.25),
         med = median, q3 = \((x) quantile(x, probs = 0.75), max = max)))
```

| | PMEN_tot | PMEN_min | PMEN_q1 | PMEN_med | PMEN_q3 | PMEN_max | PMENPSEUL_tot | PMENPSEUL_min | PMENPSEUL_q1 | PM |
|---|----------|----------|----------|----------|---------|----------|---------------|---------------|--------------|-----|
| 1 | 69155448 | 0 | 195.8763 | 453.6295 | 1145 | 2102799 | 12218557 | 0 | 25 | 53. |

```
# Transposition de la table pour un résultat plus lisible
agregats %>%
  pivot_longer(cols = everything(),
               names_to = c("type_menage", ".value"),
               names_pattern = "(.*)_(.*)")
```

```
# A tibble: 6 x 7
  type_menage      tot     min     q1     med     q3     max
  <chr>          <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
1 PMEN        69155448.     0  196.   454.  1145  2102799.
2 PMENPSEUL  12218557.     0   25   53.9  140  590386.
3 PMENSFAM   1382257.      0     0     0  15.0  89463.
4 PMENCOUPSENF 16737125.     0  60.0  132.  329.  446349.
5 PMENCOUPAENF 30424385.     0  87.8  217.  561.  730340.
6 PMENFAMMONO 8393124.      0  10.2  35.2  101.  246262.
```

Exercice 5 – c) et d)

Package {purrr}



*Liberté
Égalité
Fraternité*

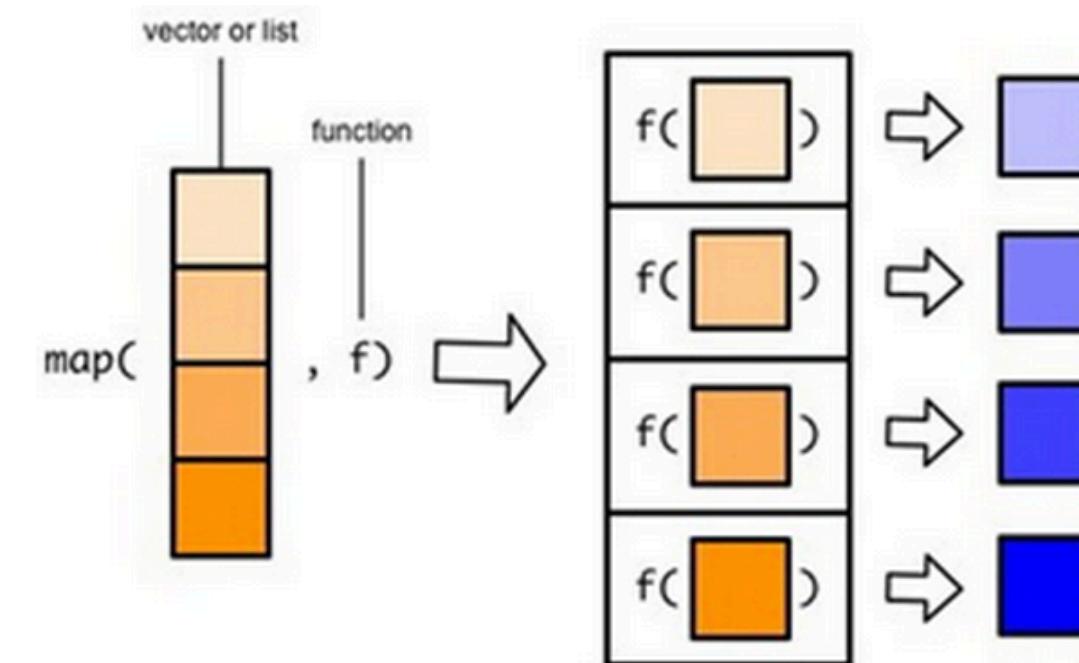
Package {purrr}

PACKAGE {PURRR}



{purrr} : Itérer sur un vecteur OU une liste

Objectif de `map()` ⇒ appliquer une même fonction à l'ensemble des éléments d'un vecteur ou d'une liste



map() s'utilise sur un vecteur ou une liste d'objets. La syntaxe générale est la suivante :

```
map( vecteur/liste à traiter ,  
      traitement )
```

{[]purrr} : Itérer sur un vecteur OU une liste

Le traitement correspond à la fonction à appliquer ⇒ on peut la renseigner de différentes manières :

- 1) Avec le nom de la fonction (sans parenthèse) = lorsque la fonction n'a qu'un seul paramètre, qui est l'élément à traiter

`map(... , max)` la fonction est appliquée
`map(... , head)` à l'ensemble des éléments

- 2) Avec une fonction anonyme

- définie avec un ~ : les éléments successifs de la liste sont alors identifiés avec le mot clé .x

.x représente la place
de la variable à traiter `map(... , ~ .x %>% summarise(moyenne = mean(POPTOT)))` la fonction summarise() est
appliquée à chaque table de la liste

- définie avec des paramètres

`map(... , function(nom_fichier) read_csv2(nom_fichier))`

chaque fichier est importé
avec la fonction read_csv2()

{purrr} : Itérer sur un vecteur OU une liste

Exemple :

création d'une fonction calculant plusieurs résultats pour un département

```
calcul_evol_departement <- function(num_departement){
  # restriction de la base au département d'intérêt
  base_RP_dep <- base_RP %>%
    filter(DEP == num_departement) %>%
    select(-DEP)

  # calcul des évolutions de population
  evolution <- base_RP_dep %>%
    group_by(annee) %>%
    summarise(population_totale = sum(PMEN),
              population_famille_mono = sum(PMENFAMMONO),
              prop_famille_mono = population_famille_mono / population_totale * 100) %>%
    mutate(across(c(population_totale, population_famille_mono), \(\text{x}\) eff_milliers(\text{x})),
           across(c(population_totale, population_famille_mono), \(\text{x}\) (\text{x} - lag(\text{x}))/lag(\text{x})*100, .names = ".col_evol"))

  return(evolution)
}
```

prend le n° du département comme paramètre

calcule pour chaque année :
la population totale, la proportion de familles monoparentales
et les évolutions d'une année sur l'autre

on peut utiliser la fonction sur plusieurs départements

```
calcul_evol_departement("31")
calcul_evol_departement("34")
```

{purrr} : Itérer sur un vecteur OU une liste

on peut appliquer la fonction à l'ensemble des départements d'Occitanie avec la fonction `map()`

`library(purrr)`

le vecteur contient l'ensemble des départements de la région

```
departements_occitanie <- c("09", "11", "12", "30", "31", "32", "34",
                               "46", "48", "65", "66", "81", "82")
```

`map()` applique la fonction `calcul_evol_departement()` à chaque département du vecteur

```
evolutions_occitanie <- departements_occitanie %>%
  map(calcul_evol_departement)
```

le résultat est une liste : on accède à ses différents éléments avec l'opérateur `[[]]`

```
evolutions_occitanie[[3]]
> evolutions_occitanie[[3]]
# A tibble: 3 x 6
  annee population_totale population_famille_mono prop_famille_mono population_totale_evol population_famille_mono_evol
  <chr>        <dbl>            <dbl>           <dbl>          <dbl>            <dbl>
1 2009         268.            20.2            7.53          NA              NA
2 2014         270.            21.2            7.85          0.617          4.98
3 2020         270.            23.2            8.57          0.0930         9.28
```

{purrr} : Itérer sur un vecteur OU une liste

Les écritures suivantes sont équivalentes :

- nom de la fonction sans parenthèse

```
evolutions_occitanie <- departements_occitanie %>%
  map(calcul_evol_departement)
```

la fonction `calcul_evol_departement()` est appliquée à chaque élément du vecteur

- définition d'une fonction avec ~

```
evolutions_occitanie <- departements_occitanie %>%
  map(~ calcul_evol_departement(.x))
```

- définition d'une fonction anonyme

- avec le raccourci \()

```
evolutions_occitanie <- departements_occitanie %>%
  map(\(x) calcul_evol_departement(x))
```

`.x` ou `x` prendront successivement pour valeur les différents éléments du vecteur

- avec le mot clé function()

```
evolutions_occitanie <- departements_occitanie %>%
  map(function(x) calcul_evol_departement(x))
```

Exercice 6 - a)

{purrr} : Itérer sur un vecteur OU une liste

Il n'est pas nécessaire de créer une fonction en temps que telle : on peut le faire directement à l'intérieur de map()

Exemple :

```
evolutions_occitanie <- departements_occitanie %>%
  map(function(code_dep) {
    base_RP %>%
      filter(DEP == code_dep) %>%
      select(-DEP) %>%
      group_by(annee) %>%
      summarise(population_totale = sum(PMEN),
               population_famille_mono = sum(PMENFAMMONO),
               prop_famille_mono = population_famille_mono / population_totale * 100) %>%
      mutate(across(c(population_totale, population_famille_mono), \(\text{x}\) eff_milliers(\text{x})),
             across(c(population_totale, population_famille_mono), \(\text{x}\) (\text{x} - lag(\text{x}))/lag(\text{x})*100, .names = ".\{.col\}_evol"))
  })

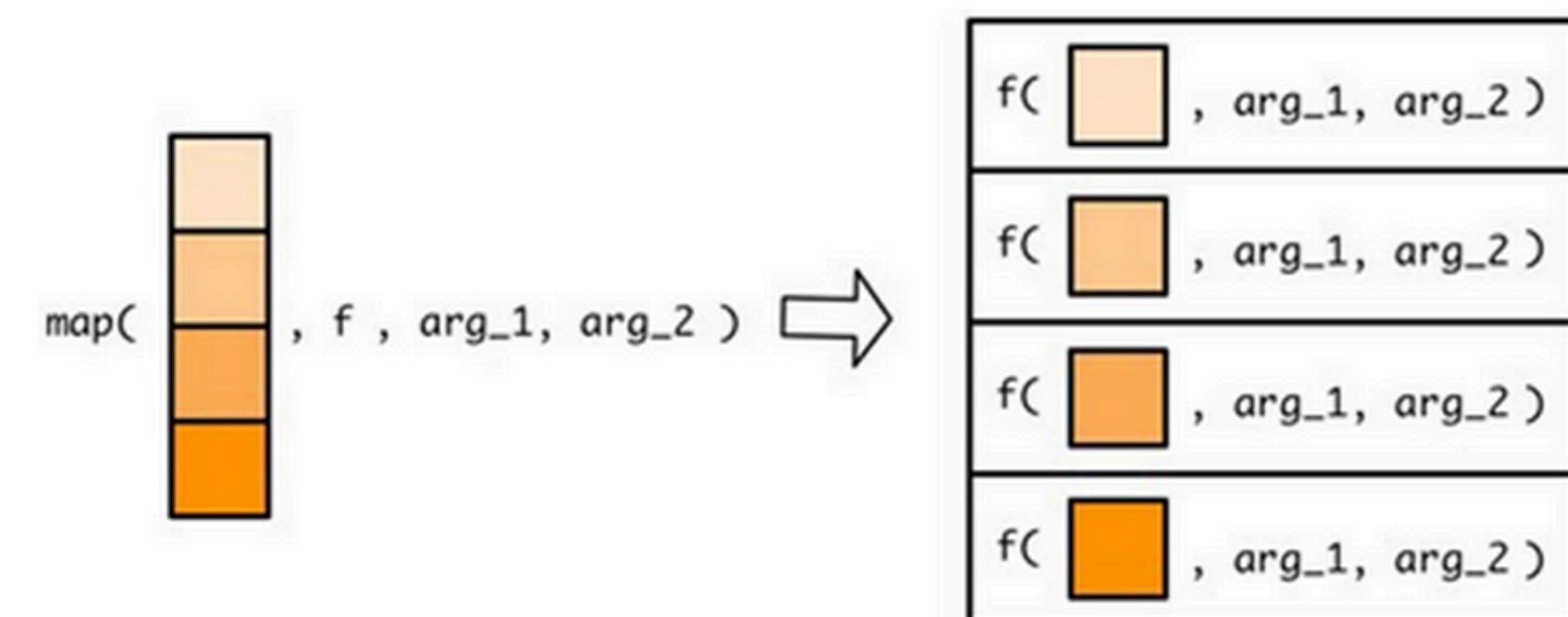
```

code_dep vaudra successivement les différentes valeurs du vecteur "departements_occitanie"

la fonction est décrite dans map()
elle ne sera pas stockée dans l'environnement global

{purrr} : Itérer sur un vecteur OU une liste

Possibilité d'ajouter des arguments supplémentaires à la fonction à appliquer



Exemple :

```
c("donnees_RP2019.csv", "donnees_RP2014.csv", "donnees_RP2009.csv") %>%
  map(read_delim,
      delim = ";",
      locale = locale(decimal_mark = "."),
      col_types = cols(CODGEO = col_character(), .default = col_double()))
```

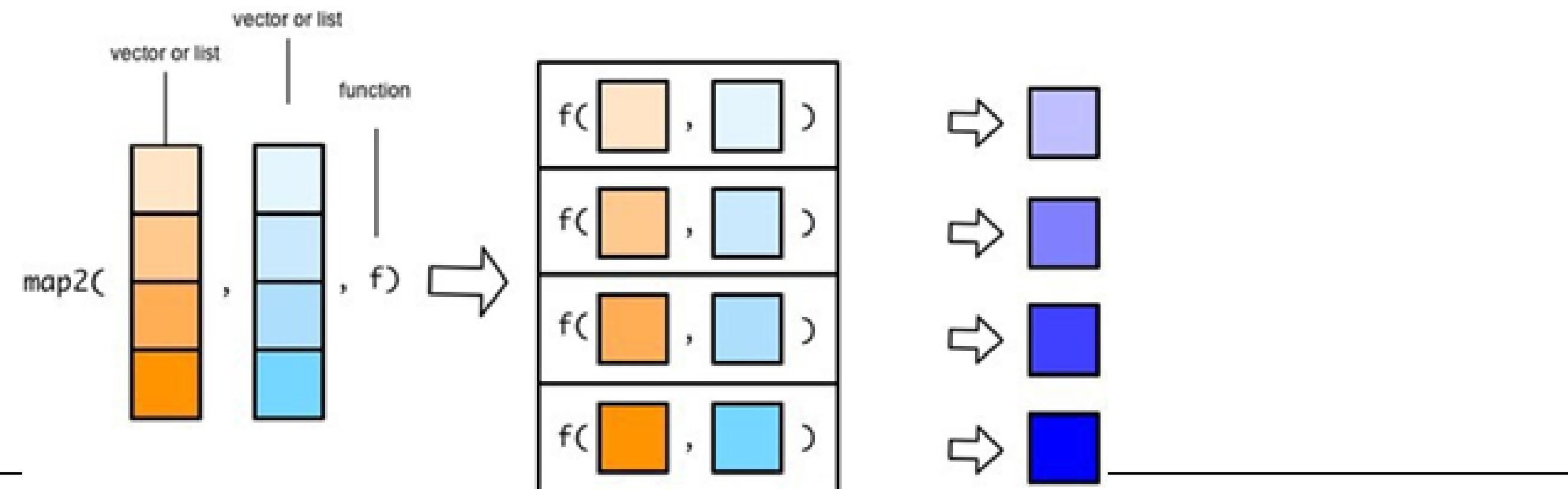
on importe les trois fichiers en une seule étape
avec la fonction `read_delim()`

on précise les différents paramètres à utiliser pour
l'import des trois fichiers

{purrr} : Itérer sur deux éléments

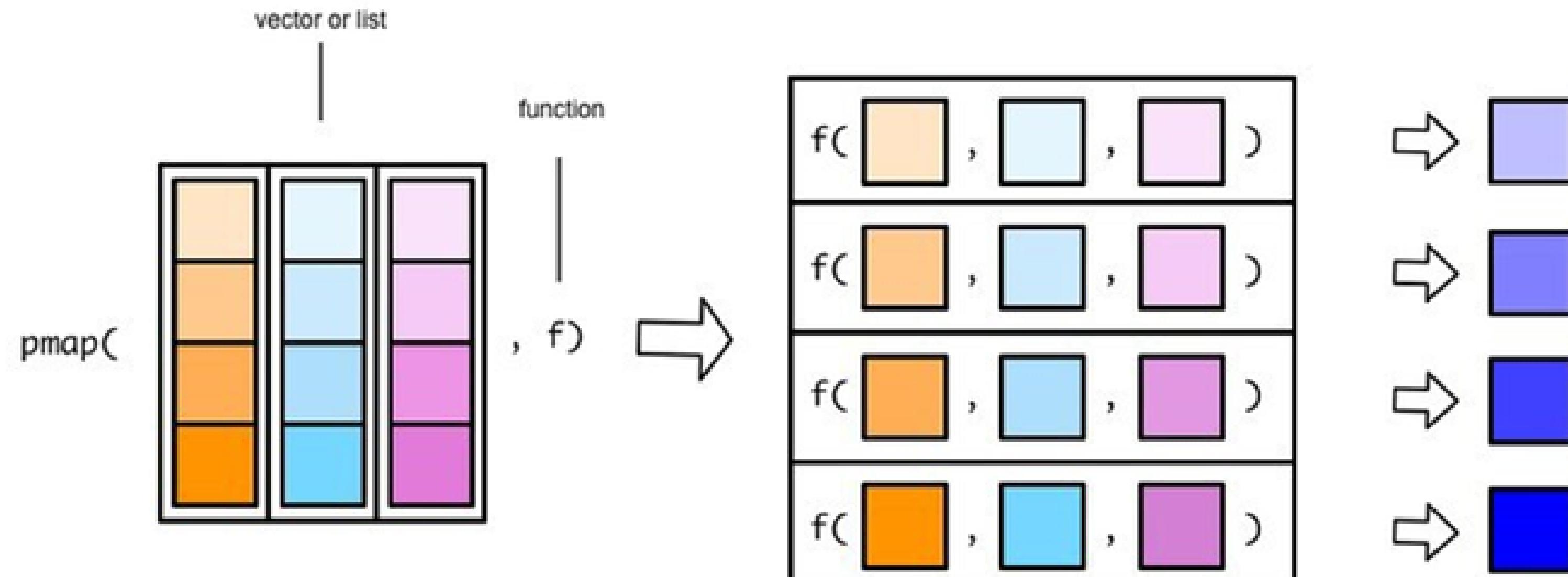
Objectif de `map2()` = appliquer une même fonction sur chacun des couples d'éléments de deux listes ou vecteurs
la fonction à appliquer peut être renseignée avec une fonction anonyme

- définie avec un ~ : les éléments successifs de la liste sont alors identifiés avec les mots clé .x et .y
- définie avec des paramètres



{purrr} : Généralisation

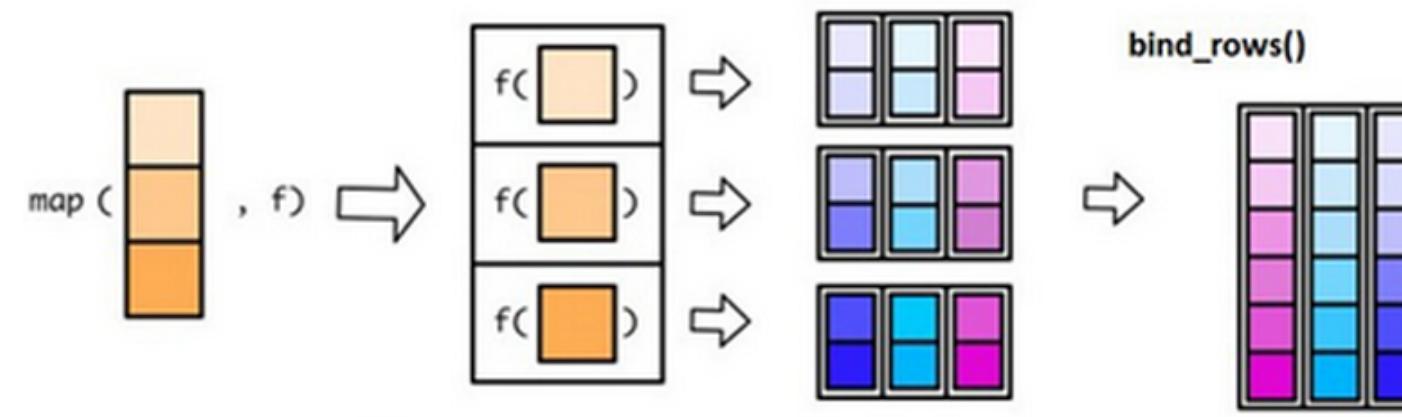
Généralisation avec la fonction **pmap()** = permet d'appliquer une même fonction sur chacun des p-uplets d'éléments des différentes listes ou vecteurs



{purrr} : Manipuler des listes

Les fonctions **map**, **map2** et **pmap** renvoient des listes

- on peut retrouver des `data.frame` plus faciles à manipuler en utilisant `bind_rows()`, ou la fonction `reduce()`
- ⇒ `bind_rows()` permet de concaténer une liste de tables



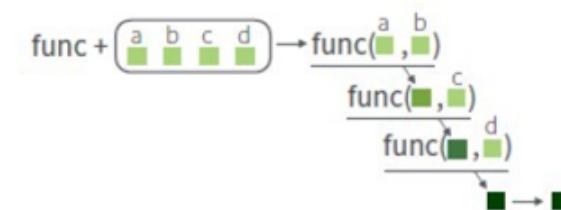
Exemple :

```
departements_occitanie %>%
  map(calcul_evol_departement) %>%
  bind_rows()

# A tibble: 39 x 7
  DEP    annee population_totale population_famille_mono prop_f
  <chr> <chr>          <dbl>            <dbl>
1 09     2009           147.             14.8
2 09     2014           148.             16.6
3 09     2020           150.             18.7
4 11     2009           346.             36.7
5 11     2014           357.             39.9
6 11     2020           366.             46.6
7 12     2009           268.             20.2
8 12     2014           270.             21.2
```

{purrr} : Manipuler des listes

⇒ `reduce()` applique une fonction de façon récursive à chaque élément d'une liste ou d'un vecteur



Exemple :

```
donnees_RP <- c("donnees_RP2019.csv", "donnees_RP2014.csv", "donnees_RP2009.csv") %>%
  map(read_delim,
      delim = ";",
      locale = locale(decimal_mark = "."),
      col_types = cols(CODEGEO = col_character(), .default = col_double())) %>%
  reduce(full_join)
```



après import, les trois fichiers sont joints les uns après les autres pour obtenir une seule table contenant toutes les informations

donnees_RP

| | CODEGEO | PMEN19 | PMEN14 | PMEN09 |
|---|---------|--------|--------|--------|
| 1 | <chr> | <dbl> | <dbl> | <dbl> |
| 2 | 01001 | 821 | 765 | 787 |
| 3 | 01002 | 263 | 256 | 209 |
| 4 | 01004 | 13795 | 13547 | 12781 |
| | 01005 | 1780 | 1602 | 1580 |



*Liberté
Égalité
Fraternité*

{purrr} : Fonctions à effets de bord

Pour les fonctions qui ne renvoient aucun résultat, on utilise la fonction **walk()**

⇒ même fonctionnement que pour les fonctions map, mais spécifiques pour les fonctions à effets de bord

Exemples : impression dans la console, affichage de graphiques, export de données...

- Extension à `walk2()` et `pwalk()`

{purrr} : Fonctions à effets de bord

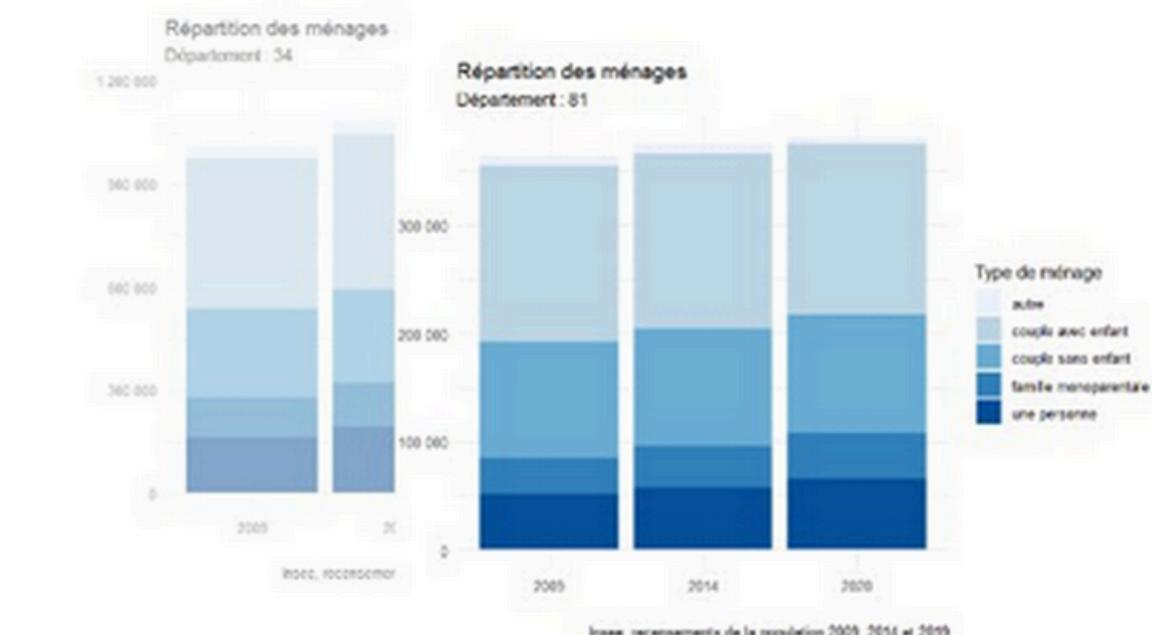
Exemple :

création de graphiques pour chaque département avec `map()`

```
graph_departements <- departements_occitanie %>%
  map(function(code_dep) {
    base_RP %>%
      filter(DEP == code_dep) %>%
      ggplot(aes(x = annee, y = nb_personnes, fill = type_menage)) +
      geom_col() +
      scale_fill_brewer() +
      scale_y_continuous(labels = function(x) format(x, scientific = FALSE, big.mark = " ")) +
      theme_minimal() +
      labs(title = "Répartition des ménages",
           subtitle = glue::glue("Département : {code_dep}"),
           caption = "Insee, recensements de la population 2009, 2014 et 2019",
           x = "", y = "", fill = "Type de ménage")
  })
  
```

affichage des graphiques
avec `walk()`

`graph_departements %>% walk(print)`



{purrr} : Fonctions à effets de bord

Exemple :

```
departements_occitanie <- c("09", "11", "12", "30", "31", "32", "34",
                           "46", "48", "65", "66", "81", "82")
libelles_departements_occitanie <- c("Ariège", "Aude", "Aveyron", "Gard", "Haute-Garonne",
                                         "Gers", "Hérault", "Lot", "Lozère", "Hautes-Pyrénées",
                                         "Pyrénées-Orientales", "Tarn", "Tarn-et-Garonne")

calcul des évolutions
pour chaque département
avec map()
xport dans plusieurs
fichiers csv de noms différents
avec walk2()

departements_occitanie %>%
  map(calcul_evol_departement) %>%
  walk2(libelles_departements_occitanie,
        ~ write_csv2(x,
                     file = glue:::glue("resultats_{.y}.csv")))

resultats_Ariège.csv
resultats_Aude.csv
resultats_Aveyron.csv
resultats_Gard.csv
resultats_Gers.csv
resultats_Haute-Garonne.csv
resultats_Hautes-Pyrénées.csv
resultats_Hérault.csv
resultats_Lot.csv
resultats_Lozère.csv
resultats_Pyrénées-Orientales.csv
resultats_Tarn.csv
resultats_Tarn-et-Garonne.csv
```

Exercice 7



*Liberté
Égalité
Fraternité*

Chapitre 03 : Annexes : Pour aller plus loin

- Passer un nombre indéfini de paramètres
- Retourner plusieurs objets dans une fonction
- Utiliser plusieurs return dans une fonction
- Utilisation de across : règle de renommage
- {purrr} : Contrôler le format de sortie
- {purrr} : Nommer les éléments d'une liste
- {purrr} : modifier le comportement d'une fonction

Annexes: Pour aller plus loin

Passer un nombre indéfini de paramètres

Il est possible de prévoir un nombre indéfini de paramètres lors de l'élaboration d'une fonction avec

Cela permet d'accéder aux paramètres d'une “sous-fonction” utilisée dans le corps de la fonction créée.

```
# Fonction pour lire le 2e onglet d'un fichier Excel
read.xlsx.onglet2 <- function(xlsxFile, ...){
  x <- read.xlsx(xlsxFile, sheet = 2, ...)
  return(x)
}

# Fonctionne
mes_donnees <- read.xlsx.onglet2("mon_fichier.xlsx")
# Fonctionne aussi !
mes_donnees <- read.xlsx.onglet2("mon_fichier.xlsx",
  rows = 3:40, cols = 1:7, fillMergedCells = TRUE)
```

Autres paramètres de la fonction `openxlsx::read.xlsx`

Passer un nombre indéfini de paramètres

Les ... permettent également de prévoir que l'utilisateur puisse passer un nombre indéfini de paramètres

```
moyenne_par_groupe <- function(df, var_moyenne, ...){  
  moyennes <- df %>%  
    group_by(...) %>%  
    summarise(moyenne = mean({{ var_moyenne }}), na.rm = TRUE))  
  return(moyennes)  
}
```

le troisième argument accepte un nombre indéfini de variables

toutes ces variables seront passées dans l'instruction group_by pour calculer les moyennes pour chacun des groupes

on peut utiliser la fonction en renseignant le nombre de variables souhaitées

- 1 seule :

```
diamonds %>% moyenne_par_groupe(var_moyenne = price, cut)
```

| cut | moyenne |
|-------------|---------|
| <ord> | <db7> |
| 1 Fair | 4359. |
| 2 Good | 3929. |
| 3 Very Good | 3982. |
| 4 Premium | 4584. |
| 5 Ideal | 3458. |

- 3 variables :

```
diamonds %>% moyenne_par_groupe(var_moyenne = price, cut, color, clarity)
```

| cut | color | clarity | moyenne |
|---------|-------|---------|---------|
| <ord> | <ord> | <ord> | <db7> |
| 1 Fair | D | I1 | 7383 |
| 2 Fair | D | SI2 | 4355. |
| 3 Fair | D | S11 | 4273. |
| 4 Fair | D | V2 | 4513. |
| 5 Fair | D | VS1 | 2921. |
| 6 Fair | D | VVS2 | 3607 |
| 7 Fair | D | VVS1 | 4473 |
| 8 Fair | D | IF | 1620. |
| 9 Fair | E | I1 | 2095. |
| 10 Fair | E | SI2 | 4172. |

- ou plus...!

Retourner plusieurs objets dans une fonction

L'instruction `return()` ne peut renvoyer qu'un seul objet ⇒ pour renvoyer plusieurs éléments dans une fonction, il faut les placer dans une liste

```
eff_milliers <- function(x){
  x_milliers_arrondi <- round(x / 1000, digits = 0)
  x_milliers_3digits <- round(x / 1000, digits = 3)
  return(list(milliers_arrondi = x_milliers_arrondi,
              milliers_decimale = x_milliers_3digits))
}
```

```
> eff_milliers(245631)
```

```
$milliers_arrondi
[1] 246
```

Pour y accéder : `eff_milliers(245631)[[1]]`

ou `eff_milliers(245631)$milliers_arrondi`

```
$milliers_decimale
[1] 245.631
```

Pour y accéder : `eff_milliers(245631)[[2]]`

ou `eff_milliers(245631)$milliers_decimale`

Utiliser plusieurs Return dans une fonction

Il est possible de mettre plusieurs return dans le bloc d'instructions d'une fonction.

=> Dans ce cas, l'exécution s'arrête au premier return rencontré.

```
# Fonction qui lit tous les csv d'un répertoire indiqué en paramètre
# Si le répertoire ne contient pas de csv, pas d'erreur, la fonction
# renvoie NULL
lire_csv_dir <- function(dir) {
  # List des csv dans le répertoire
  list_csv_rep <- list.files(path = dir, pattern = "\\.csv$",
                               ignore.case = TRUE, full.names = TRUE)
  # Traitement conditionnel
  if (length(list_csv_rep) == 0) {
    print("Le répertoire ne contient pas de csv")
    return(NULL)
  } else {
    print(list_csv_rep)
    resultat <- map(list_csv_rep, read.delim)
    return(resultat)
  }
}

Exécution arrêtée au second return
Exécution arrêtée au premier return

> lire_csv_dir("/var/data/nfs/CERISE/00-Espace-Personnel/damien.dotta")
[1] "Le répertoire ne contient pas de csv"
NULL
> lire_csv_dir("/var/data/nfs/CERISE/00-Espace-Personnel/damien.dotta/CERISE/01-Espace-de-Partage
/Formations/Formations_R_initiation/Donnees")
[1] "/var/data/nfs/CERISE/00-Espace-Personnel/damien.dotta/CERISE/01-Espace-de-Partage/Formations
/Formations_R_initiation/Donnees/menus_2018_2019_toulouse.csv"
[2] "/var/data/nfs/CERISE/00-Espace-Personnel/damien.dotta/CERISE/01-Espace-de-Partage/Formations
/Formations_R_initiation/Donnees/type_plat.csv"
```

Utilisation de `across()` : Règle de renommage

Une règle de renommage des variables issues de `across` peut être donnée via l'argument `.names`.

On lui donne une chaîne de caractère concaténée où :

- `{.col}` désigne la variable à traiter
- `{.fn}` la fonction de traitement.

Par exemple : `.names = "{.col}_{.fn}"`

Si aucune règle de renommage n'est précisée, R se débrouille :

- Dans un `mutate`, les variables d'origine sont écrasées
- Dans un `summarise` avec un seul agrégat à calculer, conserve le nom des variables d'origine
- Dans un `summarise` avec plusieurs agrégats, suffixe la fonction d'agrégation au nom des variables d'origine

Utilisation de `across()` Règle de renommage

```
# Mutate avec across et une règle de renommage :
```

```
base_RP <- base_RP %>% mutate(  
  across(starts_with("PMEN"), eff_milliers, .names = "{.col}_milliers"))
```

| PMEN | PMENPSEUL | PMENSFAM | PMENCOUPSENF | PMENCOUPAENF | PMENFAMMONO | PMEN_milliers | PMENPSEUL_milliers |
|-------------|------------|------------|--------------|--------------|-------------|---------------|--------------------|
| 821.18714 | 67.90005 | 18.434908 | 221.21890 | 462.68665 | 50.946633 | 0.821 | 0.068 |
| 262.70264 | 33.76941 | 0.000000 | 62.91485 | 151.66554 | 14.352843 | 0.263 | 0.034 |
| 13795.00884 | 2987.24621 | 172.295878 | 3280.47666 | 5622.39654 | 1732.593550 | 13.795 | 2.987 |
| 1780.00000 | 185.00000 | 50.000000 | 440.00000 | 1030.00000 | 75.000000 | 1.780 | 0.185 |

De nouvelles variables sont ajoutées à la fin de la table

```
# calculer plusieurs agrégats sur plusieurs variables
```

```
agregats <- base_RP %>% summarise(  
  across(PMEN:PMENFAMMONO,  
    list(tot = sum, min = min, q1 = \((x) quantile(x, probs = 0.25),  
        med = median, q3 = \((x) quantile(x, probs = 0.75), max = max),  
    .names = "{.fn}. {.col}")))
```

| tot.PMEN | min.PMEN | q1.PMEN | med.PMEN | q3.PMEN | max.PMEN | tot.PMENPSEUL | min.PMENPSEUL | q1.PMENPSEUL | n |
|----------|----------|----------|----------|---------|----------|---------------|---------------|--------------|---|
| 69155448 | 0 | 195.8763 | 453.6295 | 1145 | 2102799 | 12218557 | 0 | 25 | 5 |

{purrr}: Contrôler le format de sortie

Possibilité de contrôler le format de sortie avec les dérivés de la fonction map

| fonction | sortie |
|----------------|-----------------------|
| map | liste |
| map_chr | vecteur de caractères |
| map_int | vecteur d'entiers |
| map dbl | vecteur de numériques |
| map_lgl | vecteur de booléens |

map_chr() permet d'obtenir les villes les plus peuplées sous forme d'un vecteur de caractères plutôt qu'une liste

```
# récupérer les villes les plus peuplées en 2020 pour chaque département
# sous forme de vecteur
villes <- departements_occitanie %>%
  map_chr(function(code_dep){
    base_RP %>%
      filter(DEP == code_dep & annee == 2020) %>%
      slice_max(PMEN) %>%
      pull(LIB_MOD)
  })

```

```
> villes
[1] "Pamiers"      "Narbonne"     "Rodez"        "Nîmes"        "Toulouse"     "Auch"         "Montpellier" "Cahors"       "Mende"        "Tarbes"
[11] "Perpignan"   "Albi"        "Montauban"
```

{purrr}: Nommer les éléments d'une liste

Retour sur le calcul des évolutions départementales des populations par composition familiale :

```
library(purrr)

departements_occitanie <- c("09", "11", "12", "30", "31", "32", "34",
                           "46", "48", "65", "66", "81", "82")

evolutions_occitanie <- departements_occitanie %>%
  map(calcul_evol_departement)
```

Par défaut, **les différents éléments de la liste résultats ne sont pas nommés**. On ne peut accéder aux différents éléments que par index :

```
> # Evolution pour l'Aveyron en utilisant le code dep
> evolutions_occitanie[["12"]]
NULL
```

{purrr}: Nommer les éléments d'une liste

La fonction `set_names()` du package purrr permet de nommer les éléments d'une liste en entrée :

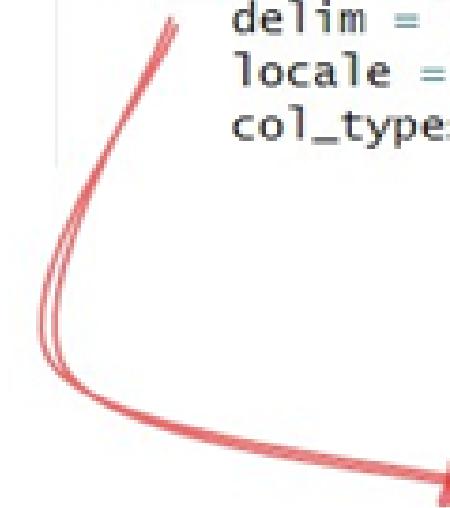
```
# On attribue comme noms aux éléments de la liste résultat
# les n° de départements correspondant
evolutions_occitanie <- departements_occitanie %>%
  map(calcul_evol_departement) %>%
  set_names(departements_occitanie)

# On peut désormais accéder aux données de l'Aveyron de deux façons
evolutions_occitanie[[3]]      # Par index (l'Aveyron est le 3e département de notre vecteur)
evolutions_occitanie[["12"]]    # Par nom
```

{purrr}: Modifier le comportement d'une fonction

Les fonctions de {purrr} ne renvoient un résultat que s'il n'y a aucune erreur dans l'ensemble... sinon, l'exécution est arrêtée et aucun résultat n'est retourné, même si l'erreur est à la fin !

```
import_RP <- c("donnees_RP2024.csv", "donnees_RP2019.csv", "donnees_RP2014.csv", "donnees_RP2009.csv") %>%  
  map(read_delim,  
       delim = ";",  
       locale = locale(decimal_mark = ".")  
       col_types = cols(CODGEO = col_character(), .default = col_double()))
```



Error in `map()`:
i In index: 1.
Caused by error:
! 'donnees_RP2024.csv' does not exist in current working directory

si l'un des fichiers n'existe pas :
toute l'exécution est arrêtée !

{purrr}: Modifier le comportement d'une fonction

`safely()` ⇒ retourne le résultat et le message d'erreur de la fonction

```
import_RP <- c("donnees_RP2024.csv", "donnees_RP2019.csv", "donnees_RP2014.csv", "donnees_RP2009.csv") %>%
  map(safely(read_delim),
      delim = ";",
      locale = locale(decimal_mark = "."),
      col_types = cols(CODEGEO = col_character(), .default = col_double()))
[[1]]
[[1]]$result
NULL

[[1]]$error
<simpleError: 'donnees_RP2024.csv' does not exist in current working directory
```

en utilisant `safely()`

on obtient une liste avec le résultat de l'import
et le message d'erreur éventuel

on peut ensuite récupérer les données de cette façon :

```
donnees_RP <- import_RP %>%
  map("result") %>%
  compact() %>%
  reduce(full_join)
```

ne conserve que les éléments "result" de la liste

supprime les éléments NULL

joint les fichiers les uns après les autres

| CODEGEO | PMEN19 | PMEN14 | PMEN09 |
|---------|--------|--------|--------|
| | <chr> | <dbl> | <dbl> |
| 1 01001 | 821 | 765 | 787 |
| 2 01002 | 263 | 256 | 209 |
| 3 01004 | 13795 | 13547 | 12781 |
| 4 01005 | 1780 | 1602 | 1580 |

{purrr}: Modifier le comportement d'une fonction

`possibly()` ⇒ retourne le résultat ou une valeur par défaut en cas d'erreur

```
import_RP <- c("donnees_RP2024.csv", "donnees_RP2019.csv", "donnees_RP2014.csv", "donnees_RP2009.csv") %>%
  map(possibly(read_delim, otherwise = NULL),
    delim = ";",
    locale = locale(decimal_mark = "."),
    col_types = cols(CODEO = col_character(), .default = col_double()))
```

en utilisant `possibly()`
on obtient une liste avec le résultat de l'import
ou une valeur choisie en cas d'erreur

valeur à attribuer en cas
d'erreur de la fonction

[[1]]
NULL

[[2]]
A tibble: 4 x 2
CODEO PMEN19
<chr> <dbl>
1 01001 821
2 01002 263
3 01004 13795
4 01005 1780

on peut ensuite récupérer les données de cette façon :

```
donnees_RP <- import_RP %>%
  compact() %>%
  reduce(full_join)
```

supprime les éléments NULL

joint les fichiers les uns après les autres

| | CODEO | PMEN19 | PMEN14 | PMEN09 |
|---|-------|--------|--------|--------|
| 1 | 01001 | 821 | 765 | 787 |
| 2 | 01002 | 263 | 256 | 209 |
| 3 | 01004 | 13795 | 13547 | 12781 |
| 4 | 01005 | 1780 | 1602 | 1580 |



*Liberté
Égalité
Fraternité*

Chapitre 04 : Liens utiles

Liens utiles



Ecriture de fonctions

- Ecriture de fonctions :

<https://juba.github.io/tidyverse/14-fonctions.html> <https://juba.github.io/tidyverse/17-if-boucles.html>

- BONUS ! gestion des erreurs :

<https://thinkr.fr/controle-et-gestion-des-erreurs-dans-r/>

- Programmer avec le tidyverse :

<https://juba.github.io/tidyverse/19-programmer-tidyverse.html>

<https://thinkr.fr/comment-creer-des-fonctions-dans-le-tidyverse-avec-la-tidyeval-et-le-stash-stash/>

Itérations de fonctions

- Utilisation de across() :

<https://juba.github.io/tidyverse/15-dplyr-avance.html#sec-across>

<https://www.icem7.fr/r/across-plus-puissant-flexible-quil-ny-parait/>

- Introduction à purrr :

<https://dcl-prog.stanford.edu/iteration.html>

<https://thinkr.fr/code-ronrone-purrr/>

<https://speakerdeck.com/jennybc/purrr-workshop?slide=91>

- Aides mémoire :

<https://rstudio.com/resources/cheatsheets/>