TP1 : Manipulations de base avec le langage R

Zakaria ELhajoui

8 avril 2022

Contents

1 Objectifs du TP:

Les objectifs fixés pour ce TP sont :

- Se familiariser avec l'environement ***RStudio***
- Utiliser quelques fonctions de base du langage ***R***
- Manipuler et transformer les données avec les packages ***dplyr***, ***tidyr*** et d'autres.
- Créer des représentations graphiques pour les données avec les fonctions natives et le package ***ggplot2***
- CTRL + ALt + I

2 Prise en main

2.1 Opérations de base

Pour conserver le résultat d'une opération, on peut le stocker dans un objet à l'aide de l'opérateur d'assignation <-. Cette "flèche" stocke ce qu'il y a à sa droite dans un objet dont le nom est indiqué à sa gauche.

```
x <- 5
x = 10
x
```

```
## [1] 10
```

On préviligie <- pour effectuer l'assignantion.

Pour afficher le contenu d'un objet, on exécute une commande comportant juste le nom de l'objet.

```
x
```

```
## [1] 10
```

```
# Opérations arithmétiques entre variables
y <- 10
y + 5
```

```
## [1] 15
```

```
y * 10
```

```
## [1] 100
```

```
y / 2
```

```
## [1] 5
```

```
y - 6
```

```
## [1] 4
```

2.2 Vecteurs

2.2.1 Création d'un vecteur

Imaginons maintenant qu'on a demandé le salaire de 5 personnes et qu'on souhaite calculer leur salaire moyen. On pourrait créer autant d'objets que de salaires et faire l'opération mathématique qui va bien :

```
# Création des variables salaires (salaire1,....,salaire5)

salaire1 <- 4000
salaire2 <- 2600
salaire3 <- 3000
salaire4 <- 6500
salaire5 <- 7000
salaire_moyen <- (salaire1 + salaire2 + salaire3 + salaire4 + salaire5) / 5

salaire_moyen
```

```
## [1] 4620
```

Cette manière de faire n'est clairement pas pratique du tout. On va donc plutôt stocker l'ensemble de nos salaires dans un seul objet, de type vecteur, avec la syntaxe suivante :

```
# Créer le vecteur salaires qui regroupe un ensemble de salaires

salaires <- c(4000,2600,3000,6500,7000)
```

Un vecteur dans R est un objet qui peut contenir plusieurs informations du même type, potentiellement en très grand nombre.

Pour afficher le contenu de l'objet salaire, on a qu'à écrire son nom puis exécuter la commande.

```
# Afficher le contenu du vecteurs salaires

salaires
```

```
## [1] 4000 2600 3000 6500 7000
```

Supposant qu'on veut fixer une augmentation de salaire à tous les employés de 5%, on peut procéder plus facilement avec un objet de type vecteur puisque ce dernier, lorsqu'on lui applique une opération, celle-ci s'applique à toutes les valeurs qu'il contient.

```
# Augmentation des salaires

salaire_modifie <- salaires * 1.05
salaire_modifie
```

```
## [1] 4200 2730 3150 6825 7350
```

Un vecteur peut contenir des nombres, mais il peut aussi contenir du texte. Imaginons qu'on a demandé aux 5 mêmes personnes leur niveau de diplôme : on peut regrouper l'information dans un vecteur de chaînes de caractères. Une chaîne de caractère contient du texte libre, délimité par des guillemets simples ou doubles.

```
# Création d'un vecteur diplomes avec les niveaux suivants : CAP, BAC, BAC+2, BAC+2, BAC+3 puis l'affic

diplomes <- c("CAP","BAC", "BAC+2", "BAC+2", "BAC+2")
diplomes
```

```
## [1] "CAP"   "BAC"   "BAC+2" "BAC+2" "BAC+2"
```

Génération rapide d'un vecteur contenant toutes tous les nombres entre deux valeurs.

```
# Génération rapide d'un vecteur x  avec l'opérateur :

x <- 1:25
x
```

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

Utilisation de quelques fonction pour créer un vecteur :

```
# Utilisation de la fonction seq pour créer un vecteur v

v <- seq(4,15,2)
v
```

```
## [1]  4  6  8 10 12 14
```

```
# Utilisation de la fonction rep pour créer un vecteur w

w <- rep("a",10)
w
```

```
##  [1] "a" "a" "a" "a" "a" "a" "a" "a" "a" "a"
```

2.2.2 Accès aux éléments d'un vecteur

Notons qu'on peut accéder à un élément particulier d'un vecteur en faisant suivre le nom du vecteur de crochets contenant le numéro de l'élément désiré.

```
# Accès à un seul éléments
x[1]
```

```
## [1] 1
```

```r
salaire_modifie[2]
```

```
## [1] 2730
```

```r
# Accès à plusieurs éléments

x[4:7]
```

```
## [1] 4 5 6 7
```

```r
x <- 1:15

# Accès conditionnel
x[x>9]
```

```
## [1] 10 11 12 13 14 15
```

```r
# Accès à tous les éléments sauf quelques uns
x[-4]
```

```
##  [1]  1  2  3  5  6  7  8  9 10 11 12 13 14 15
```

2.2.3 Application de quelques fonctions sur les vecteurs

Pour avoir le nombre d'éléments dans un vecteur:

```r
# Nombre d'élément d'un vecteur avec la fonction length

length(salaire_modifie)
```

```
## [1] 5
```

```r
salaire_modifie |> length()
```

```
## [1] 5
```

```r
# Queleques fonctions : # max, mean, sd, sum, prod, cumsum, cumprod, range, unique
# appliquée sur salaires
salaire_modifie
```

```
## [1] 4200 2730 3150 6825 7350
```

```r
max(salaire_modifie)
```

```
## [1] 7350
```

```r
mean(salaire_modifie)
```

```
## [1] 4851
```

```r
sd(salaire_modifie)
```

```
## [1] 2118.816
```

```r
sum(salaire_modifie)
```

```
## [1] 24255
```

```r
cumsum(salaire_modifie)
```

```
## [1]  4200  6930 10080 16905 24255
```

```r
cumprod(salaire_modifie)
```

```
## [1] 4.200000e+03 1.146600e+07 3.611790e+10 2.465047e+14 1.811809e+18
```

```r
range(salaire_modifie)
```

```
## [1] 2730 7350
```

```r
unique(salaire_modifie)
```

```
## [1] 4200 2730 3150 6825 7350
```

```r
# Utilser la fonction unique sur la variable diplomes

unique(diplomes)
```

```
## [1] "CAP"   "BAC"   "BAC+2"
```

La fonction sort permet d'ordonner les éléments d'un vecteur :

```r
# Fonction sort sur le vecteur ages
ages <- c(25,13,20,30,29,41,35,31,60)
sort(ages, decreasing = T)
```

```
## [1] 60 41 35 31 30 29 25 20 13
```

```r
ages |> sort(decreasing = TRUE)
```

```
## [1] 60 41 35 31 30 29 25 20 13
```

Ou avec la fonction rev qui permet d'inverser l'ordre des éléments d'un vecteur:

```r
# Fonction rev

rev(sort(ages))
```

```
## [1] 60 41 35 31 30 29 25 20 13
```

```r
ages |> sort() |> rev()
```

```
## [1] 60 41 35 31 30 29 25 20 13
```

Représentation graphique d'un vecteur

```r
# Utilisation de la fonction hist sur la variable ages

hist(x=ages,main="Histogrammes des ages", col="skyblue")
```

2.3 Facteurs

En ***R***, un facteur (factor, en anglais) est un vecteur dont les éléments ne peuvent prendre que des modalités prédéfinies. Ce qui caractérise un facteur en R est le fait qu'elle dispose de l'attribut Levels (niveaux). En pratique, un facteur est typiquement utilisé pour stocker les valeurs observées d'une variable catégorielle (couleur, sexe, jours de la semaine, religion, ...). Pour créer un facteur, on utilise la fonction factor(). Typiquement, on commence par définir un vecteur classique (caractère, numérique ou logique) et puis on le transforme en facteur.

```r
# Création d'une variable puis un facteur
fh <- c("H", "F", "F", "F", "F", "H", "F", "H", "H", "F", "F", "F")
sexe <- factor(fh)
sexe
```

```
##  [1] H F F F F H F H H F F F
## Levels: F H
```

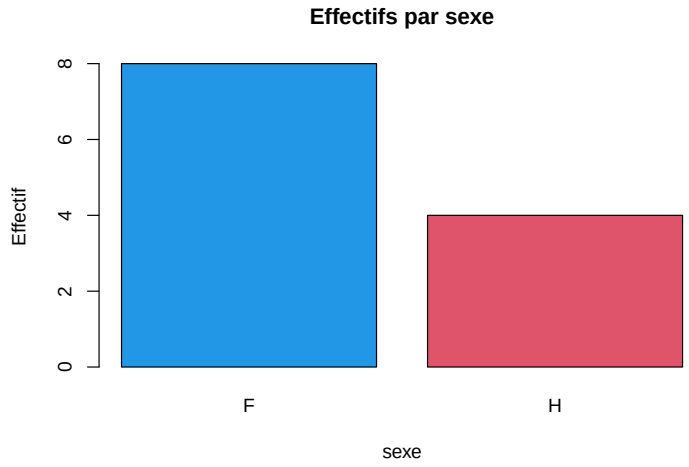
Un facteur inclue non seulement les valeurs de la variable catégorique correspondante mais aussi les différents niveaux possibles de cette variable (même éventuellement ceux qui ne sont pas représentés dans les données).

Beaucoup de fonctions en ***R*** réservent un traitement spécial pour les facteurs, c'est pourquoi il est important de bien définir un facteur comme tel pour qu'il soit correctement traité par ***R***. Voici un exemple.

Dans le cas d'un vecteur, la fonction plot émet une erreur : plot(fh)

Par contre, dans le cas d'un facteur on aura :

```
# Tracer un graphique de la variable sexe avec la fonction plot
plot(sexe, main = "Effectifs par sexe", xlab ="sexe", ylab="Effectif", col=c(4,10))
```



En interne, ***R*** encode les valeurs d'un facteur sous forme d'entiers: 1 pour le premier niveau, 2 pour le le deuxième niveau, et ainsi de suite. Pour voir cela, examinons la structure de l'objet sexe.

```
# Fonction str sur la variable sexe
sexe |> str()
```

```
##  Factor w/ 2 levels "F","H": 2 1 1 1 1 2 1 2 2 1 ...
```

Cette sortie indique que la variable ***sexe*** est bel et bien un facteur avec deux modalités "F" ou "H", encodées comme 1 et 2, respectivement.

2.3.1 L'argument levels

L'argument optionnel levels de la fonction ***factor()*** spécifie quels sont les niveaux possibles du facteur et quel sera l'ordre d'affichage de ces derniers. Par défaut les niveaux possibles d'un facteur sont les valeurs uniques du vecteur utilisé pour le créer et, toujours par defaut, les niveaux d'un facteur sont affichés par ordre alphanumérique croissant. À ce propos, dans les sorties ci-dessus, on voit bien que "F" apparaît toujours avant "H" . On peut changer cela facilement en changeant l'ordre des niveaux comme ceci:

```
# Changer les niveaux possibles du facteur avec la fonction factor et levels

sexe <- factor(sexe, levels = c('H','F'))
sexe
```

```
##  [1] H F F F F H F H H F F F
## Levels: H F
```

Dans l'exemple suivant, de représente le résultat du lancement d'un dé dix fois.

```
x <- c(3, 2, 2, 1, 3, 1, 3, 1, 6, 6)
de <- factor(x)
de
```

```
##  [1] 3 2 2 1 3 1 3 1 6 6
## Levels: 1 2 3 6
```

Si on exécute le code suivant, on obtient un Warning et une valeur manquante dans de.

```
de[11] <- 5
```

```
## Warning in `[<-.factor`(`*tmp*`, 11, value = 5): invalid factor level, NA
## generated
```

```
de
```

```
##  [1] 3    2    2    1    3    1    3    1    6    6    <NA>
## Levels: 1 2 3 6
```

Cela est normal puisque 5 ne fait pas partie de la liste des modalités de l'objet ***de***. Pour corriger cela, il suffit de donner la liste complète des modalités lorsqu'on crée le facteur.

```
# Préciser les levels() pendant la création d'un facteur
x <- c(3, 2, 2, 1, 3, 1, 3, 1, 6, 6)
de <- factor(x, levels = 1:6)
de
```

```
##  [1] 3 2 2 1 3 1 3 1 6 6
## Levels: 1 2 3 4 5 6
```

2.3.2 La fonction levels()

La fonction levels() sert à extraire les niveaux possibles d'un facteur:

```
# Afficher les niveaux d'un facteur
sexe |> levels()
```

```
## [1] "H" "F"
```

```
de |> levels()
```

```
## [1] "1" "2" "3" "4" "5" "6"
```

Cette fonction peut aussi être utilisée pour modifier, ajouter, ou supprimer des niveaux. Voici quelques exemples.

- Renommer un niveau spécifique:

```r
sexe |> levels()
```

```
## [1] "H" "F"
```

```r
levels(sexe)[1] <- "Homme"
levels(sexe)[2] <- "femme"
sexe
```

```
##  [1] Homme femme femme femme femme Homme femme Homme Homme femme femme femme
## Levels: Homme femme
```

- Renommer tous les niveaux:

```r
levels(sexe) <- c("H","F")
sexe
```

```
##  [1] H F F F F H F H H F F F
## Levels: H F
```

- Ajouter des niveaux à un facteur existant:

```r
levels(sexe)[3] <- "G"
sexe[13] <- "G"
sexe
```

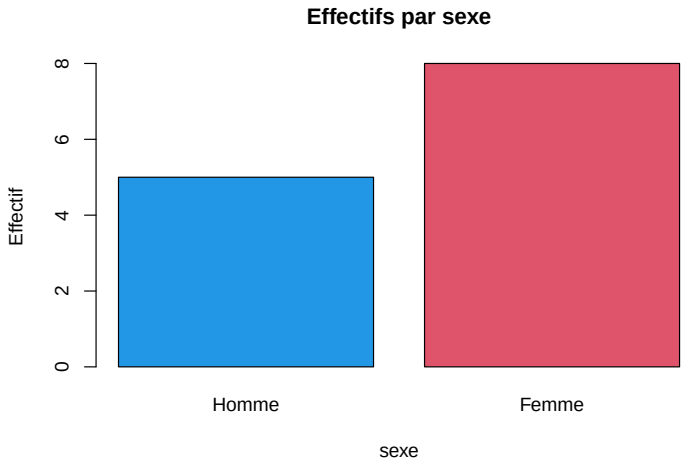
```
##  [1] H F F F F H F H H F F F G
## Levels: H F G
```

- Supprimer/regrouper des niveaux:

```r
levels(sexe) <- c("Homme","Femme","Homme","Femme")
sexe
```

```
##  [1] Homme Femme Femme Femme Femme Homme Femme Homme Homme Femme Femme Femme
## [13] Homme
## Levels: Homme Femme
```

```r
# Représentation graphique de l'objet sexe
plot(sexe, main = "Effectifs par sexe", xlab ="sexe", ylab="Effectif", col=c(4,10))
```



3 Manipulation des données

dplyr est l'un des packages les plus populaires et les plus utiles de l'univers ***R***. Il couvre tous les aspects de la manipulation des données : filtrage, sélection, tri et bien d'autres.

Dans notre exemple, nous allons découvrir les noms de filles les plus populaires de 2015 et obtenir plus de détails sur leur popularité de 1880 à 2017.

3.1 Packages

Nous allons analyser les noms de bébés du paquet *babynames*. Veuillez vous assurer que vous l'avez installé avant d'essayer de le charger. Il en va de même pour *dplyr*.

```
#install.packages("babynames")
#install.packages("dplyr")
```

- *babynames*: includes data for this tutorial
- *dplyr*: includes all data manipulation

```
# Chargement des packages
library(babynames)
library(dplyr)
```

3.2 Préparation des données

On charge notre dataset avec la commande **data()** et obtenir un aperçu de sa structure avec les commandes : **tbl_df** et **glimpse**.

```
data("babynames")
glimpse(babynames)
```

```
## Rows: 1,924,665
## Columns: 5
## $ year <dbl> 1880, 1880, 1880, 1880, 1880, 1880, 1880, 1880, 1880, 1880, 1880,~
## $ sex  <chr> "F", "F", "F", "F", "F", "F", "F", "F", "F", "F", "F", "F", "F", ~
## $ name <chr> "Mary", "Anna", "Emma", "Elizabeth", "Minnie", "Margaret", "Ida",~
## $ n    <int> 7065, 2604, 2003, 1939, 1746, 1578, 1472, 1414, 1320, 1288, 1258,~
## $ prop <dbl> 0.07238359, 0.02667896, 0.02052149, 0.01986579, 0.01788843, 0.016~
```

```
tbl_df(babynames)
```

```
## Warning: `tbl_df()` was deprecated in dplyr 1.0.0.
## Please use `tibble::as_tibble()` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was generated.
```

```
## # A tibble: 1,924,665 x 5
##     year sex   name          n   prop
##    <dbl> <chr> <chr>     <int>  <dbl>
##  1  1880 F     Mary       7065 0.0724
##  2  1880 F     Anna       2604 0.0267
##  3  1880 F     Emma       2003 0.0205
##  4  1880 F     Elizabeth  1939 0.0199
##  5  1880 F     Minnie     1746 0.0179
##  6  1880 F     Margaret   1578 0.0162
##  7  1880 F     Ida        1472 0.0151
##  8  1880 F     Alice      1414 0.0145
##  9  1880 F     Bertha     1320 0.0135
## 10  1880 F     Sarah      1288 0.0132
## # ... with 1,924,655 more rows
```

Il y a cinq colonnes : *année*, *sexe*, *nom*, un compte n, et la proportion correspondante *prop*.

Affichage de quelques lignes du dataset :

```
babynames |> head()
```

```
## # A tibble: 6 x 5
##    year sex   name          n   prop
##   <dbl> <chr> <chr>     <int>  <dbl>
## 1  1880 F     Mary       7065 0.0724
## 2  1880 F     Anna       2604 0.0267
## 3  1880 F     Emma       2003 0.0205
## 4  1880 F     Elizabeth  1939 0.0199
## 5  1880 F     Minnie     1746 0.0179
## 6  1880 F     Margaret   1578 0.0162
```

```
babynames |> tail(n=3)
```

```
## # A tibble: 3 x 5
##    year sex   name      n       prop
##   <dbl> <chr> <chr> <int>      <dbl>
## 1  2017 M     Zylin     5 0.00000255
## 2  2017 M     Zylis     5 0.00000255
## 3  2017 M     Zyrie     5 0.00000255
```

3.3 Filtrage des données

Nous voulons garder uniquement les données de d'une année spécifique(2015 par exemple) et pouvons obtenir cela avec **filter()**.

```
most <- babynames |> filter(year == 2017) |> filter(sex == 'F')
most
```

```
## # A tibble: 18,309 x 5
##     year sex   name          n    prop
##    <dbl> <chr> <chr>     <int>   <dbl>
##  1  2017 F     Emma      19738 0.0105
##  2  2017 F     Olivia    18632 0.00994
##  3  2017 F     Ava       15902 0.00848
##  4  2017 F     Isabella  15100 0.00805
##  5  2017 F     Sophia    14831 0.00791
##  6  2017 F     Mia       13437 0.00717
##  7  2017 F     Charlotte 12893 0.00688
##  8  2017 F     Amelia    11800 0.00629
##  9  2017 F     Evelyn    10675 0.00569
## 10  2017 F     Abigail   10551 0.00563
## # ... with 18,299 more rows
```

Nous avons un dataframe avec tous les noms, mais seulement pour l'année 2015.

Nous devons également filtrer pour le sexe, donc ajouter un filtre supplémentaire.

```
most_female <- babynames |> filter(year == 2017) |> filter(sex == 'F')
```

Afficher n dernières observations de notra dataset :

```
most_female |> tail(7)
```

```
## # A tibble: 7 x 5
##    year sex   name        n       prop
##   <dbl> <chr> <chr>   <int>      <dbl>
## 1  2017 F     Zyannah     5 0.00000267
## 2  2017 F     Zyiana      5 0.00000267
## 3  2017 F     Zyianna     5 0.00000267
## 4  2017 F     Zykira      5 0.00000267
## 5  2017 F     Zymia       5 0.00000267
## 6  2017 F     Zynia       5 0.00000267
## 7  2017 F     Zyonn       5 0.00000267
```

3.4 Tier les données

Les données peuvent être triées avec **arrange()**. Cette fonction peut avoir plusieurs colonnes de tri. L'ordre par défaut est ascendant. Si un ordre descendant est nécessaire, il peut être effectué avec la fonction **desc()**.

Dans notre exemple, les données sont triées par ***sexe*** et par proportion décroissante ***prop***.

```
most_female <- babynames |> filter(year == 2015) |> filter(sex == 'F') |> arrange(name, desc(prop))

most_female
```

```
## # A tibble: 19,074 x 5
##     year sex   name          n       prop
##    <dbl> <chr> <chr>     <int>      <dbl>
##  1  2015 F     Aabha         7 0.0000036
##  2  2015 F     Aabriella     5 0.00000257
##  3  2015 F     Aada          5 0.00000257
##  4  2015 F     Aadhira       8 0.00000411
##  5  2015 F     Aadhya      265 0.000136
##  6  2015 F     Aadrika       9 0.00000463
##  7  2015 F     Aadvika       7 0.0000036
##  8  2015 F     Aadya       159 0.0000817
##  9  2015 F     Aafia         6 0.00000308
## 10  2015 F     Aahana       31 0.0000159
## # ... with 19,064 more rows
```

3.5 Sélection des colonnes

Souvent, toutes les colonnes ne sont pas nécessaires pour une analyse ultérieure. La fonction **select()** peut être appliquée pour réduire le nombre de colonnes. Cela peut être fait en définissant les colonnes à conserver ou à exclure.

Dans notre exemple, seul le *nom* est conservé. Une alternative aurait été ***select(-year, -sex, -n, -prop)***.

```
most_female <- babynames %>%
  filter(year == 2015) %>%
  filter(sex == 'F') %>%
  arrange(name, desc(prop)) %>%
  select (name)
```

Seul le Top N sera analysé, on applique à la fin la fonction ***head*** avec le paramètre $n = N$.

```
most_female <- babynames |>
  filter(year == 2015) |>
  filter(sex == 'F') |>
  arrange(name, desc(prop)) |>
  select(name) |>
  head(5)
most_female
```

```
## # A tibble: 5 x 1
##   name
##   <chr>
## 1 Aabha
## 2 Aabriella
## 3 Aada
## 4 Aadhira
## 5 Aadhya
```

Ce sont par ordre décroissant les noms de filles les plus célèbres de 2015.

3.6 Représentation graphique

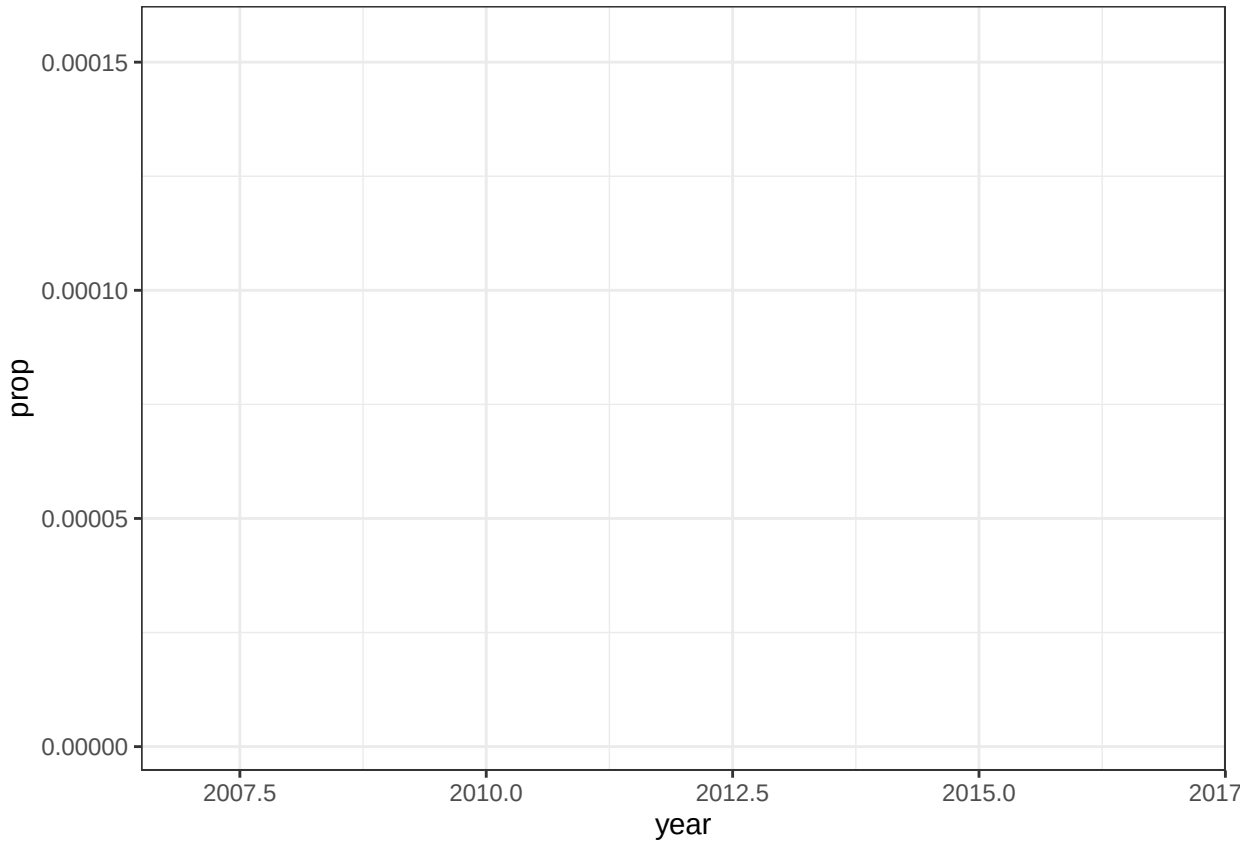
Maintenant, il faut montrer l'évolution de ces noms. Le dataframe original est filtré pour ces noms. La fonction **mutate()** peut créer de nouvelles variables.

```
 namesToplot <- babynames |>
 filter(sex =='F')  |>
#mutate(prop = prop * 100) |>
filter(name %in% most_female$name)
```

Le graphique est créé avec **ggplot()**. Il montre le Top N des noms de 2015 et leur évolution depuis 1880.

```
#install.packages("ggplot2")
library(ggplot2)
```

```
g <- ggplot(namesToplot,aes(year, prop, color = name))
# g <- g + geom_line + xlab("Year [-]") + ylab("Proportion[-]") + ggtitle("Les noms les plus populaires
g <- g + theme_bw()
g
```



Emma était très célèbre en 1880 et dans les décennies suivantes et n'a connu un renouveau que récemment. Les autres noms du Top N n'ont été connus que récemment.

4 Réorganisation des données

4.1 Introduction

Les données peuvent être remodelées en :

- transformant des colonnes en lignes
- des lignes en colonnes

Ceci est directement combiné avec la notion des données larges et longues (**wide and tidy-data**).

Pour plus d'informations, consultez sur le lien : [Wide and Tidy-data] (https://r4ds.had.co.nz/tidy-data.html).

Soit l'exemple suivant :

```
# Création d'un dataframe
school_wide <- data.frame(nom = c("Rayane", "Karima", "Anir","Yasmine"),
             mathématique = c(12, 13, 14, 15),
             sport = c(13, 11, 12, 14),
             art = c(13, 12, 11, 10),
           physique = c(13, 12, 11, 13))
school_wide
```

```
##       nom mathématique sport art physique
## 1  Rayane           12    13  13       13
## 2  Karima           13    11  12       12
## 3    Anir           14    12  11       11
## 4 Yasmine           15    14  10       13
```

On utilise le package ***tidyr*** pour faire les transformations des données. On doit l'installer puis le charger.

```
# install.packages("tidyr")
```

4.2 Transformation des données larges vers données longues

Pour transformer des données de ***wide***(large) à ***tidy***(longue), on peut utiliser les fonctions du paquet **tidyr**. La fonction concernée est **gather()**. On doit préciser quelques paramètres :

- le paramètre ***key***, qui représente les colonnes. On appelle la colonne résultante matière.
- le paramètre ***value***, qui sera le nom des colonnes contenant les valeurs.
- Enfin, on définit les colonnes à utiliser pour la transformation. Ici, le range des données est ***2:5***.

```
# ??tidyr
```

4.3 Transformation des données longues vers données larges

La fonction ayant l'effet inverse de **gather()** est **spread()**. On lui passe exactement les mêmes paramètres :

- ***key*** : on définit ici le nom de la colonne clé (key)
- ***value*** : on lui passe le nom choisi de la colonne avec les valeurs

4.4 Pourquoi y a-t-il deux formats différents ?

Les deux ont le droit d'exister. Les données larges proviennent généralement d'autres programmes. Excel ou SPSS travaillent généralement avec des données larges.

Pour des tâches spécifiques, il est préférable de transformer les données larges (également appelées données désordonnées) en données ordonnées. Les deux exemples qui suivent, vont montrer l'intérêt de cette transformation.

4.4.1 Exemple 1 : Calculer les moyennes par groupe

Supposons qu'on veut calculer la moyenne des notes par matière. Avec des données larges, on procède comme ceci :

Pour obtenir le résultat désiré, on répéte le code plusieurs fois, ce qui est à éviter. Pour trois colonnes, c'est acceptable, mais pas pour plus.

Voici une solution basée sur tidy-data. On doit charger le package ***dplyr*** et utiliser les fonctions ***group_by()*** et **summurise()**. Le code deviendra beaucoup plus propre.

dplyr et ***tidyr*** font partie de tidyverse, qui est un ensemble de paquets(packages) différents.

4.4.2 Exemple 2 : Représentation graphique avec ggplot2

Supposons qu'on veut tracer les résultats obtenus avec ***ggplot2***. Ceci est fait avec trois lignes de code avec les données longues. Dans le cas contraire,une représentation similaire basée sur wide-data résultera en un code beaucoup plus complexe.