

Dates 1: Reconnaître et Savoir Formatter des Dates

GRAPH Network & WHO, supported by the Global Fund to fight HIV, TB & Malaria

October 2023

This document is a draft of a lesson made by the GRAPH Network, a non-profit headquartered at the University of Geneva Global Health Institute, in collaboration with the World Health Organization, under a Global Fund 2023 grant to create e-learning modules to build in-country data capacity for epidemiological and impact analysis for National HIV, TB and malaria programs

Introduction
Learning Objectives
Packages
Datasets
PID Malawi
Séjours hospitaliers
Introduction aux Dates en R
Conversion de chaînes de caractères en dates
Avec les fonctions natives de R (Base R)
Avec lubridate
Gérer des dates mixtes avec <code>lubridate:::parse_date_time()</code>
Modifier l'Affichage des Dates
EN RÉSUMÉ
Answer Key

```
## [1] "fr_FR.UTF-8"
```

Introduction

Comprendre comment manipuler les dates est une compétence cruciale lorsque l'on travaille avec des données de santé. Les calendriers de vaccination, la surveillance des maladies, et les changements dans les indicateurs de santé à l'échelle de la population nécessitent tous de travailler avec des dates. Dans cette leçon, nous allons apprendre comment R stocke et affiche les dates, ainsi que comment les manipuler, les analyser et les formater efficacement. Commençons !

Learning Objectives

- Vous comprenez comment les dates sont stockées et manipulées dans R
- Vous comprenez comment convertir des chaînes de caractères en dates
- Vous savez gérer les colonnes de dates de formats mixtes
- Vous êtes capable de changer l'affichage des dates

Packages

Veuillez charger les packages nécessaires pour cette leçon avec le code ci-dessous :

```
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse,
                lubridate)
```

Datasets

PID Malawi

Le premier jeu de données que nous utiliserons contient des données liées aux pulvérisation intradomiciliaire d'insecticide (PID) dans le cadre des efforts de lutte contre le paludisme entre 2014 et 2019 à Illovo, au Malawi. Notez que le jeu de données est au format long, chaque ligne représentant une période pendant laquelle les PID ont eu lieu dans un village. Étant donné que le même village est pulvérisé à plusieurs reprises à différents moments, les noms de village se répètent. Les jeux de données au format long sont souvent utilisés lorsque l'on traite des données de séries chronologiques avec des mesures répétées, car ils sont plus faciles à manipuler pour les analyses et les visualisations.

```
pid <- read_csv(here("data/Illovo_PID.csv"))
```

```
pid
```

```
## # A tibble: 5 × 9
##   village      cible_PID  reelle_PID couverture_p
##   <chr>        <dbl>     <dbl>       <dbl>
## 1 Mess          87        64         73.6
## 2 Nkombedzi    183       169        92.4
## 3 B Compound   16         16         100
## 4 D Compound   3          2          66.7
## 5 Post Office  6          3          50
##   date_debut_defaut date_fin_defaut date_debut_typique
##   <date>           <date>           <chr>
## 1 2014-04-07     2014-04-17     07/04/2014
## 2 2014-04-22     2014-04-27     22/04/2014
## 3 2014-05-13     2014-05-13     13/05/2014
## 4 2014-05-13     2014-05-13     13/05/2014
## 5 2014-05-13     2014-05-13     13/05/2014
## # ℹ 2 more variables: date_debut_longue <chr>,
## #   date_debut_mixte <chr>
```

Les variables incluses dans le jeu de données sont les suivantes :

- **village**: nom du village où la PID a eu lieu
- **cible_PID**: nombre de structures ciblées pour la PID
- **reelle_PID**: nombre de structures réellement PID
- **date_debut_defaut**: le jour où la PID a commencé, au format par défaut “aaaa-mm-jj”
- **date_fin_defaut**: jour où la PID s'est terminée, au format par défaut “aaaa-mm-jj”
- **date_debut_typique**: jour où la pulvérisation a commencé, au format “jj/mm/aaaa”
- **date_debut_longue**: jour où la PID a commencé, avec le mois écrit en entier, jour à deux chiffres, puis année à quatre chiffres
- **date_debut_mixte**: jour où la PID a commencé avec un mélange de différents formats

Séjours hospitaliers

Le deuxième jeu de données constitue des données factices de séjours hospitaliers simulés. Il contient les dates d'admission et de sortie de 150 patients. Tout comme le jeu de données PID, les dates d'admission sont formatées de différentes manières afin que vous puissiez exercer vos compétences en matière de formatage.

```
sej_hosp <- read_csv(here("data/sejours_hospitaliers.csv"))
```

```
sej_hosp
```

```
## # A tibble: 5 × 6
##   num_patient date_adm_defaut date_adm_courant
##       <dbl> <date>           <chr>
## 1 1 2021-05-23 05/23/2021
## 2 2 2022-12-07 12/07/2022
## 3 3 2022-03-27 03/27/2022
## 4 4 2022-04-28 04/28/2022
## 5 5 2023-06-28 06/28/2023
## #> #>   date_adm_abrege date_adm_mixte date_sortie_defaut
## #>   <chr>           <chr>           <date>
## #> 1 23 mai 2021    2021/05/23 2021-06-27
## #> 2 07 déc. 2022   12-07-2022 2023-02-19
## #> 3 27 mars 2022  2022/03/27 2022-05-31
## #> 4 28 avr. 2022  28.04.2022 2022-07-04
## #> 5 28 juin 2023  06-28-2023 2023-07-31
```

Introduction aux Dates en R

Dans R, il existe une classe spécifique conçue pour gérer les dates, appelée `Date`. Le format par défaut pour cette classe est "aaaa-mm-jj". Par exemple, le 31 décembre 2000 serait représenté comme `2000-12-31`.

Cependant, si vous entrez simplement une telle chaîne de caractères de date, R va initialement la considérer comme un caractère :

```
class("2000-12-31")
```

```
## [1] "character"
```

Si nous voulons créer une `Date`, nous pouvons utiliser la fonction `as.Date()` et écrire la date en suivant le format par défaut :

```
my_date <- as.Date("2000-12-31")
class(my_date)
```

```
## [1] "Date"
```

WATCH OUT



Notez le "D" majuscule dans la fonction `as.Date()` !

Maintenant que vos objets sont de la classe `Date`, vous pouvez maintenant faire des calculs simples comme trouver la différence entre deux dates :

```
as.Date("2000-12-31") - as.Date("2000-12-20")
```

```
## Time difference of 11 days
```

Ceci ne serait bien sûr pas possible si vous aviez de simples caractères :

```
"2000-12-31" - "2000-12-20"
```

```
Error in "2000-12-31" - "2000-12-20" :
non-numeric argument to binary operator
```

De nombreuses autres opérations s'appliquent uniquement à la classe Date. Nous les explorerons en détail plus tard.

Le format par défaut pour `as.Date()` est “aaaa-mm-jj”. D'autres formats courants comme “mm/jj/aaaa” ou “jj mois, aaaa” ne fonctionneront pas par défaut :

```
as.Date("12/31/2000")
as.Date("31 dec, 2000")
```

Cependant, R acceptera également “/” au lieu de “-” tant que l'ordre est toujours “aaaa/mm/jj”. Les dates s'afficheront au format par défaut “aaaa-mm-jj” :

```
as.Date("2000/12/31")
```

```
## [1] "2000-12-31"
```

En résumé, les seuls formats qui fonctionnent par défaut sont “aaaa-mm-jj” et “aaaa/mm/jj”. Plus tard dans cette leçon, nous allons apprendre à gérer différents formats de date et on vous donnerons des conseils sur la coercion de dates importées sous forme de chaînes de caractères dans la classe Date. Pour l'instant, l'essentiel est de comprendre que les dates ont leur propre classe avec ses propres propriétés de formatage.

SIDE NOTE



Il existe une autre classe de données utilisée pour les dates, appelée `POSIXct`. Cette classe gère les dates et heures ensemble, et le format par défaut est “aaaa-mm-jj hh:mm:ss”. Cependant, dans le cadre de ce cours, nous ne travaillerons pas avec cette classe car ce niveau d'analyse est beaucoup moins courant dans le domaine de la santé publique.

Conversion de chaînes de caractères en dates

Revenons à nos données PID et regardons comment R a classé nos variables de date !

```
pid %>%
  select(contains("date"))
```

```
## # A tibble: 112 × 5
##   date_debut_defaut date_fin_defaut date_debut_typique
##   <date>            <date>          <chr>
## 1 2014-04-07        2014-04-17      07/04/2014
```

```

## 2 2014-04-22      2014-04-27      22/04/2014
## 3 2014-05-13      2014-05-13      13/05/2014
## 4 2014-05-13      2014-05-13      13/05/2014
## 5 2014-05-13      2014-05-13      13/05/2014
## 6 2014-05-15      2014-05-26      15/05/2014
## 7 2014-05-27      2014-05-27      27/05/2014
## 8 2014-05-27      2014-05-27      27/05/2014
## 9 2014-05-28      2014-06-16      28/05/2014
## 10 2014-06-18     2014-06-27     18/06/2014
##   date_debut_longue date_debut_mixte
##   <chr>           <chr>
## 1 07 avril 2014    07-04-2014
## 2 22 avril 2014    22-04-2014
## 3 13 mai 2014     13-05-2014
## 4 13 mai 2014     13 mai 2014
## 5 13 mai 2014     13-05-2014
## 6 15 mai 2014     15 mai 2014
## 7 27 mai 2014     27 mai 2014
## 8 27 mai 2014     27 mai 2014
## 9 28 mai 2014     28-05-2014
## 10 18 juin 2014    2014/06/18
## # i 102 more rows

```

Comme nous pouvons le voir, les deux colonnes reconnues comme des dates sont `date_debut_defaut` et `date_fin_defaut`, qui suivent le format “aaaa-mm-jj” de R :

	date_debut_defaut	date_fin_defaut	date_debut_typique
1	👉<date>👉	👉<date>👉	<chr>
2	2014-04-07	2014-04-17	07/04/2014
	2014-04-22	2014-04-27	22/04/2014

Toutes les autres colonnes de date dans notre jeu de données ont été importées comme des chaînes de caractères (“chr”), et si nous voulons les transformer en dates, nous devons indiquer à R qu’elles sont des dates, ainsi que spécifier l’ordre des composants de la date.

Vous vous demandez peut-être pourquoi il est nécessaire de spécifier l’ordre. Eh bien, imaginez qu’on ait une date écrite 01-02-03. Est-ce le 2 janvier 2003 ? Le 1er février 2003 ? Ou peut-être le 2 mars 2001 ? Il existe tellement de conventions différentes pour écrire les dates que si R devait deviner le format, il y aurait inévitablement des cas où il se tromperait.

Pour résoudre ce problème, il existe deux façons principales de convertir des chaînes en dates qui impliquent de spécifier l’ordre des composants. La première approche s’appuie sur les fonctions natives de R (appelé couramment par son nom anglais, “Base R”), et la seconde utilise un package appelé `lubridate` de la bibliothèque `tidyverse`. Regardons d’abord la fonction native de R !

Avec les fonctions natives de R (Base R)

Dans l'introduction nous avons vu comment convertir des chaînes de caractères en dates avec les fonctions natives de R, plus précisément avec la fonction `as.Date()`. Essayons de l'appliquer à notre colonne `date_debut_typique` sans spécifier l'ordre des composants pour voir ce qui se passe.

```
pid %>%
  mutate(date_debut_typique = as.Date(date_debut_typique)) %>%
  select(date_debut_typique)
```

```
## # A tibble: 5 × 1
##   date_debut_typique
##   <date>
## 1 2007-04-20
## 2 2022-04-20
## 3 2013-05-20
## 4 2013-05-20
## 5 2013-05-20
```

Évidemment, ce n'est pas du tout ce que nous voulions ! Si nous regardons la variable d'origine, nous pouvons voir qu'elle est formatée "jj/mm/aaaa". R a essayé d'appliquer son format par défaut à ces dates, ce qui donne ces résultats étranges.

WATCH OUT



Souvent, R vous retournera un message d'erreur si vous essayez de convertir des caractères ambiguës en dates sans spécifier l'ordre de leurs composants. Mais, comme nous venons de le voir, ce n'est pas toujours le cas ! Vérifiez toujours que votre code s'est exécuté comme prévu et ne vous fiez jamais seulement aux messages d'erreur pour vous assurer que vos transformations de données ont fonctionné correctement.

Pour que R interprète correctement nos dates, nous devons utiliser l'option `format` et spécifier les composants de notre date à l'aide d'une série de symboles. Le tableau ci-dessous montre les symboles pour les composants de format les plus courants :

Composant	Symbol	Exemple
Année, en format long (4 chiffres)	%Y	2023
Année, format abrégé (2 chiffres)	%y	23
Mois, en format numérique (1-12)	%m	01
Mois écrit en format long	%B	janvier
Mois écrit en format abrégé	%b	janv
Jour du mois	%d	31
Jour de la semaine, en format numérique (1-7 en		

Composant	Symbol	Exemple
commençant par dimanche)	%u	5
Jour de la semaine écrit en format long	%A	vendredi
Jour de la semaine écrit en format abrégé	%a	ven

**Add note about systems computer language

Si on revient à notre variable d'origine `date_debut_typique`, on voit qu'elle est formatée "jj/mm/aaaa", ce qui correspond au jour du mois, suivi du mois représenté par un nombre (01-12), puis de l'année en format long (4 chiffres). Si nous utilisons ces symboles, nous devrions obtenir les résultats que nous cherchons.

```
pid %>%
  mutate(date_debut_typique = as.Date(date_debut_typique, format="%d%m%Y"))
```

```
## # A tibble: 5 × 9
##   village      cible_PID reelle_PID couverture_p
##   <chr>        <dbl>     <dbl>       <dbl>
## 1 Mess           87        64         73.6
## 2 Nkombedzi     183       169        92.4
## 3 B Compound    16         16         100
## 4 D Compound    3          2          66.7
## 5 Post Office   6          3          50
##   date_debut_defaut date_fin_defaut date_debut_typique
##   <date>            <date>            <date>
## 1 2014-04-07      2014-04-17      NA
## 2 2014-04-22      2014-04-27      NA
## 3 2014-05-13      2014-05-13      NA
## 4 2014-05-13      2014-05-13      NA
## 5 2014-05-13      2014-05-13      NA
## # i 2 more variables: date_debut_longue <chr>,
## #   date_debut_mixte <chr>
```

Bon, ce n'est toujours pas ce que nous voulions. Avez-vous une idée de la raison pour laquelle ça n'a pas marché ? C'est parce que les composants de nos dates sont séparés par un slash "/", que nous devons inclure dans notre option de format. Essayons à nouveau !

```
pid %>%
  mutate(date_debut_typique = as.Date(date_debut_typique,
                                         format="%d/%m/%Y")) %>%
  select(date_debut_typique)
```

```
## # A tibble: 5 × 1
##   date_debut_typique
##   <date>
## 1 2014-04-07
## 2 2014-04-22
## 3 2014-05-13
```

```
## 4 2014-05-13  
## 5 2014-05-13
```

Cette fois ça a parfaitement fonctionné ! Maintenant nous savons comment convertir des chaînes de caractères en dates en utilisant la fonction native de R `as.Date()` avec l'option `format`.

Convertir date longue

Essayez de convertir la colonne `date_debut_longue` des données PID en classe `Date`. N'oubliez pas d'inclure tous les éléments dans l'option de format, y compris les symboles qui séparent les composants de la date !

Trouver les erreurs de code

Est-ce que vous arrivez à trouver toutes les erreurs dans le code suivant ?

```
as.Date("26 juin, 1987", format = "%d%b%y")
```

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

Avec lubridate

Le package `lubridate` nous donne une façon beaucoup plus simple de convertir des chaînes de caractères en dates que les fonctions natives de R. Avec ce package, il suffit de spécifier l'ordre dans lequel apparaissent l'année, le mois, et le jour en utilisant respectivement les lettres "y" pour l'année, "m" pour le mois et "d" pour le jour (correspondant à "year", "month" et "day" en anglais). Avec ces fonctions, ce n'est pas nécessaire de spécifier les caractères qui séparent les différents composants de la date.

Regardons quelques exemples :

```
mdy("04/30/2002")
```

```
## [1] "2002-04-30"
```

```
dmy("30 avril 2002")
```

```
## Warning: All formats failed to parse. No formats found.
```

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

```
ymd("2002-04-03")
```

```
## [1] "2002-04-03"
```

Facile ! Et comme nous pouvons le voir, nos dates sont affichées en utilisant le format R par défaut. Maintenant que nous arrivons à utiliser les fonctions du package lubridate, essayons de les appliquer à la variable `date_debut_longue` de notre jeu de données.

```
pid %>%
  mutate(date_debut_longue = mdy(date_debut_longue)) %>%
  select(date_debut_longue)
```

```
## Warning: There was 1 warning in `mutate()` .
## i In argument: `date_debut_longue =
##   mdy(date_debut_longue)` .
## Caused by warning:
## ! 77 failed to parse.
```

```
## # A tibble: 5 × 1
##   date_debut_longue
##   <date>
## 1 2014-07-20
## 2 NA
## 3 NA
## 4 NA
## 5 NA
```

Parfait, c'est exactement ce qu'on voulait !

Convertir date typique Essayez de convertir la colonne `date_debut_typique` du jeu de données PID en classe `Date` en utilisant les fonctions du package `lubridate`.

Formatage de base et lubridate

Le tableau suivant contient les formats trouvés dans les colonnes `date_adm_abrege` et `date_adm_mixte` de notre jeu de données de patients hospitalisés. Est ce que vous arrivez à remplir les cellules vides ?

Exemple	Base R	Lubridate
07 déc 2022		
03-27-2022		mdy
28.04.2022		%Y/%m/%d

Maintenant que nous connaissons deux façons de convertir des chaînes de caractères en classe `Date` en spécifiant l'ordre des composants ! Mais que faire si nous avons plusieurs formats de date dans la même colonne ? Passons à la section suivante pour le découvrir !

Gérer des dates mixtes avec `lubridate::parse_date_time()`

Lorsque l'on travaille avec des dates, il arrive parfois d'avoir différents formats au sein de la même colonne. Heureusement, lubridate dispose d'une fonction pratique à cet effet ! La fonction `parse_date_time()` est similaire aux fonctions que nous avons vues précédemment dans le package lubridate, mais avec plus de flexibilité et la possibilité d'inclure plusieurs formats de date dans le même appel en utilisant l'argument `orders`. Jetons un rapide coup d'œil à son fonctionnement avec quelques exemples simples.

Pour comprendre comment utiliser `parse_date_time()`, appliquons-le à une seule chaîne de caractères que nous voulons convertir en date.

```
parse_date_time("30/07/2001", orders="dmy")
```

```
## [1] "2001-07-30 UTC"
```

C'est parfait ! Utiliser la fonction de cette façon est équivalent à utiliser la fonction `dmy()`. Cependant, la vraie puissance de `parse_date_time()` se révèle lorsque nous avons plusieurs dates avec des formats différents.

SIDE NOTE



La partie "UTC" est le fuseau horaire par défaut utilisé pour analyser la date. Celui-ci peut être modifié avec l'argument `tz=`, mais changer le fuseau horaire par défaut est rarement nécessaire lorsqu'on traite uniquement de dates, contrairement à des dates-heures.

Regardons un autre exemple avec deux formats différents :

```
parse_date_time(c("1 janv 2000", "07/30/2001"), orders=c("dmy", "mdy"))
```

```
## Warning: 1 failed to parse.
```

```
## [1] NA                 "2001-07-30 UTC"
```

Notez que cet exemple spécifique fonctionnera toujours si vous changez l'ordre dans lequel vous présentez les formats :

```
parse_date_time(c("1 janv 2000", "07/30/2001"), orders=c("mdy", "dmy"))
```

```
## [1] NA           "2001-07-30 UTC"
```

Le dernier bloc de code fonctionne toujours car `parse_date_time()` vérifie chaque format spécifié dans l'argument `orders` jusqu'à trouver une correspondance. Cela signifie que, que vous listiez "dmy" en premier ou "mdy" en premier, il essaiera les deux formats sur chaque chaîne de date pour voir lequel convient. L'ordre n'a pas d'importance pour des chaînes de dates distinctes qui ne peuvent correspondre qu'à un seul format.

Cependant, lorsque l'on traite des dates ambiguës comme "01/02/2000" ou "01/03/2000", qui pourraient être interprétées soit comme le 2 janvier et le 3 janvier, soit comme le 1er février et le 1er mars respectivement, l'ordre dans `orders` a vraiment de l'importance :

```
parse_date_time(c("01/02/2000", "01/03/2000"), orders=c("mdy", "dmy"))
```

```
## [1] "2000-01-02 UTC" "2000-01-03 UTC"
```

Dans l'exemple ci-dessus, parce que "mdy" est listé en premier, la fonction interprète les dates comme étant le 2 janvier et le 3 janvier. Mais, si vous changez l'ordre et listiez "dmy" en premier, elle interpréterait les dates comme étant le 1er février et le 1er mars :

```
parse_date_time(c("01/02/2000", "01/03/2000"), orders=c("dmy", "mdy"))
```

```
## [1] "2000-02-01 UTC" "2000-03-01 UTC"
```

Par conséquent, lorsqu'il y a une ambiguïté potentielle dans les chaînes de dates, l'ordre dans lequel vous spécifiez les formats devient très important.

Utilisation de `parse_date_time`

Les dates dans le code ci-dessous sont le 9 novembre 2002, le 4 décembre 2001 et le 5 juin 2003. Complétez le code pour les convertir de caractères en dates.

```
parse_date_time(c("11/09/2002", "12/04/2001", "2003-06-05"), orders=c(...))
```

Revenons à notre jeu de données, cette fois sur la colonne `date_debut_mixte`.

```
pid %>%  
  select("date_debut_mixte")
```

```
## # A tibble: 5 × 1  
##   date_debut_mixte  
##   <chr>  
## 1 07-04-2014  
## 2 22-04-2014
```

```
## 3 13-05-2014
## 4 13 mai 2014
## 5 13-05-2014
```

Étant donné que cette colonne a été créée spécifiquement pour ce cours, nous connaissons les différents formats de date qu'elle contient. Dans votre propre travail, assurez-vous toujours de connaître le format de vos dates, car nous savons que certaines peuvent être ambiguës.

Ici, nous travaillons avec quatre formats différents, plus précisément :

- aaaa/mm/jj
- jj mois aaaa
- jj-mm-aaaa
- mm/jj/aaaa

Voyons à quoi cela ressemble dans lubridate par rapport au R de base :

Example date	Base R	Lubridate
2014/05/13	%Y/%m/%d	ymd
13 mai 2014	%B%d%Y	dmy
27-05-2014	%d-%m-%Y	dmy
07/21/14	%m/%d/%y	mdy

Ici, lubridate considère qu'il n'y a que trois formats différents ("ymd", "mdy" et "dmy"). Maintenant que nous savons comment nos données sont formatées, nous pouvons utiliser la fonction `parse_date_time()` pour les changer en classe `Date`.

```
pid %>%
  select(date_debut_mixte) %>%
  mutate(date_debut_mixte = parse_date_time(date_debut_mixte, orders =
    c("mdy", "ymd", "dmy")))
```

```
## Warning: There was 1 warning in `mutate()` .
## i In argument: `date_debut_mixte =
##   parse_date_time(date_debut_mixte, orders = c("mdy",
##     "ymd", "dmy"))` .
## Caused by warning:
## ! 25 failed to parse.
```

date_debut_mixte
2014-04-07
2014-04-22
2014-05-13

date_debut_mixte

2014-05-13

C'est beaucoup mieux ! R a correctement formaté notre colonne et elle est désormais reconnue comme une variable de type date. Vous vous demandez peut-être si l'ordre des formats est nécessaire dans ce cas. Essayons un ordre différent pour le découvrir !

```
pid %>%
  select(date_debut_mixte) %>%
  mutate(date_debut_mixte = parse_date_time(date_debut_mixte, orders =
    c("dmy", "mdy", "ymd")))
```

```
## Warning: There was 1 warning in `mutate()` .
## i In argument: `date_debut_mixte =
##   parse_date_time(date_debut_mixte, orders = c("dmy",
##     "mdy", "ymd"))` .
## Caused by warning:
## ! 25 failed to parse.
```

date_debut_mixte

2014-04-07

2014-04-22

2014-05-13

NA

2014-05-13

Cela ne semble pas avoir fait de différence, les dates sont toujours formatées correctement ! Si vous vous demandez pourquoi l'ordre importait dans notre exemple précédent mais pas ici, c'est lié au fonctionnement de la fonction `parse_date_time()`. Lorsqu'elle reçoit plusieurs ordres, la fonction tente de trouver la meilleure correspondance pour un sous-ensemble d'observations en considérant les séparateurs de dates et en favorisant l'ordre dans lequel les formats ont été fournis. Dans notre dernier exemple, les deux dates étaient séparées par un "/" et les deux formats fournis ("dmy" et "mdy") étaient des formats possibles, la fonction a donc favorisé le premier donné.

```
parse_date_time(c("01/02/2000", "01/03/2000"), orders=c("mdy", "dmy"))
```

```
## [1] "2000-01-02 UTC" "2000-01-03 UTC"
```

```
parse_date_time(c("01/02/2000", "01/03/2000"), orders=c("dmy", "mdy"))
```

```
## [1] "2000-02-01 UTC" "2000-03-01 UTC"
```

Dans nos données PID, nous avions aussi des formats qui pouvaient être ambigus comme jj-mm-aaaa et mm/jj/aaaa. Mais ici, la fonction peut utiliser les séparateurs comme indice pour trouver des règles de formatage et distinguer les différents formats. Par exemple, si nous avons une date ambiguë comme 01-02-2000, mais aussi une date avec le même séparateur qui n'est pas ambiguë comme 30-05-2000, la fonction déterminera que la réponse la plus probable est que toutes les dates séparées par un “-” sont au format jj-mm-aaaa, et appliquera cette règle de manière récursive aux données d'entrée. Si vous voulez en savoir plus sur les détails de la fonction `parse_date_time()`, cliquez ici ou exécutez `?parse_date_time` dans R !

Utilisation de `parse_date_time` avec `adm_date_messy` A l'aide du tableau que vous avez rempli pour l'exercice de la [Section 6.2 Lubridate](#), utilisez la fonction `parse_date_time()` pour changer la classe de la colonne `date_adm_mixte` du jeu de données de patients hospitalisés en Date, ip!

Modifier l’Affichage des Dates

Jusqu'à présent, nous avons converti des chaînes de caractères de divers formats en classe `Date` qui suit un format par défaut “aaaa-mm-jj”. Mais que faire si nous voulons que nos dates s'affichent dans un format spécifique qui est différent de ce format par défaut, comme lorsque nous créons des rapports ou des graphiques ? Cela est rendu possible en reconvertissant les dates en chaînes de caractères en utilisant la fonction `format()` !

La fonction `format()` vous offre une grande flexibilité pour personnaliser l'apparence de vos dates selon vos préférences. Vous pouvez accomplir cela en utilisant les mêmes symboles que nous avons vu avec la fonction `as.Date()`, en les ordonnant pour correspondre à l'apparence souhaitée de votre date. Revenons au tableau pour rafraîchir notre mémoire sur la façon dont les différentes parties d'une date sont représentées dans R.

Composant	Symbol	Exemple
Année, en format long (4 chiffres)	%Y	2023
Année, format abrégé (2 chiffres)	%y	23
Mois, en format numérique (1-12)	%m	01
Mois écrit en format long	%B	janvier
Mois écrit en format abrégé	%b	janv
Jour du mois	%d	31
Jour de la semaine, en format numérique (1-7 en commençant par dimanche)	%u	6
Jour de la semaine écrit en format long	%A	vendredi
Jour de la semaine écrit en format abrégé	%a	ven

Très bien, essayons maintenant d'appliquer cette fonction à une seule date. Disons que nous voulons que la date 2000-01-31 s'affiche comme "31 janv. 2000".

```
my_date <- as.Date("2000-01-31")
format(my_date, "%d %b. %Y")
```

```
## [1] "31 Jan. 2000"
```

Créer un vecteur de dates Créez un vecteur de dates contenant la date du 7 mai 2018. Formatez ensuite la date en jj/mm/aaaa en tant que caractère.

Maintenant, essayons de l'utiliser sur nos données PID. Créons une nouvelle variable appelée `date_debut_char` à partir de la colonne `date_debut_defaut`. Nous allons la formater pour qu'elle s'affiche comme jj-mm-aaaa.

```
pid %>%
  mutate(date_debut_char = format(date_debut_defaut, "%d-%m-%Y")) %>%
  select(date_debut_defaut)
```

```
## # A tibble: 5 × 1
##   date_debut_defaut
##   <date>
## 1 2014-04-07
## 2 2014-04-22
## 3 2014-05-13
## 4 2014-05-13
## 5 2014-05-13
```

Super ! Faisons un dernier exemple en utilisant notre variable `date_fin_defaut` et en la formatant comme jj mois aaaa.

```
pid %>%
  mutate(end_date_char = format(date_fin_defaut, "%d %B %Y")) %>%
  select(date_fin_defaut)
```

```
## # A tibble: 5 × 1
##   date_fin_defaut
##   <date>
## 1 2014-04-17
## 2 2014-04-27
## 3 2014-05-13
## 4 2014-05-13
## 5 2014-05-13
```

Génial !

EN RÉSUMÉ

Félicitations pour avoir terminé la première leçon sur les dates ! Maintenant que vous comprenez comment les dates sont stockées, affichées et formatées dans R, vous pouvez passer à la section suivante où vous apprendrez à effectuer des manipulations avec les dates et à créer des graphiques de séries temporelles de base.

Answer Key

Convertir une date longue

```
irs <- irs %>%
  mutate(start_date_long = as.Date(start_date_long, format="%B, %d %Y"))
```

Trouver les erreurs de code

```
as.Date("June 26, 1987", format = "%B %d, %Y")
```

Convertir une date typique

```
irs %>%
  mutate(start_date_typical = dmy(start_date_typical))
```

Formatage de base et lubridate

Date example	Base R	Lubridate
07 dec, 2022	%b %d, %Y	dmy
03-27-2022	%m-%d-%Y	mdy
28.04.2022	%d.%m.%Y	dmy
2021/05/23	%Y/%m/%d	ymd

Utilisation de parse_date_time

```
parse_date_time(c("11/09/2002", "12/04/2001", "2003-06-05"), orders=c("mdy",
  "ymd"))
```

Utilisation de parse_date_time avec adm_date_messy

```
ip %>%
  mutate(adm_date_messy = parse_date_time(adm_date_messy, orders = c("mdy",
    "dmy", "ymd")))
```

Créer un vecteur de dates

```
my_date <- as.Date("2018-05-07")
format(my_date, "%m/%d/%Y")
```

Contributors

The following team members contributed to this lesson:

(make sure to update the contributor list accordingly!)

Dates 2 : Manipuler les dates dans R

GRAPH Network & WHO, supported by the Global Fund to fight HIV, TB & Malaria

October 2023

This document is a draft of a lesson made by the GRAPH Network, a non-profit headquartered at the University of Geneva Global Health Institute, in collaboration with the World Health Organization, under a Global Fund 2023 grant to create e-learning modules to build in-country data capacity for epidemiological and impact analysis for National HIV, TB and malaria programs

Introduction
Objectifs d'apprentissage
Packages
Données
Calcul des intervalles de date
Utilisation de l'opérateur “-”
Utilisation de l'opérateur d'intervalle du package lubridate
Comparaison
Extraction des composants de la date
Arrondir
WRAP UP!
Answer Key

Introduction

Vous avez maintenant une bonne compréhension de la façon dont les dates sont stockées, affichées et formatées dans R. Dans cette leçon, vous apprendrez à effectuer des analyses simples avec les dates, telles que le calcul de l'intervalle de temps entre les intervalles de dates et la création de graphiques de séries chronologiques ! Ces compétences sont cruciales pour toute personne travaillant avec des données de santé publique, car elles sont la base de la compréhension des tendances temporelles telles que la propagation des maladies, les changements dans les indicateurs de santé au niveau de la population et l'impact des mesures préventives !

Objectifs d'apprentissage

- Vous savez comment calculer les intervalles entre les dates
- Vous savez comment extraire des composants des colonnes de date
- Vous savez comment arrondir les dates
- Vous êtes capable de créer des graphiques de séries chronologiques simples

Packages

Veuillez charger les packages nécessaires pour cette leçon avec le code ci-dessous :

```
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse,
  here,
  lubridate,
  ggplot2)
```

Données

Les premières données avec lesquelles nous travaillerons sont les données IRS de la leçon précédente. Consultez la première leçon sur les dates pour plus d'informations sur le contenu de ces données de pulvérisation intradomiciliaire d'insecticide (IRS).

```
données_pulvérisation <- read_csv(here("data/Iollovo_data.csv"))
```

```
données_pulvérisation
```

```
## # A tibble: 5 × 9
##   village      target_spray sprayed coverage_p
##   <chr>        <dbl>    <dbl>     <dbl>
## 1 Mess          87       64      73.6
## 2 Nkombedzi    183      169      92.4
## 3 B Compound    16       16      100
## 4 D Compound     3        2      66.7
## 5 Post Office    6        3      50
##   start_date_default end_date_default start_date_typical
##   <date>            <date>           <chr>
## 1 2014-04-07      2014-04-17      07/04/2014
## 2 2014-04-22      2014-04-27      22/04/2014
## 3 2014-05-13      2014-05-13      13/05/2014
## 4 2014-05-13      2014-05-13      13/05/2014
## 5 2014-05-13      2014-05-13      13/05/2014
## # i 2 more variables: start_date_long <chr>,
## #   start_date_messy <chr>
```

Le deuxième ensemble de données que nous utiliserons contient des données de la même étude à partir de laquelle les données IRS ont été extraites. Ici, nous avons le taux d'incidence mensuel moyen du paludisme pour 1000 personnes de 2015 à 2019 pour les villages ayant reçu l'IRS (cas) et les villages n'ayant pas reçu l'IRS (témoins). Le jeu de données contient également la température minimale moyenne et la température maximale moyenne à Illovo pour chaque mois de 2015 à 2019.

```
données_météo_paludisme <- read_csv(here("data/Iollovo_ir_weather.csv"))
```

```
données_météo_paludisme
```

```

## # A tibble: 5 × 5
##   date      ir_case ir_control avg_min avg_max
##   <date>     <dbl>     <dbl>    <dbl>    <dbl>
## 1 2015-01-10     42.9     19.6    21.2    31.6
## 2 2015-02-03     61.0     10.1    21.5    32.9
## 3 2015-03-11     74.1     56.8    20.6    33.4
## 4 2015-04-15     95.2     34.7    18.5    32.3
## 5 2015-05-05     89.8     31.9    15.9    31.4

```

Les variables incluses dans ces données sont les suivantes :

- `date` : Points temporels mensuels allant de 2015 à 2019, avec le jour du mois générée de manière aléatoire
- `cas_IRS` : Taux d'incidence moyen du paludisme pour 1000 personnes pour les villages ayant reçu l'IRS
- `témoins` : Taux d'incidence moyen du paludisme pour 1000 personnes pour les villages n'ayant pas reçu l'IRS
- `moyenne_min` : Température minimale moyenne mensuelle en degrés Celsius
- `moyenne_max` : Température maximale moyenne mensuelle en degrés Celsius

MÉTÉO

```
météo <- read_csv(here("data/Illovo_weather.csv"))
```

```
météo
```

```

## # A tibble: 5 × 4
##   date      min_temp max_temp rain
##   <date>     <dbl>    <dbl>  <dbl>
## 1 2015-01-01     21.5    29.9  21.7
## 2 2015-01-02     19.6    30.4  2.2
## 3 2015-01-03     21.6    29.9  25.8
## 4 2015-01-04     20.0    29.5  1
## 5 2015-01-05     20.0    32.2  53

```

Calcul des intervalles de date

Pour commencer, nous allons examiner deux façons de calculer les intervalles, la première en utilisant l'opérateur “-” en R de base, et la seconde en utilisant l'opérateur d'intervalle du package `lubridate`. Jetons un coup d’œil à ces deux méthodes et comparons-les.

Utilisation de l'opérateur “-”

La première façon de calculer les différences de temps consiste à utiliser l'opérateur “-” pour soustraire une date d'une autre. Créons deux variables de date et essayons !

```
date_1 <- as.Date("2000-01-01") # 1er janvier 2000  
date_2 <- as.Date("2000-01-31") # 31 janvier 2000  
date_2 - date_1
```

```
## Time difference of 30 days
```

C'est aussi simple que ça ! Ici, nous pouvons voir que R renvoie la différence de temps en jours.

Utilisation de l'opérateur d'intervalle du package lubridate

La deuxième façon de calculer des intervalles de temps consiste à utiliser l'opérateur %--% du package `lubridate`. Nous pouvons voir ici que la sortie est légèrement différente de la sortie de R de base.

```
date_1 %--% date_2
```

```
## [1] 2000-01-01 UTC--2000-01-31 UTC
```

Notre sortie est un intervalle entre deux dates. Si nous voulons savoir combien de jours se sont écoulés, nous devons utiliser la fonction `days()`. Le (1) ici indique à `lubridate` de compter par incrément de 1 jour à la fois.

```
date_1 %--% date_2/days(1)
```

```
## [1] 30
```

Techniquement, spécifier `days(1)` n'est pas vraiment nécessaire, nous pouvons également laisser les parenthèses vides (c'est-à-dire `days()`) et obtenir le même résultat, car la valeur par défaut de `lubridate` est de compter par incrément de 1. Cependant, si nous voulons compter par incrément de 5 jours par exemple, nous pouvons spécifier `days(5)` et le résultat retourné sera de 6, car $5 \times 6 = 30$.

```
date_1 %--% date_2/days(5)
```

```
## [1] 6
```

Utilisez les trois méthodes différentes pour calculer le temps entre les dates ci-dessous. Utilisez la fonction `weeks()` à la place de `days()` dans la méthode lubridate pour obtenir la réponse en semaines.

```
oct_31 <- as.Date("2023-10-31")
jul_20 <- as.Date("2023-07-20")
```

Comparaison

Alors, quelle est la meilleure méthode à utiliser ? Eh bien, vous pouvez probablement voir que lubridate offre plus de flexibilité, et grâce à la manière dont il gère les dates (les détails étant hors du champ de cette formation), il est plus précis !

Jetons un coup d'œil à un exemple de cela en calculant un intervalle de 6 ans. Avec R de base, les résultats sont donnés en jours. Si nous voulons obtenir le résultat en années, nous devons utiliser la fonction `as.numeric()` pour transformer les résultats en un nombre sans l'unité "jours" et diviser par 365,25. Nous utilisons 365,25 parce qu'il y a un jour supplémentaire tous les 4 ans (c'est-à-dire les années bissextiles), donc en moyenne, il y a 365,25 jours par an. Avec lubridate, nous pouvons simplement spécifier `years()`. Voyons comment chaque méthode se comporte dans l'exemple suivant.

```
date_1 <- as.Date("2000-01-01") # 1er janvier 2000
date_2 <- as.Date("2006-01-01") # 1er janvier 2006
as.numeric(date_2-date_1)/365.25
```

```
## [1] 6.001369
```

```
date_1 %--% date_2/years()
```

```
## [1] 6
```

D'accord, ils sont très similaires, mais pas exactement les mêmes ! Évidemment, la réponse correcte est que 6 ans se sont écoulés entre les deux dates données. Cependant, les dates peuvent devenir compliquées, car le nombre de jours dans une année, voire dans un mois, n'est pas toujours le même. Essayer de calculer des intervalles de dates avec R de base est une approximation plutôt qu'une réponse exacte. Dans l'exemple ci-dessus, nous n'avons pas pu obtenir exactement 6 ans en utilisant l'opérateur moins en R de base, même si nous divisons par 365, 365,25 ou 366.

```
as.numeric(date_2-date_1)/365
```

```
## [1] 6.005479
```

```
as.numeric(date_2-date_1)/365.25
```

```
## [1] 6.001369  
  
as.numeric(date_2-date_1) / 366
```

```
## [1] 5.989071
```

Lubridate, en revanche, peut gérer ces complexités, il donnera donc une réponse exacte. Les différences sont légères, mais en termes de flexibilité et de précision, lubridate est le grand gagnant !

Bien que nous ne couvrions pas les date-heures dans ce cours, il est bon de savoir que lubridate est particulièrement utile pour traiter les fuseaux horaires et les changements d'heure d'été.

Pouvez-vous appliquer cela à notre jeu de données de pulvérisation ? Créez une nouvelle variable appelée `durée_pulvérisation` et utilisez l'opérateur `%--%` de lubridate pour calculer le nombre de jours entre `start_date_default` et `end_date_default`.

Extraction des composants de la date

Parfois, lors de votre nettoyage ou de votre analyse de données, vous devrez peut-être extraire un composant spécifique de votre variable de date. Un ensemble de fonctions utiles dans le package `lubridate` vous permet de le faire exactement. Par exemple, si nous voulions créer une colonne avec seulement le mois où la pulvérisation a commencé à chaque intervalle, nous pourrions utiliser la fonction `month()` de la manière suivante :

```
données_pulvérisation %>%  
  mutate(mois_début = month(start_date_default)) %>%  
  select(village, start_date_default, mois_début)
```

```
## # A tibble: 5 × 3  
##   village      start_date_default mois_début  
##   <chr>        <date>                <dbl>  
## 1 Mess        2014-04-07              4  
## 2 Nkombedzi  2014-04-22              4  
## 3 B Compound 2014-05-13              5  
## 4 D Compound  2014-05-13              5  
## 5 Post Office 2014-05-13              5
```

Comme nous pouvons le voir ici, cette fonction renvoie le mois sous forme de numéro de 1 à 12. Pour notre première observation, la pulvérisation a commencé au cours du quatrième mois, donc en avril. C'est aussi simple que ça ! Si nous voulons que R affiche le

mois écrit plutôt que le numéro en dessous, nous pouvons utiliser l'argument `label=TRUE`.

```
données_pulvérisation %>%
  mutate(mois_début = month(start_date_default, label=TRUE)) %>%
  select(village, start_date_default, mois_début)
```

```
## # A tibble: 5 × 3
##   village      start_date_default mois_début
##   <chr>        <date>            <ord>
## 1 Mess         2014-04-07       Apr
## 2 Nkombedzi    2014-04-22       Apr
## 3 B Compound   2014-05-13       May
## 4 D Compound   2014-05-13       May
## 5 Post Office  2014-05-13       May
```

De même, si nous voulions extraire l'année, nous utiliserions la fonction `year()`.

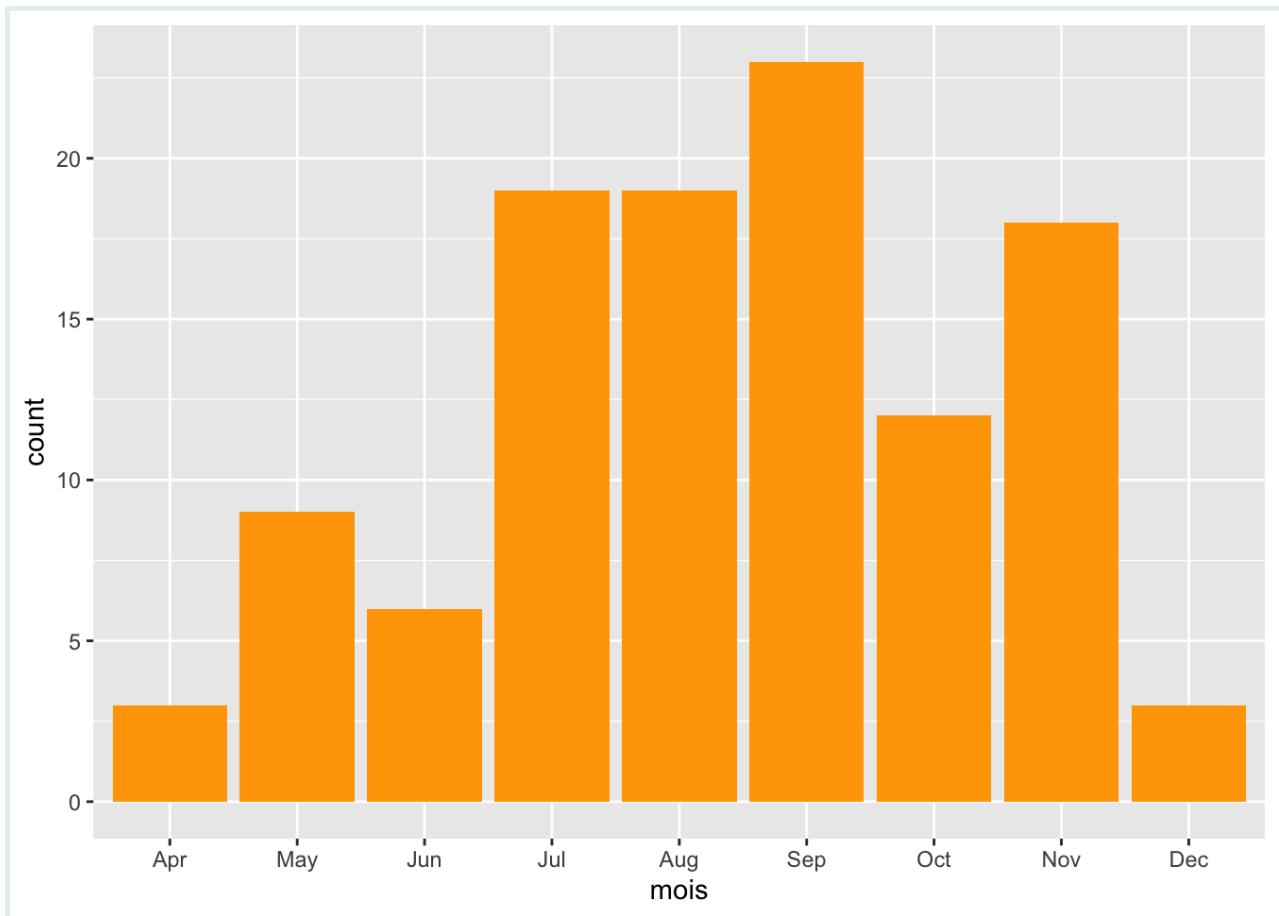
```
données_pulvérisation %>%
  mutate(année_début = year(start_date_default)) %>%
  select(village, start_date_default, année_début)
```

```
## # A tibble: 5 × 3
##   village      start_date_default année_début
##   <chr>        <date>            <dbl>
## 1 Mess         2014-04-07       2014
## 2 Nkombedzi    2014-04-22       2014
## 3 B Compound   2014-05-13       2014
## 4 D Compound   2014-05-13       2014
## 5 Post Office  2014-05-13       2014
```

Créez une nouvelle variable appelée `jour_semaine_début` et extrayez le jour de la semaine où la pulvérisation a commencé de la même manière qu'indiqué ci-dessus, mais avec la fonction `wday()`. Essayez d'afficher les jours de la semaine écrits plutôt que numériquement.

Une raison pour laquelle vous pourriez vouloir extraire des composants de date spécifiques est lorsque vous souhaitez visualiser vos données, ce qui est très simple à faire dans R en utilisant le package `ggplot2` ! Par exemple, disons que nous voulions comparer les mois où la pulvérisation commence, nous pouvons le faire en créant une nouvelle variable de mois avec la fonction `month()` et en traçant un graphique à barres avec `geom_bar`.

```
données_pulvérisation %>%
  mutate(mois = month(start_date_default, label = TRUE)) %>%
  ggplot(aes(x = mois)) +
  geom_bar(fill = "orange")
```



Ici, nous pouvons voir que la plupart des campagnes de pulvérisation ont commencé entre juillet et novembre, sans aucune en janvier, février et mars. Les auteurs de l'article à partir duquel ces données ont été extraites ont déclaré que les campagnes de pulvérisation visaient à se terminer juste au début de la saison des pluies (novembre-avril) au Malawi. Cela était à la fois pour des raisons pratiques et en prévision d'une transmission plus élevée du paludisme. Nous pouvons voir ce schéma temporel de pulvérisation reflété dans notre graphique !

Créez un nouveau graphique représentant les mois où la campagne de pulvérisation s'est terminée et comparez-le au graphique de son commencement. Cela correspond-il à vos attentes compte tenu de la variable `durée_pulvérisation` que vous avez créée dans l'exercice de la section précédente ?

Arrondir

Parfois, il est nécessaire d'arrondir nos dates vers le haut ou vers le bas si nous voulons analyser ou visualiser nos données de manière significative. Tout d'abord, voyons ce que nous entendons par "arrondir" avec quelques exemples simples.

Prenons la date du 17 mars 2012. Si nous voulions arrondir vers le bas au mois le plus proche, alors nous utiliserions la fonction `floor_date()` de `lubridate` avec l'argument `unit="month"`.

```
ma_date_inf <- as.Date("2012-03-17")
floor_date(ma_date_inf, unit="month")
```

```
## [1] "2012-03-01"
```

Comme nous pouvons le voir, notre date est maintenant le 1er mars 2012.

Si nous voulions arrondir vers le haut, nous pouvons utiliser la fonction `ceiling_date()`. Essayons ceci avec la date du 3 janvier 2020.

```
ma_date_sup <- as.Date("2020-01-03")
ceiling_date(ma_date_sup, unit="month")
```

```
## [1] "2020-02-01"
```

Avec `ceiling_date()`, le 3 janvier a été arrondi au 1er février.

Enfin, nous pouvons également simplement arrondir sans spécifier vers le haut ou vers le bas, et les dates sont automatiquement arrondies à l'unité spécifiée la plus proche.

```
mes_dates <- as.Date(c("2000-11-03", "2000-11-27"))
round_date(mes_dates, unit="month")
```

```
## [1] "2000-11-01" "2000-12-01"
```

Ici, nous pouvons voir qu'en arrondissant au mois le plus proche, le 3 novembre est arrondi au 1er novembre, et le 27 novembre est arrondi au 1er décembre.

Nous pouvons également arrondir vers le haut ou vers le bas à l'année la plus proche. Que pensez-vous que sera la sortie si nous arrondissons vers le bas la date du 29 novembre 2001 à l'année la plus proche :

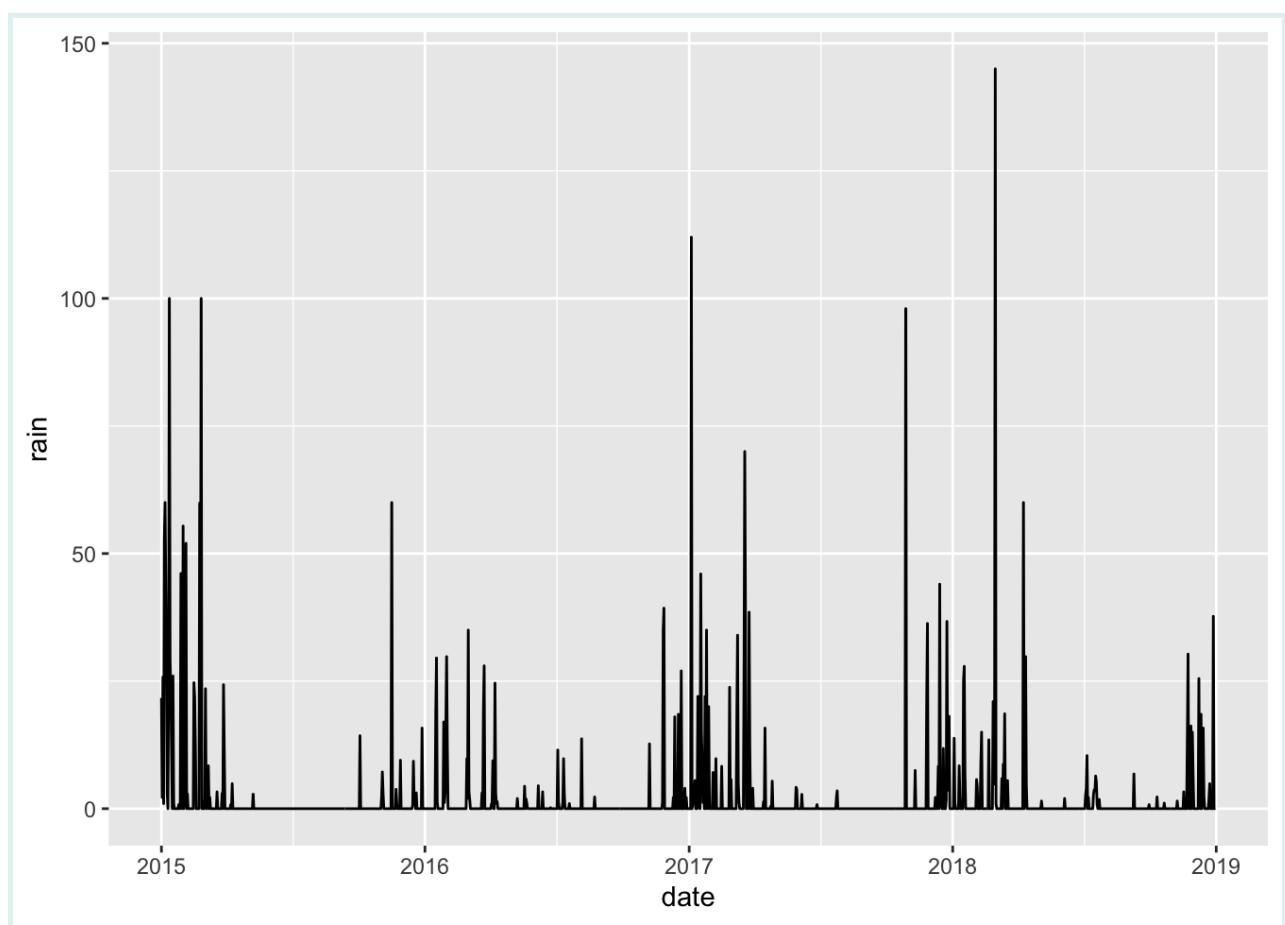
```
date_arrondie <- as.Date("2001-11-29")
floor_date(date_arrondie, unit="year")
```

J'espère que ce que nous entendons par "arrondir" est un peu plus clair ! Alors, pourquoi cela pourrait-il être utile avec nos données ? Eh bien, passons maintenant à nos données météorologiques.

```
météo
```

Comme nous pouvons le voir, nos données météorologiques sont enregistrées quotidiennement, mais ce niveau de détail n'est pas idéal pour étudier comment les schémas météorologiques affectent la transmission du paludisme, qui suit un schéma saisonnier. Les données météorologiques quotidiennes peuvent être assez bruyantes compte tenu de la variation significative d'un jour à l'autre. Par exemple, un graphique des précipitations quotidiennes ne serait pas très informatif. Essayons pour voir :

```
météo %>%
  ggplot() +
  geom_line(aes(date, rain))
```



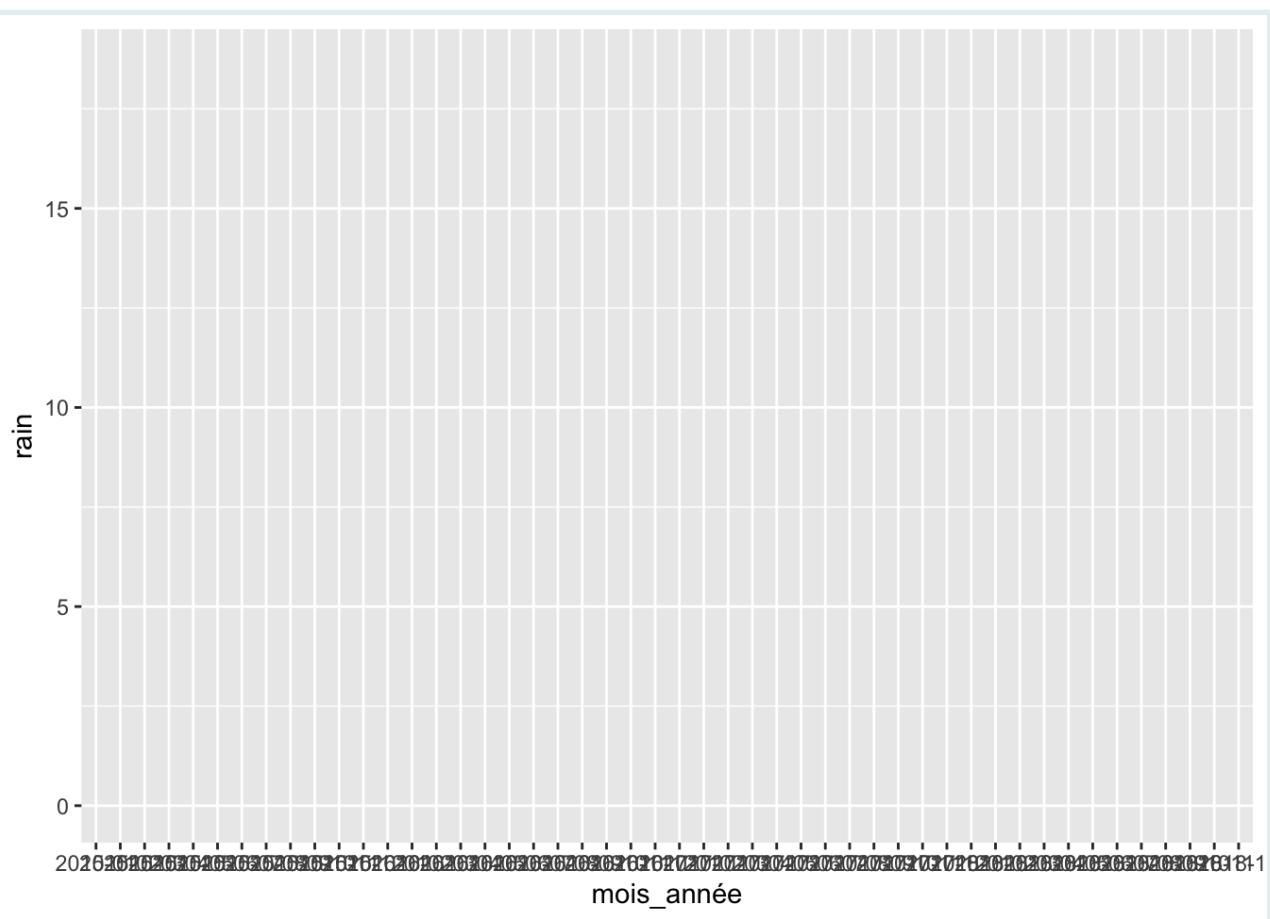
Outre le fait qu'il soit visuellement confus, ce graphique ne permet pas d'illustrer efficacement les tendances saisonnières. L'agrégation mensuelle est une approche plus efficace pour capturer les variations saisonnières et réduire le bruit. Si nous voulions représenter les précipitations mensuelles moyennes, notre première tentative pourrait être d'utiliser la fonction `str_sub()` pour extraire les sept premiers caractères de notre date (le mois et l'année).

```
météo_mauvaise <- météo %>%
  mutate(mois_année=str_sub(date, 1, 7)) %>%
  group_by(mois_année) %>%
  summarise(rain=mean(rain))
météo_mauvaise
```

Cependant, si nous essayons de le représenter, notre nouvelle variable `mois_année` n'est plus une variable de date, nous ne pouvons donc pas représenter des graphiques dans le temps car elle n'est pas continue !

```
météo_mauvaise %>%
  ggplot() +
  geom_line(aes(mois_année, rain))
```

```
## `geom_line()` : Each group consists of only one
## observation.
## i Do you need to adjust the group aesthetic?
```

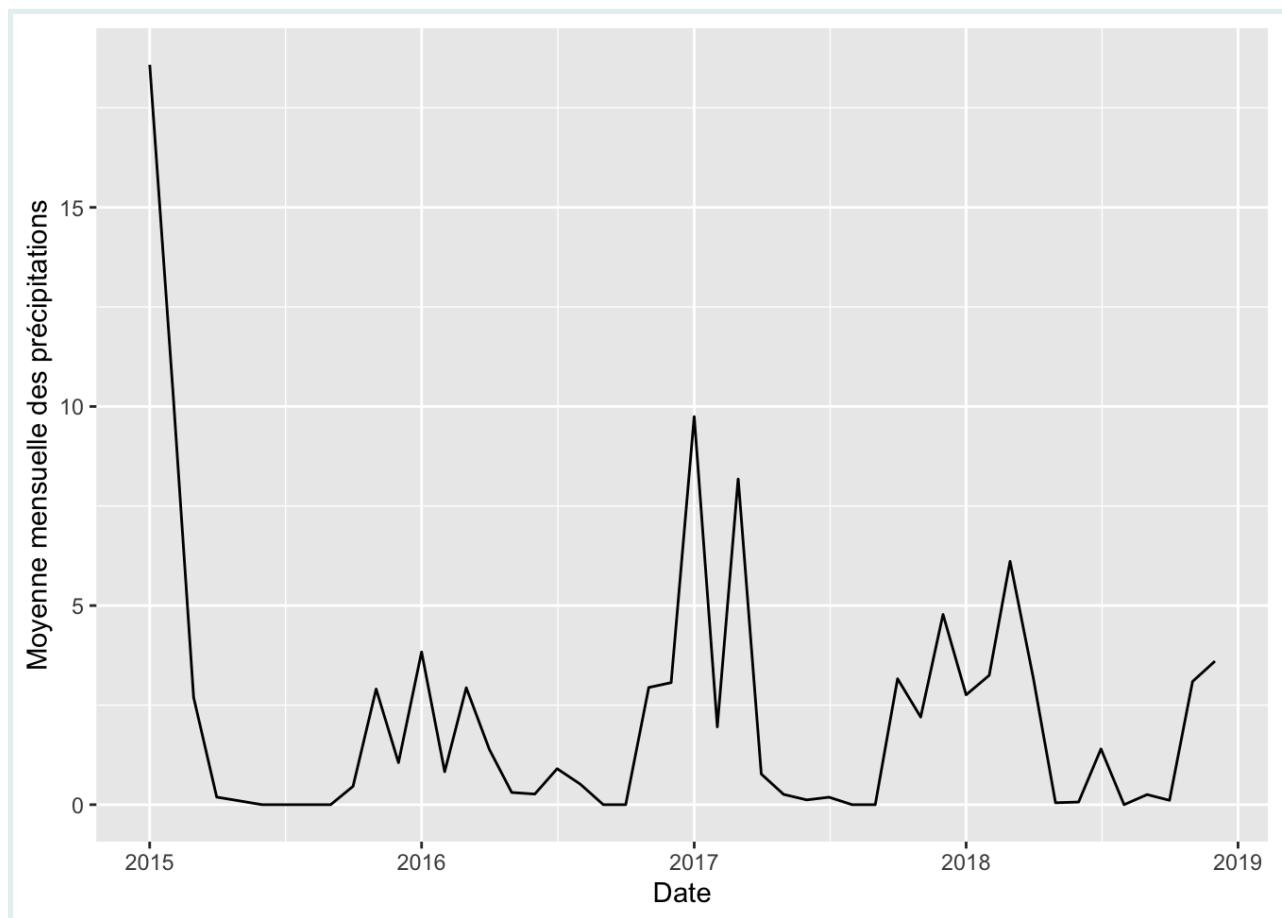


La meilleure façon de faire cela est d'abord d'arrondir nos dates au mois à l'aide de la fonction `floor_date()`, puis de regrouper nos données par notre nouvelle variable `mois_année`, et enfin de calculer la moyenne mensuelle. Essayons maintenant.

```
météo <- météo %>%
  mutate(mois_année=floor_date(date, unit="month")) %>%
  group_by(mois_année) %>%
  summarise(rain=mean(rain))
météo
```

Maintenant nous pouvons représenter nos données et nous aurons un graphique de la pluviométrie mensuelle moyenne sur la période de pulvérisation de 4 ans.

```
météo %>%  
  ggplot() +  
  geom_line(aes(mois_année, rain)) +  
  labs(x="Date", y="Moyenne mensuelle des précipitations")
```



Cela semble beaucoup mieux ! Maintenant, nous obtenons une image beaucoup plus claire des tendances saisonnières et des variations annuelles.

À l'aide des données météorologiques, créez un nouveau graphique représentant les températures minimales et maximales moyennes mensuelles de 2015 à 2019.

WRAP UP!

[XXX NICE WRAP UP MESSAGE OR SUMMARY IF NEEDED HERE XXX]

Answer Key

```
données_pulvérisation %>%
  mutate(jour_semaine_début = wday(start_date_default, label=TRUE)) %>%
  select(village, start_date_default, jour_semaine_début)
```

```
## # A tibble: 5 × 3
##   village      start_date_default jour_semaine_début
##   <chr>        <date>                <ord>
## 1 Mess         2014-04-07            Mon
## 2 Nkombedzi    2014-04-22            Tue
## 3 B Compound   2014-05-13            Tue
## 4 D Compound   2014-05-13            Tue
## 5 Post Office  2014-05-13            Tue
```

Contributors

The following team members contributed to this lesson:

(make sure to update the contributor list accordingly!)

Les factors dans R

GRAPH Network & WHO, supported by the Global Fund to fight HIV, TB & Malaria

October 2023

This document is a draft of a lesson made by the GRAPH Network, a non-profit headquartered at the University of Geneva Global Health Institute, in collaboration with the World Health Organization, under a Global Fund 2023 grant to create e-learning modules to build in-country data capacity for epidemiological and impact analysis for National HIV, TB and malaria programs

Introduction
Objectifs d'apprentissage
Packages
Jeu de données : Mortalité VIH
Qu'est-ce que les facteurs ?
Les facteurs en action
Manipuler les facteurs avec <code>forcats</code>
<code>fct_relevel</code>
<code>fct_reorder</code>
<code>fct_recode</code>
<code>fct_lump</code>
Conclusion
Corrigé
Annexe : Codebook

Introduction

Les facteurs sont une classe de données importante dans R pour représenter et travailler avec des variables catégorielles. Dans cette leçon, nous allons apprendre à créer des facteurs et à les manipuler avec des fonctions du package `forcats`, qui fait partie du `tidyverse`. Plongeons-nous dedans !

Objectifs d'apprentissage

- Vous comprenez ce que sont les facteurs et en quoi ils diffèrent des caractères dans R.
- Vous êtes capable de modifier **l'ordre** des niveaux des facteurs.
- Vous êtes capable de modifier **la valeur** des niveaux des facteurs.

Packages

```
# Load packages
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse,
               here)
```

Jeu de données : Mortalité VIH

Nous allons utiliser un jeu de données contenant des informations sur la mortalité VIH en Colombie de 2010 à 2016, hébergé sur la plateforme de données ouvertes 'Datos Abiertos Colombia'. Vous pouvez en savoir plus et accéder à l'ensemble du jeu de données [ici](#).

Chaque ligne correspond à un individu décédé du SIDA ou de complications liées au SIDA.

```
hiv_mort <- read_csv(here("data/colombia_hiv_deaths_2010_to_2016"))
```

```
## # A tibble: 5 × 25
##   municipality_type death_location birth_date birth_year
##   <chr>              <chr>          <date>        <dbl>
## 1 Municipal head    Hospital/clinic 1956-05-26  1956
## 2 Municipal head    Hospital/clinic 1983-10-10  1983
## 3 Municipal head    Hospital/clinic 1967-11-22  1967
## 4 Municipal head    Home/address    1964-03-14  1964
## 5 Municipal head    Hospital/clinic 1960-06-27  1960
## # ... with 16 more variables: birth_month <dbl>, birth_day <dbl>,
## #   death_year <dbl>, death_month <dbl>, death_day <dbl>,
## #   <chr> <dbl> <chr> <dbl> <chr> <dbl>
## #   1 May      26   2012 Sep   14
## #   2 Oct      10   2012 Mar   17
## #   3 Nov      22   2011 Oct   19
## #   4 Mar      14   2012 Nov   19
## #   5 Jun      27   2012 Jan   13
## # ... with 16 more variables: age_at_death <dbl>, gender <chr>,
## #   education_level <chr>, occupation <chr>, ...
```

Voir l'annexe au bas pour le dictionnaire de données décrivant toutes les variables.

Qu'est-ce que les facteurs ?

Les facteurs sont une classe de données importante dans R utilisée pour représenter des variables catégorielles.

Une variable catégorielle prend un ensemble limité de valeurs ou niveaux possibles. Par exemple, pays, race ou affiliation politique. Celles-ci diffèrent des variables texte libre qui prennent des valeurs arbitraires, comme des noms de personnes, titres de livres ou commentaires de médecins.



Rappel des principales classes de données dans R

RECAP



- Numérique : Représente des données numériques continues, incluant des nombres décimaux.
- Entier : Spécifiquement pour les nombres entiers sans décimales.
- Caractère : Utilisé pour les données textuelles ou chaînes de caractères.
- Logique : Représente des valeurs booléennes (VRAI ou FAUX).
- Facteur : Utilisé pour les données catégorielles avec des niveaux ou catégories prédéfinis.
- Date : Représente des dates sans heures.

Les facteurs ont quelques avantages clés par rapport aux vecteurs de caractères pour travailler avec des données catégorielles dans R :

- Les facteurs sont stockés dans R de manière légèrement plus efficace que les caractères.
- Certaines fonctions statistiques, comme `lm()`, nécessitent que les variables catégorielles soient passées en paramètre sous forme de facteurs.
- Les facteurs permettent de contrôler l'ordre des catégories ou niveaux. Cela permet de trier et tracer correctement les données catégorielles.

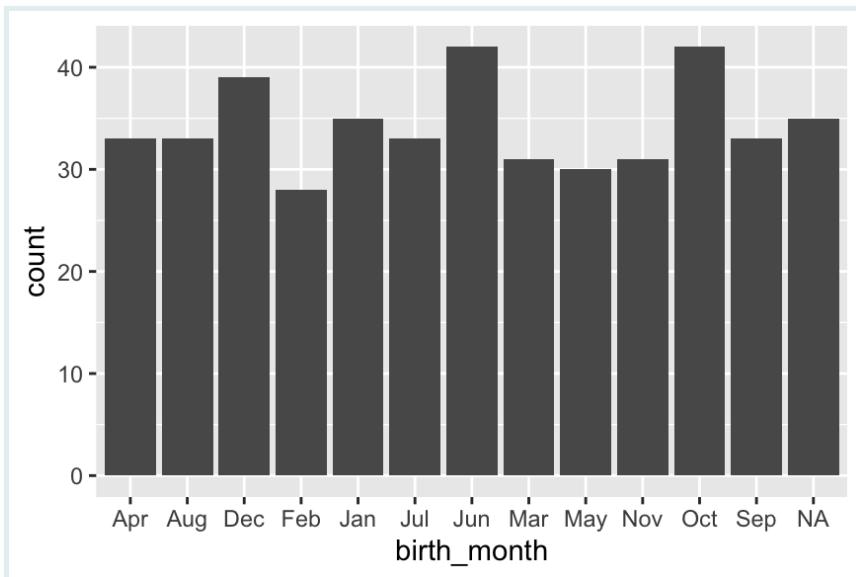
Ce dernier point, le contrôle de l'ordre des niveaux de facteurs, sera notre objectif principal.

Les facteurs en action

Voyons un exemple concret de l'intérêt des facteurs en utilisant le jeu de données `hiv_mort` que nous avons chargé précédemment.

Supposons que vous souhaitez visualiser les patients du jeu de données par leur mois de naissance. Nous pouvons le faire avec `ggplot` :

```
ggplot(hiv_mort) +  
  geom_bar(aes(x = birth_month))
```



Cependant, il y a un problème : l'axe des x (qui représente les mois) est classé alphabétiquement, avec Avril en premier à gauche, puis Août, etc. Mais les mois devraient suivre un ordre chronologique spécifique !

Nous pouvons arranger le graphique dans l'ordre souhaité en créant un facteur avec la fonction `factor()` :

```
hiv_mort_modified <-  
  hiv_mort %>%  
  mutate(birth_month = factor(x = birth_month,  
                             levels = c("Jan", "Feb", "Mar", "Apr",  
                                      "May", "Jun", "Jul", "Aug",  
                                      "Sep", "Oct", "Nov", "Dec")))
```

La syntaxe est simple : l'argument `x` prend la colonne de caractères d'origine, `birth_month`, et l'argument `levels` prend la séquence désirée de mois.

Lorsque nous inspectons le type de données de la variable `birth_month`, nous pouvons voir sa transformation :

```
# Modified dataset  
class(hiv_mort_modified$birth_month)
```

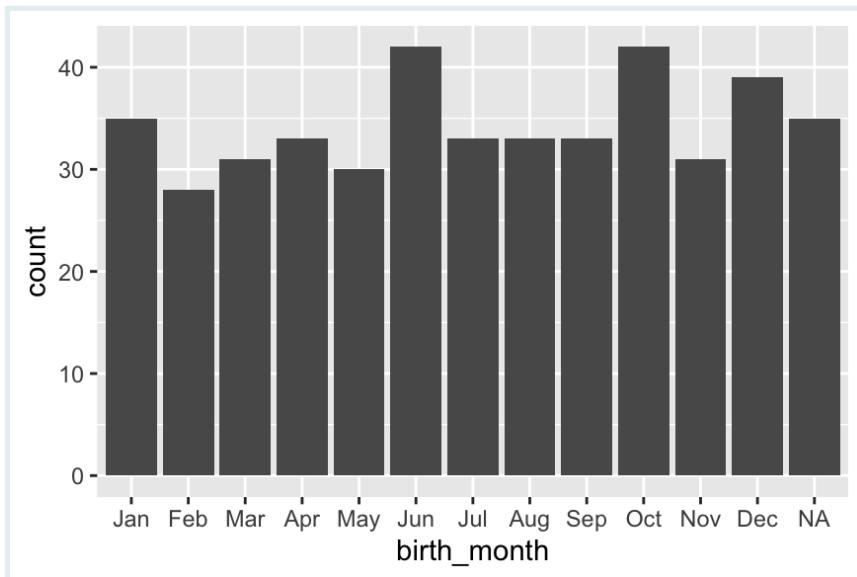
```
## [1] "factor"
```

```
# Original dataset  
class(hiv_mort$birth_month)
```

```
## [1] "character"
```

Maintenant, nous pouvons régénérer le ggplot avec le jeu de données modifié :

```
ggplot(hiv_mort_modified) +
  geom_bar(aes(x = birth_month))
```



Les mois sur l'axe des x sont maintenant affichés dans l'ordre que nous avons spécifié.

La nouvelle variable de facteur respectera également l'ordre défini dans d'autres contextes. Par exemple, comparez comment la fonction `count()` affiche les deux tableaux de fréquences ci-dessous :

```
# Original dataset
count(hiv_mort, birth_month)
```

```
## # A tibble: 13 × 2
##   birth_month     n
##   <chr>       <int>
## 1 Apr          33
## 2 Aug          33
## 3 Dec          39
## 4 Feb          28
## 5 Jan          35
## 6 Jul          33
## 7 Jun          42
## 8 Mar          31
## 9 May          30
## 10 Nov         31
## 11 Oct         42
```

```
## 12 Sep          33
## 13 <NA>         35
```

```
# Modified dataset
count(hiv_mort_modified, birth_month)
```

```
## # A tibble: 13 × 2
##   birth_month     n
##   <fct>       <int>
## 1 Jan            35
## 2 Feb            28
## 3 Mar            31
## 4 Apr            33
## 5 May            30
## 6 Jun            42
## 7 Jul            33
## 8 Aug            33
## 9 Sep            33
## 10 Oct           42
## 11 Nov           31
## 12 Dec           39
## 13 <NA>          35
```

Soyez vigilant lorsque vous créez des niveaux de facteurs ! Toutes les valeurs de la variable **qui ne sont pas incluses** dans l'ensemble des niveaux fournis à l'argument `levels` seront converties en NA.

Par exemple, si nous avons manqué certains mois dans notre exemple :

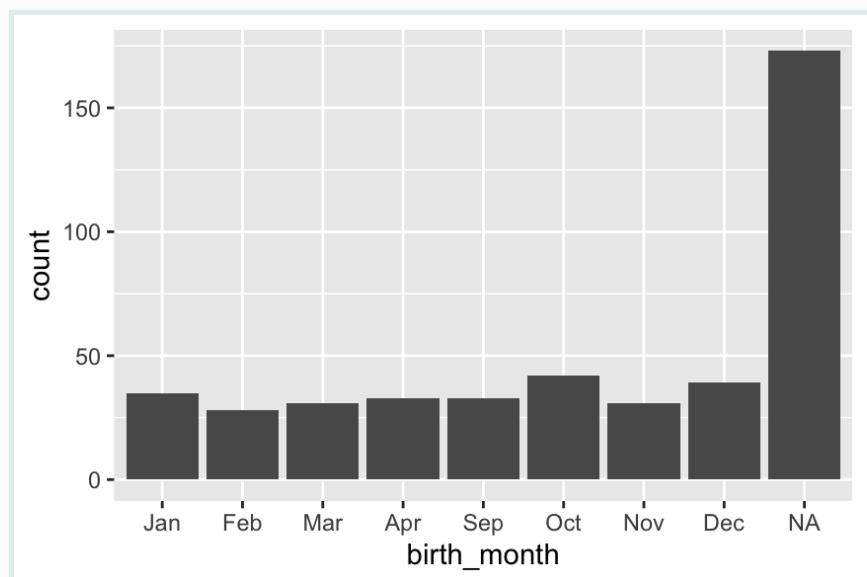
WATCH OUT



```
hiv_mort_missing_months <-
  hiv_mort %>%
  mutate(birth_month = factor(x = birth_month,
                             levels = c("Jan", "Feb", "Mar",
                                       "Apr",
                                       # missing months
                                       "Sep", "Oct", "Nov",
                                       "Dec")))
```

Nous finissons avec beaucoup de valeurs NA :

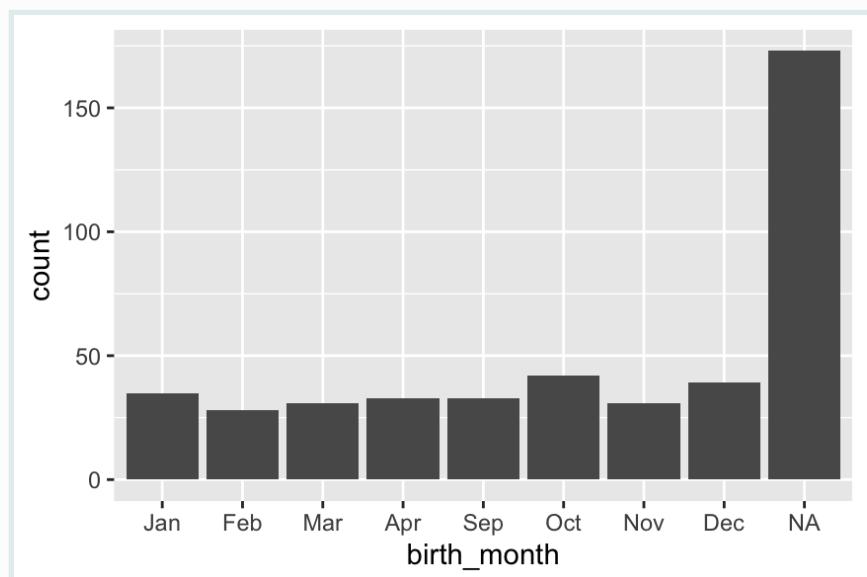
```
ggplot(hiv_mort_missing_months) +
  geom_bar(aes(x = birth_month))
```



Vous aurez le même problème s'il y a des erreurs de frappe :

```
hiv_mort_with_typos <-
  hiv_mort %>%
  mutate(birth_month = factor(x = birth_month,
                             levels = c("Jan", "Feb", "Mar",
                                         "Apr",
                                         "Moy", "Jon", "Jol",
                                         "Aog", # typos
                                         "Sep", "Oct", "Nov",
                                         "Dec")))

ggplot(hiv_mort_with_typos) +
  geom_bar(aes(x = birth_month))
```

WATCH OUT

Vous pouvez utiliser le facteur sans niveaux. Il utilise simplement l'arrangement par défaut (alphabétique) des niveaux.

```
hiv_mort_default_factor <- hiv_mort %>%
  mutate(birth_month = factor(x = birth_month))
```

SIDE NOTE

```
class(hiv_mort_default_factor$birth_month)
```

```
## [1] "factor"
```

```
levels(hiv_mort_default_factor$birth_month)
```

```
## [1] "Apr" "Aug" "Dec" "Feb" "Jan" "Jul" "Jun" "Mar"
## [9] "May" "Nov" "Oct" "Sep"
```

Q: Facteur de genre

En utilisant le jeu de données `hiv_mort`, convertissez la variable `gender` en un facteur avec les niveaux “Femme” et “Homme”, dans cet ordre.

Q: Repérage des erreurs

Quelles erreurs pouvez-vous repérer dans l'extrait de code suivant ? Quelles sont les conséquences de ces erreurs ?

```
hiv_mort <-
  hiv_mort %>%
  mutate(birth_month = factor (x = birth_month,
                               levels = c("Jan", "Feb", "Mar", "Apr",
                                         "Mai", "Jun", "Jul", "Sep",
                                         "Oct", "Nov.", "Dec")))
```

Q: Avantage des facteurs

Quel est l'avantage principal de l'utilisation de facteurs par rapport aux caractères pour les données catégorielles dans R ?

- a. Il est plus facile d'effectuer des manipulations de chaînes sur les facteurs.
 - b. Les facteurs permettent un meilleur contrôle de l'ordre des données catégorielles.
 - c. Les facteurs augmentent la précision des modèles statistiques.
-

Manipuler les facteurs avec `forcats`

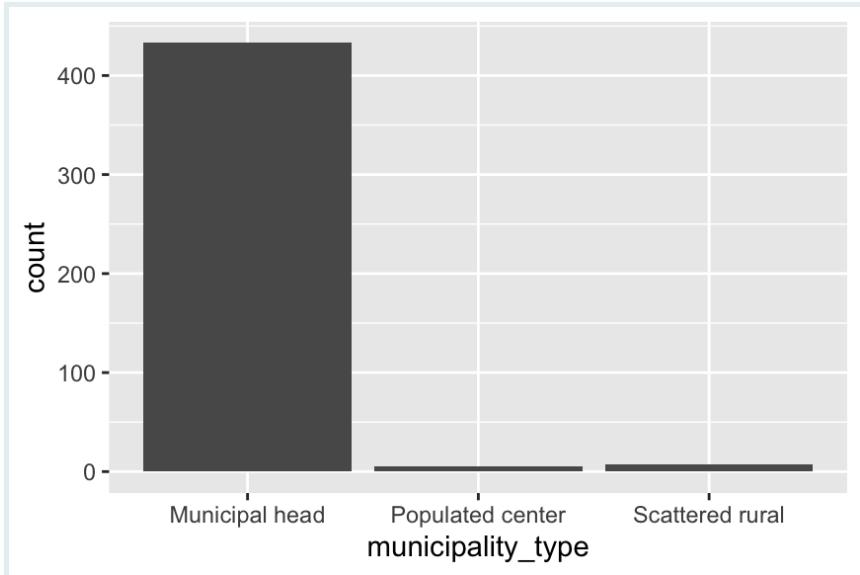
Les facteurs sont très utiles, mais ils peuvent parfois être un peu fastidieux à manipuler uniquement avec les fonctions de base R. Heureusement, le package `forcats`, membre du tidyverse, propose un ensemble de fonctions qui simplifient grandement la manipulation des facteurs. Nous allons examiner quatre fonctions ici, mais il y en a beaucoup d'autres, donc nous vous encourageons à explorer le site web de `forcats` par vous-même [ici](#)!

`fct_relevel`

La fonction `fct_relevel()` est utilisée pour changer manuellement l'ordre des niveaux de facteurs.

Par exemple, disons que nous voulons visualiser la fréquence des individus de notre jeu de données par type de municipalité. Lorsque nous créons un graphique en barres, les valeurs sont classées par ordre alphabétique par défaut :

```
ggplot(hiv_mort) +
  geom_bar(aes(x = municipality_type))
```



Mais que se passerait-il si nous voulions qu'une valeur spécifique, disons "Populated center", apparaisse en premier dans le graphique ?

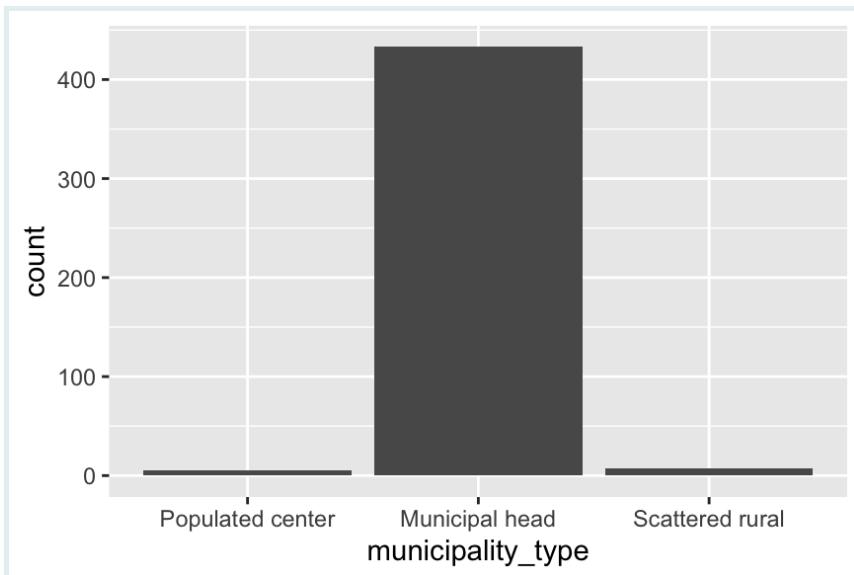
Cela peut être réalisé en utilisant `fct_relevel()`. Voici comment :

```
hiv_mort_pop_center_first <-
  hiv_mort %>%
  mutate(municipality_type = fct_relevel(municipality_type, "Populated
  center"))
```

La syntaxe est simple : nous passons la variable factorielle en premier argument, et le niveau que nous voulons déplacer au début en second argument.

Maintenant lorsque nous traçons :

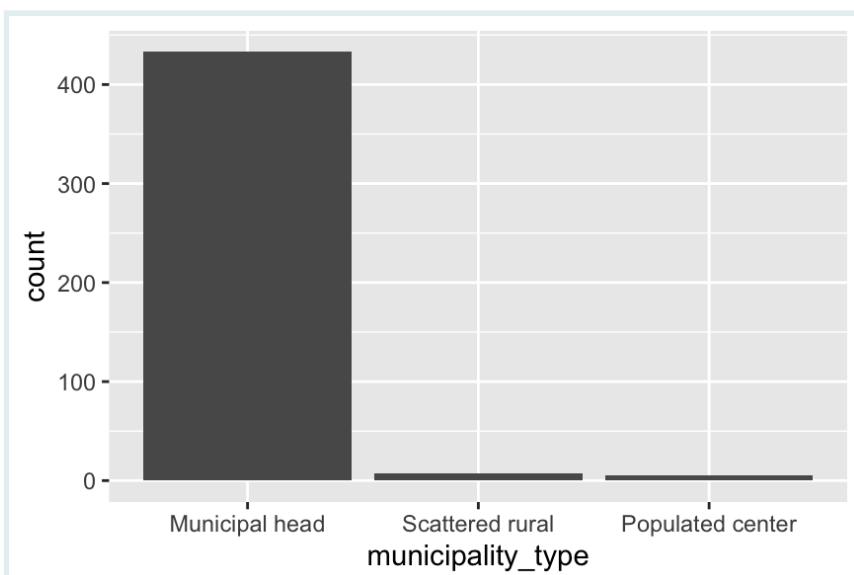
```
ggplot(hiv_mort_pop_center_first) +
  geom_bar(aes(x = municipality_type))
```



Le niveau "Centre peuplé" est maintenant le premier.

Nous pouvons déplacer le niveau "Populated center" à une position différente avec l'argument `after` :

```
hiv_mort %>%
  mutate(municipality_type = fct_relevel(municipality_type, "Populated
    center",
                                             after = 2)) %>%
# pipe directly into to plot to visualize change
  ggplot() +
  geom_bar(aes(x = municipality_type))
```

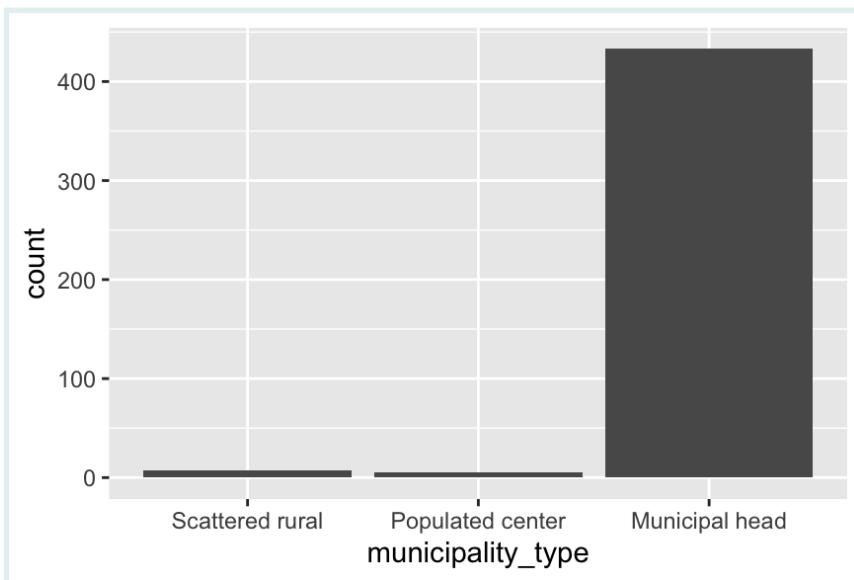


La syntaxe est : spécifier le facteur, le niveau à déplacer, et utiliser l'argument `after` pour définir à quelle position le placer après.

Nous pouvons également déplacer plusieurs niveaux à la fois en fournissant ces niveaux à `fct_relevel()` :

Ci-dessous, nous disposons tous les niveaux de facteurs pour le type de municipalité dans l'ordre souhaité :

```
hiv_mort %>%
  mutate(municipality_type = fct_relevel(municipality_type,
                                            "Scattered rural",
                                            "Populated center",
                                            "Municipal head")) %>%
  ggplot() +
  geom_bar(aes(x = municipality_type))
```



C'est similaire à la création d'un facteur depuis le début avec des niveaux dans cet ordre :

```
hiv_mort %>%
  mutate(municipality = factor(municipality_type,
                                levels = c("Scattered rural",
                                           "Populated center",
                                           "Municipal head")))
```



PRACTICE Q: Utiliser `fct_relevel`

En utilisant le jeu de données `hiv_mort`, convertissez la variable `death_location` en facteur de sorte que 'Home/address' soit le premier



niveau. Ensuite, créez un graphique en barres montrant le décompte des individus du jeu de données par `death_location`.

fct_reorder

`fct_reorder()` est utilisé pour réorganiser les niveaux d'un facteur en fonction des valeurs d'une autre variable.

Pour illustrer, créons un tableau récapitulatif avec le nombre de décès, l'âge moyen et médian au décès pour chaque municipalité :

```
summary_per_muni <-  
  hiv_mort %>%  
  group_by(municipality_name) %>%  
  summarise(n_deceased = n(),  
            mean_age_death = mean(age_at_death, na.rm = T),  
            med_age_death = median(age_at_death, na.rm = T))  
  
summary_per_muni  
  
## # A tibble: 25 × 4  
##   municipality_name n_deceased mean_age_death  
##   <chr>                <int>             <dbl>  
## 1 Aguadas                  2                 42  
## 2 Anserma                 15                37.4  
## 3 Aranzazu                 2                 37.5  
## 4 Belalcázar                4                 38.8  
## 5 Chinchiná                 62                43.6  
## 6 Filadelfia                  5                42.6  
## 7 La Dorada                 46                41.0  
## 8 La Merced                  3                 27  
## 9 Manizales                 199               41.0  
## 10 Manzanares                 3                38.3  
##   med_age_death  
##   <dbl>  
## 1 42  
## 2 37.5  
## 3 37.5  
## 4 41  
## 5 42.5  
## 6 43  
## 7 41  
## 8 28  
## 9 41  
## 10 34  
## # i 15 more rows
```

Lorsque nous traçons l'une des variables, nous voudrons peut-être arranger les niveaux de facteurs par cette variable numérique. Par exemple, pour ordonner la municipalité par la colonne de l'âge moyen :

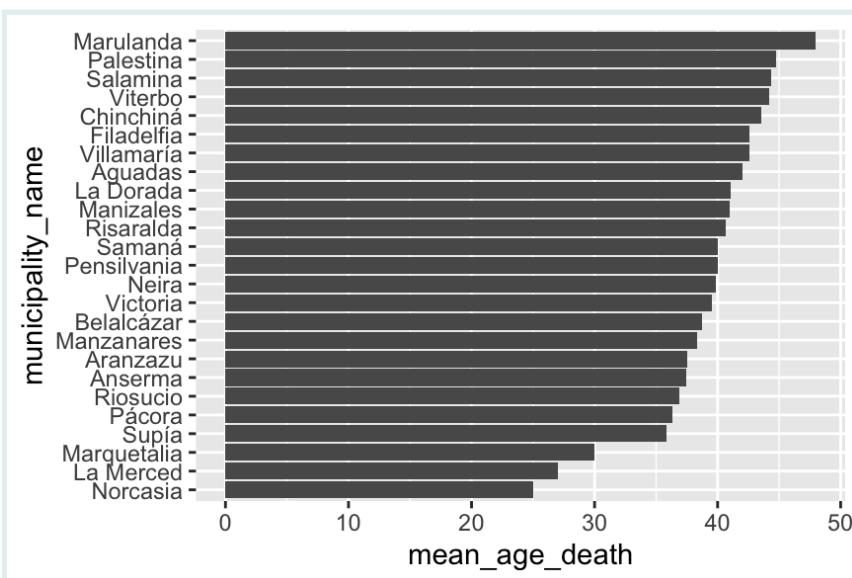
```
summary_per_muni_reordered <-
  summary_per_muni %>%
  mutate(municipality_name = fct_reorder(.f = municipality_name,
                                         .x = mean_age_death))
```

La syntaxe est :

- `.f` - le facteur à réorganiser
- `.x` - le vecteur numérique déterminant le nouvel ordre

Nous pouvons maintenant tracer un joli graphique en barres :

```
ggplot(summary_per_muni_reordered) +
  geom_col(aes(y = municipality_name, x = mean_age_death))
```



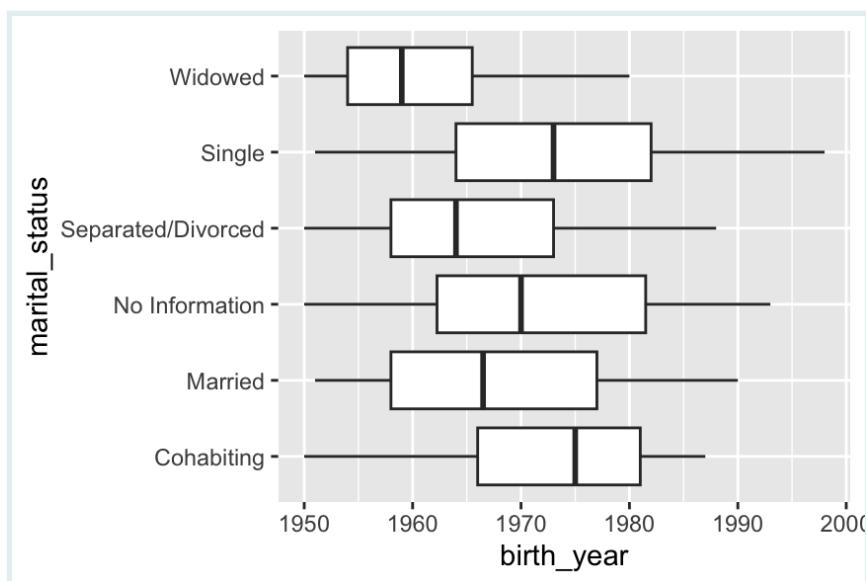
Q: Utiliser `fct_reorder`

En partant du dataframe `summary_per_muni`, réorganisez la municipalité (`municipality_name`) par la colonne `med_age_death` et tracez le graphique en barres réorganisé.

L'argument .fun

Parfois, nous voulons que les catégories de notre graphique apparaissent dans un ordre spécifique déterminé par une statistique sommaire. Par exemple, considérons le diagramme en boîte de `birth_year` par `marital_status` :

```
ggplot(hiv_mort, aes(y = marital_status, x = birth_year)) +  
  geom_boxplot()
```



Le diagramme en boîte affiche la médiane `birth_year` pour chaque catégorie de `marital_status` comme une ligne au milieu de chaque boîte. Nous voudrions peut-être arranger les catégories `marital_status` dans l'ordre de ces médianes. Mais si nous créons un tableau récapitulatif avec les médianes, comme nous l'avons fait précédemment avec `summary_per_muni`, nous ne pouvons pas créer de diagramme en boîte avec lui (allez regarder le dataframe `summary_per_muni` pour le vérifier vous-même).

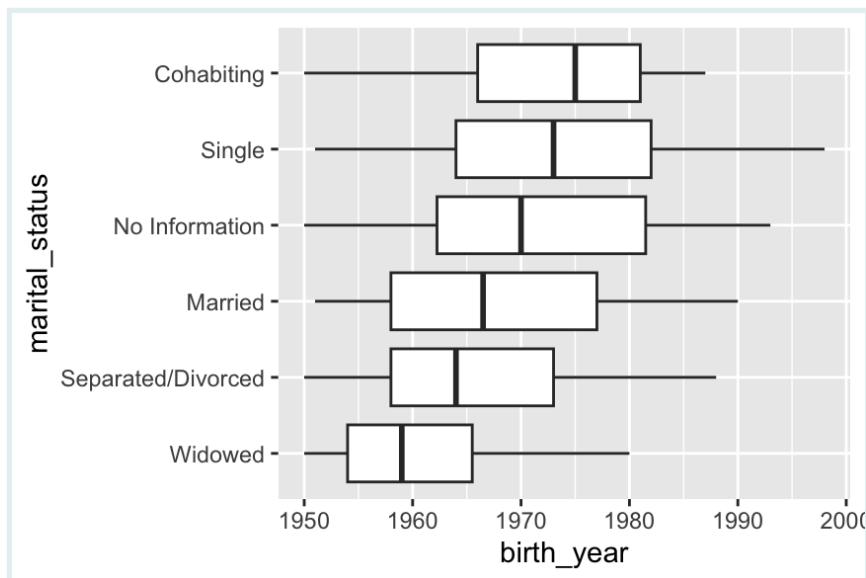
C'est là que intervient l'argument `.fun` de `fct_reorder()`. L'argument `.fun` nous permet de spécifier une fonction de résumé qui sera utilisée pour calculer le nouvel ordre des niveaux :

```
hiv_mort_arranged_marital <-  
  hiv_mort %>%  
  mutate(marital_status = fct_reorder(.f = marital_status,  
                                     .x = birth_year,  
                                     .fun = median,  
                                     na.rm = TRUE))
```

Dans ce code, nous réorganisons le facteur `marital_status` en fonction de la médiane de `birth_year`. Nous incluons l'argument `na.rm = TRUE` pour ignorer les valeurs NA lors du calcul de la médiane.

Maintenant, lorsque nous créons notre diagramme en boîte, les catégories `marital_status` sont classées par la médiane `birth_year` :

```
ggplot(hiv_mort_arranged_marital, aes(y = marital_status, x = birth_year)) +  
  geom_boxplot()
```



Nous pouvons voir que les individus ayant le statut marital “cohabiting” ont tendance à être les plus jeunes (ils sont nés les dernières années).



Q: Utiliser `.fun`

En utilisant le jeu de données `hiv_mort`, faites un diagramme en boîte de `birth_year` par `health_insurance_status`, où les catégories `health_insurance_status` sont disposées par la médiane `birth_year`.

fct_recode

La fonction `fct_recode()` nous permet de modifier manuellement les valeurs des niveaux de facteurs. Cette fonction peut être particulièrement utile lorsque vous devez renommer des catégories ou lorsque vous souhaitez fusionner plusieurs catégories en une seule.

Par exemple, nous pouvons renommer ‘Municipal head’ en ‘City’ dans la variable `municipality_type` :

```
hiv_mort_muni_recode <- hiv_mort %>%  
  mutate(municipality_type = fct_recode(municipality_type,
```

```
"City" = "Municipal head"))
# View the changelevels(hiv_mort_muni_recode$municipality_type)
```

```
## [1] "City"           "Populated center"
## [3] "Scattered rural"
```

Dans le code ci-dessus, `fct_recode()` prend deux arguments : la variable factorielle que vous souhaitez modifier (`municipality_type`) et l'ensemble des paires nom-valeur qui définissent le recodage. Le nouveau niveau ("City") est à gauche du signe égal et l'ancien niveau ("Municipal head") est à droite.

`fct_recode()` est particulièrement utile pour compresser plusieurs catégories en moins de niveaux.

Nous pouvons explorer cela en utilisant la variable `education_level`. Actuellement, elle possède six catégories :

```
count(hiv_mort, education_level)
```

```
## # A tibble: 6 × 2
##   education_level     n
##   <chr>              <int>
## 1 No information     88
## 2 None                22
## 3 Post-secondary      29
## 4 Preschool            3
## 5 Primary             187
## 6 Secondary            116
```

Par souci de simplicité, regroupons-les en seulement trois catégories - "primary & below", "secondary & above" et "others":

```
hiv_mort_educ_simple <-
  hiv_mort %>%
  mutate(education_level = fct_recode(education_level,
                                       "primary & below" = "Primary",
                                       "primary & below" = "Preschool",
                                       "secondary & above" = "Secondary",
                                       "secondary & above" = "Post-
                                       secondary",
                                       "others" = "No information",
                                       "others" = "None"))
```

Cela condense joliment les catégories :

```
count(hiv_mort_educ_simple, education_level)
```

```
## # A tibble: 3 × 2
##   education_level     n
##   <fct>             <int>
## 1 others              110
## 2 secondary & above  145
## 3 primary & below   190
```

Par mesure de précaution, nous pouvons arranger les niveaux dans un ordre raisonnable, avec "others" comme dernier niveau :

```
hiv_mort_educ_sorted <-
  hiv_mort_educ_simple %>%
  mutate(education_level = fct_relevel(education_level,
                                         "primary & below",
                                         "secondary & above",
                                         "others"))
```

Cela condense joliment les catégories :

```
count(hiv_mort_educ_sorted, education_level)
```

```
## # A tibble: 3 × 2
##   education_level     n
##   <fct>             <int>
## 1 primary & below   190
## 2 secondary & above 145
## 3 others              110
```

Q: Utiliser fct_recode



En utilisant le jeu de données `hiv_mort`, convertissez `death_location` en facteur.

Ensuite, utilisez `fct_recode()` pour renommer 'Public way' dans `death_location` en 'Public place'. Tracez les fréquences de la variable mise à jour.



`fct_recode` vs `case_when/if_else`

Vous vous demandez peut-être pourquoi nous avons besoin de `fct_recode()` alors que nous pouvons utiliser `case_when()` ou `if_else()` voire même `recode()` pour substituer des valeurs spécifiques. Le problème est que ces autres fonctions peuvent perturber votre variable factorielle.

Pour illustrer, disons que nous choisissons d'utiliser `case_when()` pour apporter une modification à la variable `education_level` du dataframe `hiv_mort_educ_sorted`.

Pour rappel, cette variable est un facteur avec trois niveaux, arrangés dans un ordre spécifié, avec "primary & below" en premier et "others" en dernier :

```
count(hiv_mort_educ_sorted, education_level)
```

```
## # A tibble: 3 × 2
##   education_level     n
##   <fct>             <int>
## 1 primary & below    190
## 2 secondary & above  145
## 3 others              110
```

SIDE NOTE



Disons que nous voulions remplacer "others" par "other", en enlevant le "s". Nous pouvons écrire :

```
hiv_mort_educ_other <-
  hiv_mort_educ_sorted %>%
  mutate(education_level = if_else(education_level ==
    "others",
    "other", education_level))
```

Après cette opération, la variable n'est plus un facteur :

```
class(hiv_mort_educ_other$education_level)
```

```
## [1] "character"
```

```
count(hiv_mort_educ_other, education_level)
```

```
## # A tibble: 3 × 2
##   education_level     n
##   <chr>             <int>
## 1 other              110
## 2 primary & below    190
## 3 secondary & above  145
```

Cependant, si nous avions utilisé `fct_recode()` pour le recodage, nous n'aurions pas ce problème :

```
hiv_mort_educ_other_fct <-
  hiv_mort_educ_simple %>%
  mutate(education_level = fct_recode(education_level, "other" =
  "others"))
```

SIDE NOTE



La variable reste un facteur :

```
class(hiv_mort_educ_other_fct$education_level)
```

```
## [1] "factor"
```

Et si nous créons un tableau ou un graphique, notre ordre est préservé : “primary”, “secondary”, puis “other”:

```
count(hiv_mort_educ_other_fct, education_level)
```

```
## # A tibble: 3 × 2
##   education_level     n
##   <fct>             <int>
## 1 other              110
## 2 secondary & above  145
## 3 primary & below   190
```

fct_lump

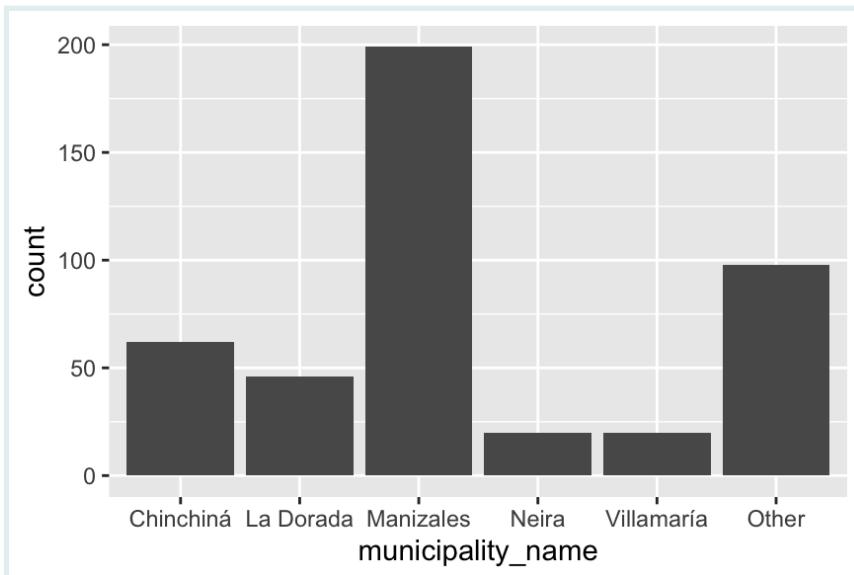
Parfois, nous avons trop de niveaux pour un tableau d'affichage ou un graphique, et nous voulons regrouper les niveaux les moins fréquents dans une seule catégorie, généralement appelée 'Other'.

C'est là que la fonction pratique `fct_lump()` intervient.

Dans l'exemple ci-dessous, nous regroupons les municipalités les moins fréquentes dans 'Other', en ne conservant que les 5 municipalités les plus fréquentes :

```
hiv_mort_lump_muni <- hiv_mort %>%
  mutate(municipality_name = fct_lump(municipality_name, n = 5))

ggplot(hiv_mort_lump_muni, aes(x = municipality_name)) +
  geom_bar()
```

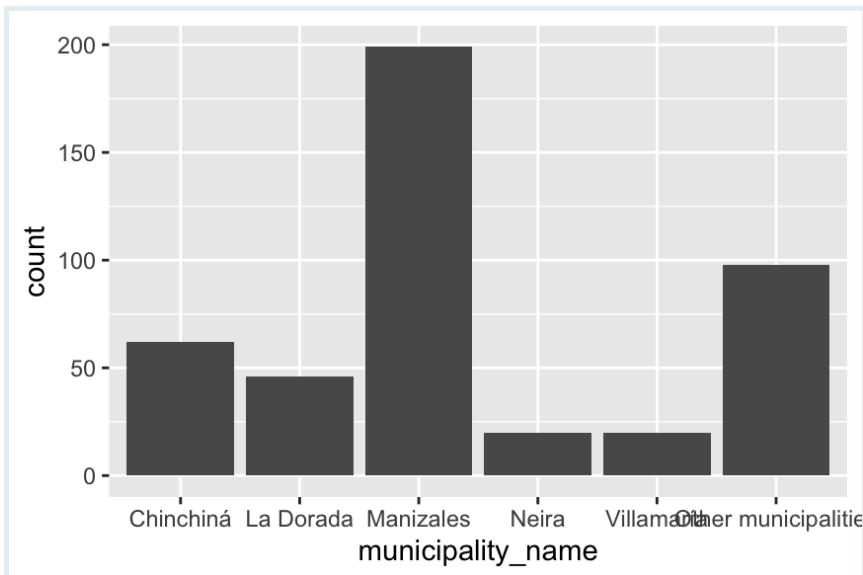


Dans l'utilisation ci-dessus, le paramètre `n = 5` signifie que les cinq municipalités les plus fréquentes sont conservées, et le reste est regroupé dans 'Other'.

Nous pouvons fournir un nom personnalisé pour l'autre catégorie avec l'argument `other_level`. Ci-dessous, nous utilisons le nom "Other municipalities".

```
hiv_mort_lump_muni_other_name <- hiv_mort %>%
  mutate(municipality_name = fct_lump(municipality_name, n = 5,
                                         other_level = "Other municipalities"))

ggplot(hiv_mort_lump_muni_other_name, aes(x = municipality_name)) +
  geom_bar()
```



De cette façon, `fct_lump()` est un outil pratique pour condenser les facteurs avec de nombreux niveaux peu fréquents en un nombre plus gérable de catégories.



Q: Utiliser `fct_lump`

En partant du jeu de données `hiv_mort`, utilisez `fct_lump()` pour créer un diagramme en barres avec la fréquence des 10 occupations les plus courantes.

Regroupez les occupations restantes dans une catégorie 'Other'.

Mettez `occupation` sur l'axe des y, et non sur l'axe des x, pour éviter le chevauchement des étiquettes.

Conclusion

Félicitations d'être arrivé jusqu'au bout. Dans cette leçon, vous avez appris les détails sur la classe de données, **les facteurs**, et comment les manipuler en utilisant des opérations de base comme `fct_relevel()`, `fct_reorder()`, `fct_recode()` et `fct_lump()`.

Bien qu'elles couvrent des tâches courantes comme le réordonnancement, le recodage et la fusion de niveaux, cette introduction ne fait qu'effleurer la surface de ce qui est

possible avec le package `forcats`. N'hésitez pas à explorer davantage sur le site web [forcats website](#).

Maintenant que vous comprenez les bases du travail avec les facteurs, vous êtes équipé pour représenter correctement vos données catégorielles dans R pour l'analyse et la visualisation en aval.

Corrigé

Q: Facteur de genre

```
hiv_mort_q1 <- hiv_mort %>%
  mutate(gender = factor(x = gender,
    levels = c("Female", "Male")))
```

Q: Repérage des erreurs

Erreurs : - "Mai" devrait être "May". - "Nov." a un point en trop. - "Aug" est manquant dans la liste des mois.

Conséquences :

Toutes les lignes avec les valeurs "May", "Nov" ou "Aug" pour `death_month` seront converties en NA dans la nouvelle variable `death_month`. Si vous créez des graphiques, `ggplot` supprimera ces niveaux avec seulement des valeurs NA.

Q: Avantage des facteurs

- Les facteurs permettent un meilleur contrôle de l'ordre des données catégorielles.

Les deux autres déclarations ne sont pas vraies.

Si vous voulez appliquer des opérations sur les chaînes de caractères comme `substr()`, `strsplit()`, `paste()`, etc., il est en fait plus simple d'utiliser des vecteurs de caractères que des facteurs.

Et bien que de nombreuses fonctions statistiques attendent des facteurs, et non des caractères, pour les prédicteurs catégoriels, cela ne les rend pas plus "précises".

Q: Utiliser `fct_relevel`

Q: Utiliser `fct_reorder`

[Q: Utiliser .fun](#)

[Q: Utiliser fct_recode](#)

[Q: Utiliser fct_lump](#)

Annexe : Codebook

Les variables du jeu de données sont :

- `municipality` : localisation municipale générale du patient [chr]
- `death_location` : lieu où le patient est décédé [chr]
- `birth_date` : date de naissance complète, formatée "YYYY-MM-DD" [date]
- `birth_year` : année de naissance du patient [dbl]
- `birth_month` : mois de naissance du patient [chr]
- `birth_day` : jour de naissance du patient [dbl]
- `death_year` : année de décès du patient [dbl]
- `death_month` : mois de décès du patient [chr]
- `death_day` : jour de décès du patient [dbl]
- `gender` : genre du patient [chr]
- `education_level` : plus haut niveau d'études atteint par le patient [chr]
- `occupation` : profession du patient [chr]
- `racial_id` : race du patient [chr]
- `municipality_code` : localisation municipale spécifique du patient [chr]
- `primary_cause_death_description` : cause primaire de décès du patient [chr]
- `primary_cause_death_code` : code de la cause primaire de décès [chr]
- `secondary_cause_death_description` : cause secondaire de décès du patient [chr]
- `secondary_cause_death_code` : code de la cause secondaire de décès [chr]
- `tertiary_cause_death_description` : cause tertiaire de décès du patient [chr]
- `tertiary_cause_death_code` : code de la cause tertiaire de décès [chr]
- `quaternary_cause_death_description` : cause quaternaire de décès du patient [chr]
- `quaternary_cause_death_code` : code de la cause quaternaire de décès [chr]

Contributeurs

Les membres de l'équipe suivants ont contribué à cette leçon :



CAMILLE BEATRICE VALERA

Project Manager and Scientific Collaborator, The GRAPH Network



KENE DAVID NWOSU

Data analyst, the GRAPH Network

Passionate about world improvement

Joindre des tables de données

GRAPH Network & WHO, supported by the Global Fund to fight HIV, TB & Malaria

October 2023

This document is a draft of a lesson made by the GRAPH Network, a non-profit headquartered at the University of Geneva Global Health Institute, in collaboration with the World Health Organization, under a Global Fund 2023 grant to create e-learning modules to build in-country data capacity for epidemiological and impact analysis for National HIV, TB and malaria programs

Prélude
Objectifs d'apprentissage
Paquets
Qu'est-ce qu'une jointure et pourquoi en avons-nous besoin ?
Syntaxe des jointures
Types de jointures
left_join()
right_join()
inner_join()
full_join()
Résumé

Prélude

La jointure de bases de données est une compétence cruciale lorsqu'on travaille avec des données relatives à la santé car elle permet de combiner des informations provenant de plusieurs sources, conduisant à des analyses plus complètes et perspicaces. Dans cette leçon, vous apprendrez à utiliser différentes techniques de jointure à l'aide du package `dplyr` de R. Commençons !

Objectifs d'apprentissage

- Vous comprenez comment fonctionnent les différentes jointures de `dplyr`
- Vous êtes capable de choisir la jointure appropriée pour vos données
- Vous pouvez joindre des ensembles de données simples en utilisant des fonctions de `dplyr`

Paquets

Veuillez charger les paquets nécessaires à cette leçon avec le code ci-dessous :

```
# Charger les paquets
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse,
                countrycode)
```

Qu'est-ce qu'une jointure et pourquoi en avons-nous besoin ?

Pour illustrer l'utilité des jointures, commençons par un exemple de jouet. Considérez les deux ensembles de données suivants. Le premier, `démographique`, contient les noms et les âges de trois patients :

```
démographique <-
  tribble(~nom,      ~age,
          "Alice",    25,
          "Bob",     32,
          "Charlie", 45)
démographique
```

```
## # A tibble: 3 × 2
##   nom      age
##   <chr>    <dbl>
## 1 Alice     25
## 2 Bob       32
## 3 Charlie   45
```

Le deuxième, `info_test`, contient les dates et les résultats des tests de tuberculose pour ces patients :

```
info_test <-
  tribble(~nom,      ~date_du_test,      ~resultat,
          "Alice",    "2023-06-05",    "Négatif",
          "Bob",      "2023-08-10",    "Positif",
          "Charlie",  "2023-07-15",    "Négatif)
info_test
```

```
## # A tibble: 3 × 3
##   nom      date_du_test resultat
##   <chr>    <chr>        <chr>
## 1 Alice    2023-06-05  Négatif
## 2 Bob     2023-08-10  Positif
## 3 Charlie 2023-07-15  Négatif
```

Nous aimerais analyser ces données ensemble, et nous avons donc besoin d'une façon de les combiner.

Une option que nous pourrions envisager est la fonction `cbind()` de base R (`cbind` est l'abréviation de column bind) :

```
cbind(démographique, info_test)
```

nom	age	nom	date_du_test	resultat
Alice	25	Alice	2023-06-05	Négatif

nom	age	nom	date_du_test	resultat
Bob	32	Bob	2023-08-10	Positif
Charlie	45	Charlie	2023-07-15	Négatif

Cela fusionne avec succès les ensembles de données, mais il ne le fait pas très intelligemment. La fonction “colle” ou “agrafe” essentiellement les deux tables ensemble. Ainsi, comme vous pouvez le remarquer, la colonne “nom” apparaît deux fois. Ce n'est pas idéal et cela posera problème pour l'analyse.

Un autre problème se pose si les lignes des deux ensembles de données ne sont pas déjà alignées. Dans ce cas, les données seront combinées de manière incorrecte avec `cbind()`. Considérez l'ensemble de données `info_test_desordonne` ci-dessous, qui a maintenant Bob dans la première ligne :

```
info_test_desordonne <-
  tribble(~nom,      ~date_du_test,      ~resultat,
          "Bob",        "2023-08-10",    "Positif", # Bob in first row
          "Alice",       "2023-06-05",    "Négatif",
          "Charlie",     "2023-07-15",    "Négatif")
```

Qu'arrive-t-il si nous `cbind()` ceci avec l'ensemble de données `demographique` original, où Bob était dans la *deuxième* ligne ?

```
cbind(demographique, info_test_desordonne)
```

nom	age	nom	date_du_test	resultat
Alice	25	Bob	2023-08-10	Positif
Bob	32	Alice	2023-06-05	Négatif
Charlie	45	Charlie	2023-07-15	Négatif

Les détails démographiques d'Alice sont maintenant alignés par erreur avec les informations de test de Bob !

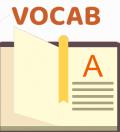
Un troisième problème se pose lorsqu'une entité apparaît plus d'une fois dans un ensemble de données. Peut-être qu'Alice a fait plusieurs tests de TB :

```
info_test_multiple <-
  tribble(~nom,      ~date_du_test,      ~resultat,
          "Alice",        "2023-06-05",    "Négatif",
          "Alice",        "2023-06-06",    "Négatif",
          "Bob",          "2023-08-10",    "Positif",
          "Charlie",      "2023-07-15",    "Négatif")
```

Si nous essayons de `cbind()` ceci avec l'ensemble de données `demographique`, nous obtiendrons une erreur, due à une incohérence dans le nombre de lignes :

```
cbind(demographique, info_test_multiple)
```

```
Erreur dans data.frame(..., check.names = FALSE) :  
  les arguments impliquent un nombre différent de lignes : 3, 4
```



VOCAB

Ce que nous avons ici est appelé une relation un-à-plusieurs—une Alice dans les données démographiques, mais plusieurs lignes Alice dans les données de test. La jointure dans de tels cas sera couverte en détail dans la deuxième leçon de jointure.

Il est évident que nous avons besoin d'une manière plus intelligente de combiner les jeux de données que `cbind()` ; nous devrons nous aventurer dans le monde des jointures.

Commençons par la jointure la plus courante, la `left_join()`, qui résout les problèmes auxquels nous avons été précédemment confrontés.

Elle fonctionne pour le cas simple, et elle ne duplique pas la colonne nom :

```
left_join(demographique, info_test)
```

```
## Joining with `by = join_by(nom)`  
  
## # A tibble: 3 × 4  
##   nom      age date_du_test resultat  
##   <chr>    <dbl> <chr>       <chr>  
## 1 Alice     25  2023-06-05  Négatif  
## 2 Bob       32  2023-08-10  Positif  
## 3 Charlie   45  2023-07-15  Négatif
```

Elle fonctionne lorsque les jeux de données ne sont pas ordonnés de la même manière :

```
left_join(demographique, info_test_desordonne)
```

```
## Joining with `by = join_by(nom)`  
  
## # A tibble: 3 × 4  
##   nom      age date_du_test resultat  
##   <chr>    <dbl> <chr>       <chr>  
## 1 Alice     25  2023-06-05  Négatif  
## 2 Bob       32  2023-08-10  Positif  
## 3 Charlie   45  2023-07-15  Négatif
```

Et elle fonctionne quand il y a plusieurs lignes de test par patient :

```
left_join(demographique, info_test_multiple)
```

```
## Joining with `by = join_by(nom)`

## # A tibble: 4 × 4
##   nom      age date_du_test resultat
##   <chr>    <dbl> <chr>       <chr>
## 1 Alice     25  2023-06-05 Négatif
## 2 Alice     25  2023-06-06 Négatif
## 3 Bob       32  2023-08-10 Positif
## 4 Charlie   45  2023-07-15 Négatif
```

Simple et magnifique !

::: note latérale

Nous utiliserons également l'opérateur pipe lors des jointures. Souvenez-vous que ceci :

```
demographique %>% left_join(info_test)
```

```
## Joining with `by = join_by(nom)`
```

est équivalent à ceci :

```
left_join(demographique, info_test)
```

```
## Joining with `by = join_by(nom)`
```

:::

Syntaxe des jointures

Maintenant que nous comprenons *pourquoi* nous avons besoin de jointures, regardons leur syntaxe de base.

Les jointures prennent deux dataframes comme deux premiers arguments : `x` (le dataframe *à gauche*) et `y` (le dataframe *à droite*). Comme pour les autres fonctions de R, vous pouvez fournir ces arguments avec ou sans nom :

```
# les deux sont identiques :
left_join(x = demographique, y = info_test) # nommé
left_join(demographique, info_test) # sans nom
```

Un autre argument crucial est `by`, qui indique la colonne ou **clé** utilisée pour connecter les tables. Nous n'avons pas toujours besoin de fournir cet argument ; il peut être *inféré* à partir des jeux de données. Par exemple, dans nos exemples originaux, "nom" est la seule colonne commune à `demographique` et `info_test`. Ainsi, la fonction de jointure suppose `by = "nom"` :

```
# ces deux sont équivalentes
left_join(x = demographique, y = info_test)
left_join(x = demographique, y = info_test, by = "nom")
```



VOCAB

La colonne utilisée pour connecter les lignes entre les tables est connue sous le nom de "clé". Dans les fonctions de jointure de `dplyr`, la clé est spécifiée dans l'argument `by`, comme on le voit dans `left_join(x = demographique, y = info_test, by = "nom")`

Que se passe-t-il si les clés sont nommées différemment dans les deux jeux de données ? Considérez le jeu de données `info_test_nom_different` ci-dessous, où la colonne "nom" a été modifiée en "destinataire_test" :

```
info_test_nom_different <-
  tribble(~destinataire_test, ~date_test, ~resultat,
          "Alice",      "2023-06-05",  "Négatif",
          "Bob",        "2023-08-10",  "Positif",
          "Charlie",    "2023-07-15",  "Négatif)
info_test_nom_different
```

```
## # A tibble: 3 × 3
##   destinataire_test date_test   resultat
##   <chr>              <chr>       <chr>
## 1 Alice              2023-06-05 Négatif
## 2 Bob                2023-08-10 Positif
## 3 Charlie            2023-07-15 Négatif
```

Si nous essayons de joindre `info_test_nom_different` à notre jeu de données `demographique` original, nous rencontrons une erreur :

```
left_join(x = demographique, y = info_test_nom_different)
```

```
Erreur dans `left_join()` :
! `by` doit être fourni lorsque `x` et `y` n'ont pas de
```

```
variables communes.  
i Utiliser `cross_join()` pour effectuer une jointure croisée.
```

L'erreur indique qu'il n'y a pas de variables communes, donc la jointure n'est pas possible.

Dans des situations comme celle-ci, vous avez deux choix : vous pouvez renommer la colonne dans le deuxième dataframe pour qu'elle corresponde à la première, ou plus simplement, spécifier sur quelles colonnes joindre en utilisant `by = c()`.

Voici comment faire cela :

```
left_join(x = demographique, y = info_test_nom_different,  
          by = c("nom" = "destinataire_test"))
```

```
## # A tibble: 3 × 4  
##   nom      age date_test  resultat  
##   <chr>    <dbl> <chr>     <chr>  
## 1 Alice      25 2023-06-05 Négatif  
## 2 Bob        32 2023-08-10 Positif  
## 3 Charlie    45 2023-07-15 Négatif
```

La syntaxe `c("nom" = "destinataire_test")` est un peu inhabituelle. Elle dit essentiellement, “Connecte `nom` du dataframe `x` avec `destinataire_test` du dataframe `y` parce qu'ils représentent les mêmes données.”

Considérez les deux ensembles de données ci-dessous, l'un avec les détails des patients et l'autre avec les dates de contrôle médical pour ces patients.

```
patients <- tribble(  
  ~id_patient, ~nom,      ~age,  
  1,           "Jean",    32,  
  2,           "Joie",    28,  
  3,           "Khan",    40  
)  
  
controles <- tribble(  
  ~id_patient, ~date_controle,  
  1,           "2023-01-20",  
  2,           "2023-02-20",  
  3,           "2023-05-15"  
)
```

Joignez l'ensemble de données `patients` avec l'ensemble de données `controles` en utilisant `left_join()`

Deux ensembles de données sont définis ci-dessous, l'un avec les détails des patients et l'autre avec les registres de vaccination pour ces patients.

```

# Détails des patients
details_patient <- tribble(
  ~numero_id,    ~nom_complet,    ~adresse,
  "A001",        "Alice",        "123 Elm St",
  "B002",        "Bob",          "456 Maple Dr",
  "C003",        "Charlie",      "789 Oak Blvd"
)

# Registres de vaccination
registres_vaccination <- tribble(
  ~code_patient, ~type_vaccin,   ~date_vaccination,
  "A001",         "COVID-19",     "2022-05-10",
  "B002",         "Grippe",       "2023-09-01",
  "C003",         "Hépatite B",   "2021-12-15"
)

```

Joignez les ensembles de données `details_patient` et `registres_vaccination`. Vous devrez utiliser l'argument `by` car les colonnes identifiant le patient ont des noms différents.

Types de jointures

Les exemples jouets jusqu'à présent ont impliqué des ensembles de données qui pouvaient être parfaitement correspondants - chaque ligne dans un ensemble de données avait une ligne correspondante dans l'autre ensemble de données.

Les données du monde réel sont généralement plus désordonnées. Souvent, il y aura des entrées dans la première table qui n'ont pas d'entrées correspondantes dans la deuxième table, et vice versa.

Pour gérer ces cas de correspondance imparfaite, il existe différents types de jointures avec des comportements spécifiques : `left_join()`, `right_join()`, `inner_join()` et `full_join()`. Dans les sections à venir, nous examinerons des exemples de la manière dont chaque type de jointure opère sur des ensembles de données avec des correspondances imparfaites.

`left_join()`

Commençons par `left_join()`, que vous avez déjà rencontré. Pour voir comment il gère les lignes non appariées, nous allons essayer de joindre notre ensemble de données `demographique` original avec une version modifiée de l'ensemble de données `infos_test`.

Pour rappel, voici l'ensemble de données `demographique`, avec Alice, Bob et Charlie :

```
demographique
```

```
## # A tibble: 3 × 2
##   nom      age
##   <chr>    <dbl>
## 1 Alice     25
## 2 Bob       32
## 3 Charlie   45
```

Pour les informations de test, nous allons supprimer Charlie et nous allons ajouter un nouveau patient, Xavier, et ses données de test :

```
infos_test_xavier <- tribble(
  ~nom,      ~date_test, ~resultat,
  "Alice",   "2023-06-05", "Négatif",
  "Bob",     "2023-08-10", "Positif",
  "Xavier",  "2023-05-02", "Négatif")
infos_test_xavier
```

```
## # A tibble: 3 × 3
##   nom      date_test resultat
##   <chr>    <chr>     <chr>
## 1 Alice    2023-06-05 Négatif
## 2 Bob      2023-08-10 Positif
## 3 Xavier   2023-05-02 Négatif
```

Si nous effectuons un `left_join()` en utilisant `demographique` comme ensemble de données de gauche (`x = demographique`) et `infos_test_xavier` comme ensemble de données de droite (`y = infos_test_xavier`), à quoi devrions-nous nous attendre ? Rappelons que Charlie n'est présent que dans l'ensemble de données de gauche, et Xavier n'est présent que dans celui de droite. Eh bien, voici ce qui se passe :

```
left_join(x = demographique, y = infos_test_xavier, by = "nom")
```

Comme vous pouvez le voir, avec la jointure *LEFT*, tous les enregistrements du dataframe *LEFT*(`demographique`) sont conservés. Donc, même si Charlie n'a pas de correspondance dans l'ensemble de données `infos_test_xavier`, il est toujours inclus dans la sortie. (Mais bien sûr, comme ses informations de test ne sont pas disponibles dans `infos_test_xavier`, ces valeurs ont été laissées à NA.)

Xavier, en revanche, qui n'était présent que dans l'ensemble de données de droite, est supprimé.

Le graphique ci-dessous montre comment cette jointure a fonctionné :



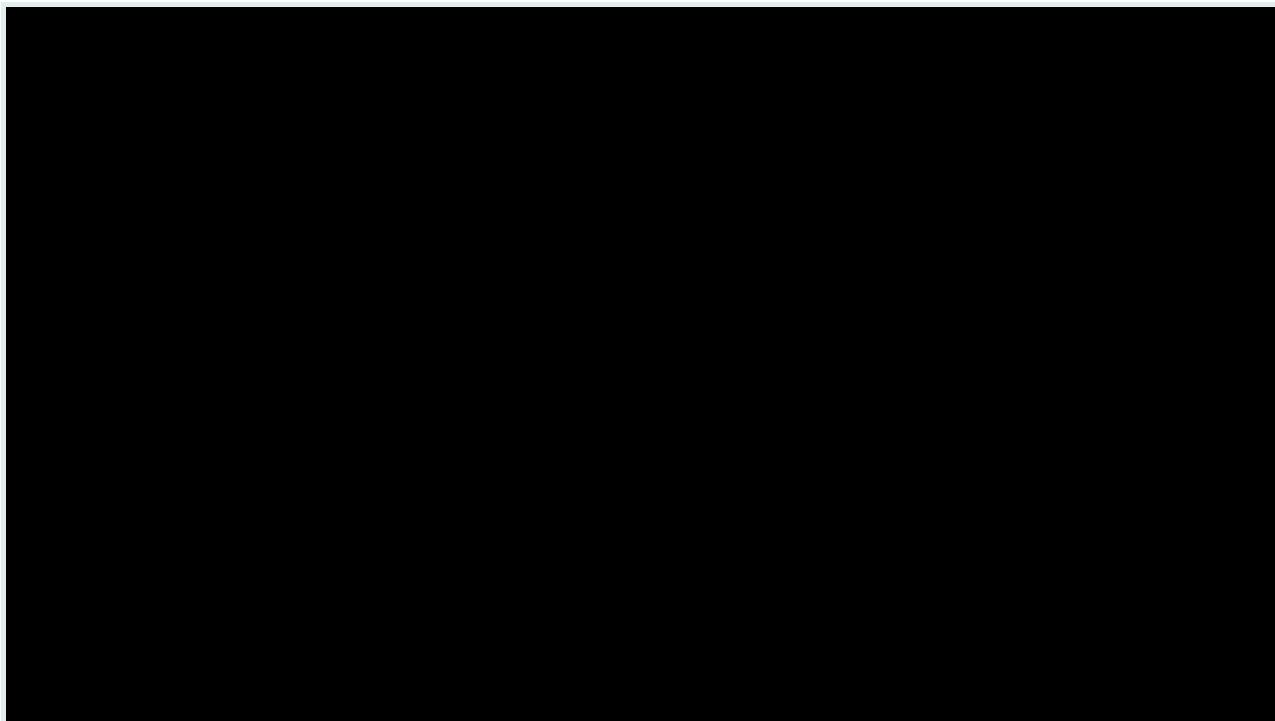
Dans une fonction de jointure telle que `left_join(x, y)`, l'ensemble de données fourni à l'argument `x` peut être appelé l'ensemble de données "de gauche", tandis que l'ensemble de données attribué à l'argument `y` peut être appelé l'ensemble de données "de droite".

Et si nous inversions les ensembles de données ? Voyons le résultat lorsque `infos_test_xavier` est l'ensemble de données de gauche et `demographique` celui de droite :

```
left_join(x = infos_test_xavier, y = demographique, by = "nom")
```

Encore une fois, `left_join()` conserve toutes les lignes de l'ensemble de données *de gauche* (maintenant `infos_test_xavier`). Cela signifie que les données de Xavier sont incluses cette fois. Charlie, en revanche, est exclu.

Le graphique ci-dessous illustre comment cela fonctionne:



Ensemble de données principal : Dans le contexte des jointures, l'ensemble de données principal désigne l'ensemble de données principal ou priorisé dans une opération. Dans une jointure à gauche, l'ensemble de données de gauche est considéré comme l'ensemble de données principal car toutes ses lignes sont conservées dans le résultat, qu'elles aient ou non une ligne correspondante dans l'autre ensemble de données.

Essayez ce qui suit. Voici deux ensembles de données - l'un avec des diagnostics de maladie (`dx_maladie`) et un autre avec des données démographiques de patients (`demographique_patient`).

```
dx_maladie <- tribble(
  ~id_patient, ~maladie,           ~date_diagnostic,
  1,          "Influenza",        "2023-01-15",
  3,          "COVID-19",         "2023-03-05",
  8,          "Influenza",        "2023-02-20",
)

demographique_patient <- tribble(
  ~id_patient, ~nom,           ~age,   ~genre,
  1,           "Fred",          28,    "Femme",
  2,           "Genevieve",     45,    "Femme",
  3,           "Henry",         32,    "Homme",
  5,           "Irene",          55,    "Femme",
  8,           "Jules",          40,    "Homme"
)
```

Utilisez `left_join()` pour fusionner ces ensembles de données, en ne conservant que les patients pour lesquels nous avons des informations démographiques. Réfléchissez bien à quel ensemble de données mettre à gauche.

Essayons un autre exemple, cette fois avec un ensemble de données plus réaliste.

Premièrement, nous avons des données sur le taux d'incidence de la tuberculose par 100 000 personnes pour 47 pays africains, de l'[OMS](#) :

```
tb_2019_afrique <- read_csv(here("data/tb_incidence_2019.csv"))
```

```
## Rows: 47 Columns: 3
## — Column specification —
## Delimiter: ","
## chr (2): country, conf_int_95
## dbl (1): cases
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this
message.
```

```
tb_2019_afrique
```

Nous voulons analyser comment l'incidence de la TB dans les pays africains varie avec les dépenses de santé par habitant du gouvernement. Pour cela, nous avons des données sur les dépenses de santé par habitant en USD, également de l'[OMS](#) :

```
dep_sante_2019 <- read_csv(here("data/health_expend_per_cap_2019.csv"))
```

```
## Rows: 185 Columns: 2
## — Column specification —
## Delimiter: ","
## chr (1): country
## dbl (1): expend_usd
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this
message.
```

```
dep_sante_2019
```

Quel ensemble de données devrions-nous utiliser comme dataframe de gauche pour la jointure ?

Comme notre objectif est d'analyser les pays africains, nous devrions utiliser `tb_2019_afrique` comme dataframe de gauche. Cela garantira que nous gardons tous

les pays africains dans l'ensemble de données joint final.

Faisons la jointure :

```
tb_dep_sante_joint <-
  tb_2019_afrique %>%
  left_join(dep_sante_2019, by = "country")
tb_dep_sante_joint
```

Maintenant, dans l'ensemble de données joint, nous avons juste les 47 lignes pour les pays africains, ce qui est exactement ce que nous voulions !

Toutes les lignes du dataframe de gauche `tb_2019_afrique` ont été conservées, tandis que les pays non africains de `dep_sante_2019` ont été écartés.

Nous pouvons vérifier si certaines lignes de `tb_2019_afrique` n'ont pas eu de correspondance dans `dep_sante_2019` en filtrant pour les valeurs NA :

```
tb_dep_sante_joint %>%
  filter(is.na(!expend_usd))
```

```
## # A tibble: 3 × 4
##   country      cases conf_int_95 expend_usd
##   <chr>        <dbl> <chr>          <dbl>
## 1 Mauritius     12 [9 - 15]       NA
## 2 South Sudan   227 [147 - 324]    NA
## 3 Comoros       35 [23 - 50]       NA
```

Cela montre que 3 pays - Maurice, le Soudan du Sud et les Comores - n'avaient pas de données sur les dépenses dans `dep_sante_2019`. Mais comme ils étaient présents dans `tb_2019_afrique`, et que c'était le dataframe de gauche, ils ont quand même été inclus dans les données jointes.

Pour en être sûr, nous pouvons rapidement confirmer que ces pays sont absents de l'ensemble de données sur les dépenses avec une déclaration de filtre :

```
dep_sante_2019 %>%
  filter(country %in% c("Mauritius", "South Sudan", "Comoros"))
```

```
## # A tibble: 0 × 2
## # i 2 variables: country <chr>, expend_usd <dbl>
```

En effet, ces pays ne sont pas présents dans `dep_sante_2019`.

Copiez le code ci-dessous pour définir deux ensembles de données.

Le premier, `cas_tb_enfants` contient le nombre de cas de TB chez les moins de 15 ans en 2012, par pays :

```

cas_tb_enfants <- tidyverse::who %>%
  filter(year == 2012) %>%
  transmute(country, cas_tb_smear_0_14 = new_sp_m014 + new_sp_f014)

cas_tb_enfants

```

```

## # A tibble: 5 × 2
##   country      cas_tb_smear_0_14
##   <chr>          <dbl>
## 1 Afghanistan     588
## 2 Albania            0
## 3 Algeria           89
## 4 American Samoa    NA
## 5 Andorra            0

```

Et `pays_continents`, du package `{countrycode}`, liste tous les pays et leur région et continent correspondants :

```

pays_continents <-
  countrycode::codelist %>%
  select(country.name.fr, continent, region)

pays_continents

```

```

## # A tibble: 5 × 3
##   country.name.fr   continent region
##   <chr>             <chr>     <chr>
## 1 Afghanistan       Asia      South Asia
## 2 Albanie           Europe    Europe & Central Asia
## 3 Algérie            Africa    Middle East & North Africa
## 4 Samoa américaines Oceania  East Asia & Pacific
## 5 Andorre            Europe    Europe & Central Asia

```

Votre objectif est d'ajouter les données de continent et de région à l'ensemble de données sur les cas de TB.

Quel ensemble de données devrait être le dataframe de gauche, `x`? Et lequel devrait être le droit, `y`? Une fois que vous avez décidé, joignez les ensembles de données de manière appropriée en utilisant `left_join()`.

`right_join()`

Un `right_join()` peut être considéré comme une image miroir d'un `left_join()`. Les mécanismes sont les mêmes, mais maintenant toutes les lignes de l'ensemble de données de *DROITE* sont conservées, tandis que seules les lignes de l'ensemble de données de gauche qui trouvent une correspondance à droite sont conservées.

Regardons un exemple pour comprendre cela. Nous utiliserons nos ensembles de données `demographique` et `infos_test_xavier` originaux :

```
demographique
```

```
infos_test_xavier
```

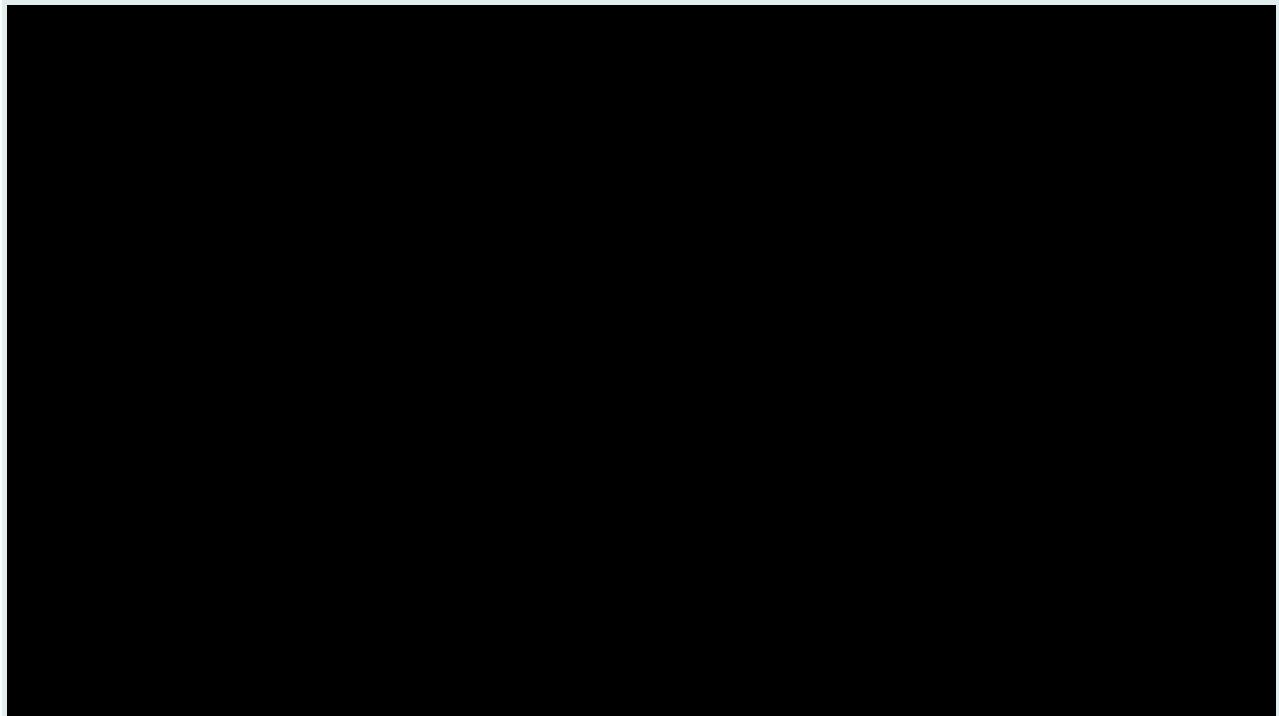
Essayons maintenant `right_join()`, avec `demographique` comme dataframe de droite :

```
right_join(x = infos_test_xavier, y = demographique)
```

```
## Joining with `by = join_by(nom)`
```

J'espère que vous commencez à comprendre cela, et que vous pourriez prédire cette sortie ! Puisque `demographique` était le dataframe de *droite*, et que nous utilisons *right-join*, toutes les lignes de `demographique` sont conservées—Alice, Bob et Charlie. Mais seulement les enregistrements correspondants dans le dataframe de gauche `infos_test_xavier` !

Le graphique ci-dessous illustre ce processus :



Un point important—le même dataframe final peut être créé avec `left_join()` ou `right_join()`, cela dépend simplement de l'ordre dans lequel vous fournissez les dataframes à ces fonctions :

```
# ici, RIGHT_join privilégie le df de DROITE, demographique
right_join(x = infos_test_xavier, y = demographique)
```

```
## Joining with `by = join_by(nom)`  
  
# ici, LEFT_join privilégie le df de GAUCHE, encore une fois démographique  
left_join(x = demographique, y = infos_test_xavier)
```

```
## Joining with `by = join_by(nom)`
```

La seule différence que vous pourriez remarquer entre left et right-join est que l'ordre final des colonnes est différent. Mais les colonnes peuvent facilement être réarrangées, donc se soucier de l'ordre des colonnes n'en vaut vraiment pas la peine.

Comme nous l'avons mentionné précédemment, les data scientists favorisent généralement `left_join()` par rapport à `right_join()`. Il est plus logique de spécifier votre ensemble de données principal d'abord, dans la position de gauche. Opter pour un `left_join()` est une bonne pratique courante en raison de sa logique plus claire, ce qui le rend moins sujet à l'erreur.

Super, maintenant nous comprenons comment fonctionnent `left_join()` et `right_join()`, passons à `inner_join()` et `full_join()` !

`inner_join()`

Ce qui distingue un `inner_join`, c'est que les lignes ne sont conservées que si les valeurs de jointure sont présentes dans *les deux* dataframes. Revenons à notre exemple de patients et de leurs résultats de test COVID. Pour rappel, voici nos ensembles de données :

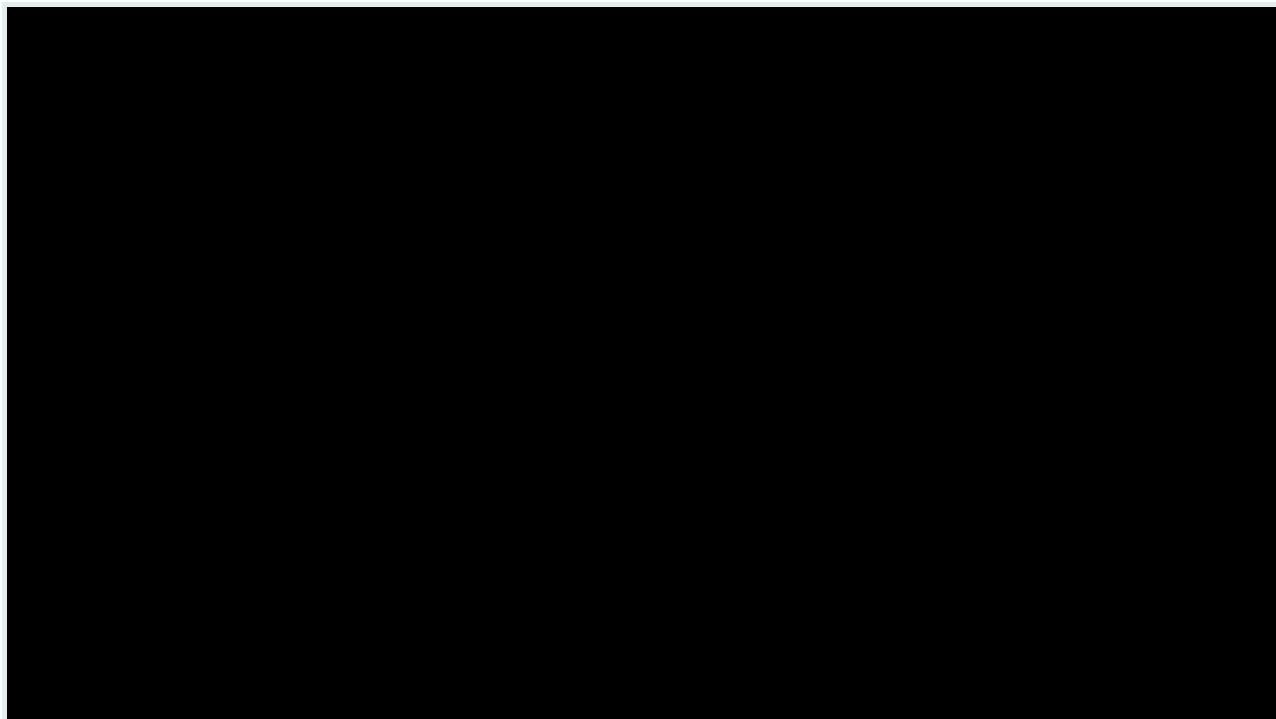
```
demographique
```

```
infos_test_xavier
```

Maintenant que nous avons une meilleure compréhension de la façon dont fonctionnent les jointures, nous pouvons déjà imaginer à quoi ressemblerait le dataframe final si nous utilisions un `inner_join()` sur nos deux dataframes ci-dessus. Si seules les lignes avec des valeurs de jointure qui sont dans *les deux* dataframes sont conservées, et que les seuls patients qui sont à la fois dans `demographique` et `infos_test` sont Alice et Bob, alors ils devraient être les seuls patients dans notre ensemble de données final ! Essayons.

```
inner_join(demographique, infos_test_xavier, by="nom")
```

Parfait, c'est exactement ce à quoi nous nous attendions ! Ici, Charlie était seulement dans l'ensemble de données `demographique`, et Xavier était seulement dans l'ensemble de données `infos_test`, donc tous deux ont été supprimés. Le graphique ci-dessous montre comment fonctionne cette jointure :



Il est logique que l'ordre dans lequel vous spécifiez vos ensembles de données ne change pas les informations qui sont conservées, étant donné que vous avez besoin de valeurs de jointure dans les deux ensembles de données pour qu'une ligne soit conservée. Pour illustrer cela, essayons de changer l'ordre de nos ensembles de données.

```
inner_join(infos_test_xavier, demographique, by="nom")
```

Comme prévu, la seule différence ici est l'ordre de nos colonnes, sinon les informations conservées sont les mêmes. ::: r-pratique

Les données suivantes concernent les épidémies d'origine alimentaire aux États-Unis en 2019, provenant du [CDC](#). Copiez le code ci-dessous pour créer deux nouveaux dataframes :

```
total_inf <- tribble(
  ~pathogene,          ~total_infections,
  "Campylobacter",   9751,
  "Listeria",         136,
  "Salmonella",       8285,
  "Shigella",          2478,
)

resultats <- tribble(
  ~pathogene,          ~n_hosp,      ~n_deces,
  "Listeria",           128,          30,
  "STEC",                582,          11,
  "Campylobacter",     1938,          42,
  "Yersinia",              200,            5,
)
```

Quels sont les pathogènes communs entre les deux ensembles de données ? Utilisez un `inner_join()` pour joindre les dataframes, afin de ne conserver que les pathogènes qui figurent dans les deux ensembles de données.

:::

Revenons à nos données sur les dépenses de santé et le niveau de revenu et appliquons ce que nous avons appris à ces ensembles de données.

```
tb_2019_afrique
```

```
dep_sante_2019
```

Ici, nous pouvons créer un nouveau dataframe appelé `inner_dep_tb` en utilisant un `inner_join()` pour ne conserver que les pays pour lesquels nous avons à la fois des données sur les dépenses de santé et les taux d'incidence de la tuberculose. Essayons-le maintenant :

```
inner_dep_tb <- tb_2019_afrique %>%
  inner_join(dep_sante_2019)
```

```
## Joining with `by = join_by(country)`
```

```
inner_dep_tb
```

Cela semble très bien ! Avec `left_join()`, le `inner_join()` est l'une des jointures les plus courantes lorsqu'on travaille avec des données, il est donc probable que vous y serez souvent confronté. C'est un outil puissant et souvent utilisé, mais c'est aussi la jointure qui exclut le plus d'informations, alors assurez-vous que vous ne voulez que des enregistrements correspondants dans votre ensemble de données final ou vous risquez de perdre accidentellement beaucoup de données ! En revanche, `full_join()` est la jointure la plus inclusive, regardons-la dans la prochaine section.

Utilisez un `inner_join()` pour joindre les dépenses de santé 2019 et . Quels pays restent dans votre ensemble de données final ?

`full_join()`

La particularité de `full_join()` est qu'il conserve *tous* les enregistrements, qu'il y ait ou non une correspondance entre les deux jeux de données. Lorsqu'il manque des informations dans notre jeu de données final, les cellules sont définies sur `NA` comme nous l'avons vu dans `left_join()` et `right_join()`.

Jetons un coup d'œil à nos jeux de données `Demographique` et `test_info` pour illustrer cela.

Voici un rappel de nos données :

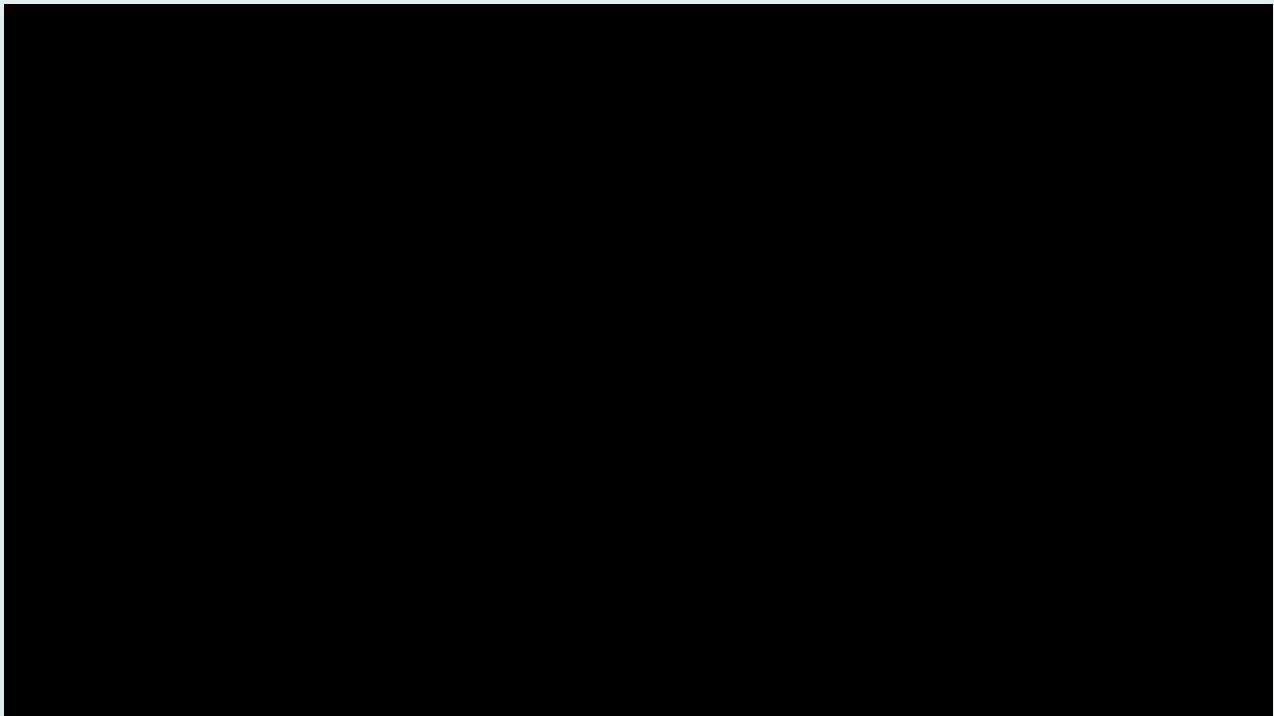
```
demographique
```

```
infos_test_xavier
```

Maintenant, effectuons un `full_join`, avec `Demographique` comme nos données principal.

```
full_join(demographique, infos_test_xavier, by="nom")
```

Comme nous pouvons le voir, toutes les lignes ont été conservées donc il n'y a eu aucune perte d'information ! Le graphique ci-dessous illustre ce processus :



Comme cette jointure n'est pas sélective, tout se retrouve dans l'ensemble de données final, donc changer l'ordre de nos ensembles de données ne changera pas les informations qui sont conservées. Cela ne changera que l'ordre des colonnes dans notre ensemble de données final. Nous pouvons le voir ci-dessous lorsque nous spécifions `test_info` (traduit par `info_test`) comme notre ensemble de données principal et `Demographic` (traduit par `Démographique`) comme notre ensemble de données secondaire.

```
full_join(infos_test_xavier, demographique, by="nom")
```

Comme nous l'avons vu ci-dessus, toutes les données des deux ensembles de données d'origine sont toujours là, avec toute information manquante définie à NA. ::: r-pratique Les dataframes suivantes contiennent les taux d'incidence mondiaux du paludisme par 100'000 personnes et les taux de mortalité mondiaux par 100'000 personnes dus au paludisme, provenant de [Our World in Data](#). Copiez le code pour créer deux petits dataframes :

```

inc_paludisme <- tribble(
  ~année, ~inc_100k,
  2010, 69.485344,
  2011, 66.507935,
  2014, 59.831020,
  2016, 58.704540,
  2017, 59.151703,
)

deces_paludisme <- tribble(
  ~année, ~deces_100k,
  2011, 12.92,
  2013, 11.00,
  2015, 10.11,
  2016, 9.40,
  2019, 8.95
)

```

Ensuite, joignez les tables ci-dessus en utilisant un `full_join()` afin de conserver toutes les informations des deux ensembles de données.

:::

Revenons à notre ensemble de données sur la tuberculose et notre ensemble de données sur les dépenses de santé.

```
tb_2019_afrique
```

```

## # A tibble: 5 × 3
##   country           cases conf_int_95
##   <chr>            <dbl> <chr>
## 1 Burundi          107  [69 - 153]
## 2 Sao Tome and Principe 114  [45 - 214]
## 3 Senegal          117  [83 - 156]
## 4 Mauritius        12   [9 - 15]
## 5 Côte d'Ivoire    137  [88 - 197]

```

```
dep_sante_2019
```

```

## # A tibble: 5 × 2
##   country       expend_usd
##   <chr>            <dbl>
## 1 Nigeria          11.0
## 2 Bahamas          1002
## 3 United Arab Emirates 1015
## 4 Nauru             1038
## 5 Slovakia         1058

```

Maintenant, créons un nouveau dataframe appelé `full_tb_sante` en utilisant un `full_join` !

```
inner_dep_tb <- tb_2019_afrique %>%
  inner_join(dep_sante_2019)
```

```
## Joining with `by = join_by(country)`
```

```
inner_dep_tb
```

Comme nous l'avons vu précédemment, toutes les lignes ont été conservées entre les deux ensembles de données avec des valeurs manquantes définies à `NA`.

Comme pour l'exercice précédent, obtenez tous les pays et leur région et continent correspondants à partir du package `{countrycode}` :

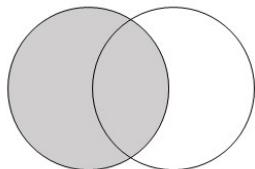
```
pays_continents <-
  countrycode::codelist %>%
  select(country.name.fr, continent, region)
```

Ensuite, utilisez un `full_join()` pour joindre avec l'ensemble de données `tb_2019_afrique`.

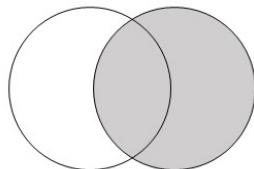
Résumé

Bravo, vous comprenez maintenant les bases de la jointure ! Le diagramme de Venn ci-dessous donne un résumé utile des différentes jointures et des informations que chacune conserve. Il peut être utile de sauvegarder cette image pour référence future !

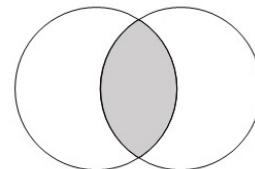
`left_join()`



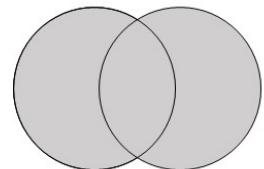
`right_join()`



`inner_join()`



`full_join()`



Joindre des tables de données (leçon 2)

GRAPH Network & WHO, supported by the Global Fund to fight HIV, TB & Malaria

October 2023

This document is a draft of a lesson made by the GRAPH Network, a non-profit headquartered at the University of Geneva Global Health Institute, in collaboration with the World Health Organization, under a Global Fund 2023 grant to create e-learning modules to build in-country data capacity for epidemiological and impact analysis for National HIV, TB and malaria programs

Introduction
Objectifs d'apprentissage
Packages
Nettoyage préalable des données
Relations un-à-plusieurs
<code>left_join()</code>
<code>inner_join()</code>
Colonnes clés multiples

Introduction

Maintenant que nous maîtrisons bien les différents types de jointures et leur fonctionnement, nous pouvons voir comment gérer des ensembles de données plus complexes et désordonnés. La jointure de données réelles issues de sources différentes nécessite souvent réflexion et nettoyage préalables.

Objectifs d'apprentissage

- Vous savez comment vérifier les valeurs discordantes entre des jeux de données
- Vous comprenez comment effectuer une jointure de type un-à-plusieurs
- Vous savez comment effectuer une jointure sur plusieurs colonnes clés

Packages

Veuillez charger les packages nécessaires pour cette leçon avec le code ci-dessous :

```
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse)
```

Nettoyage préalable des données

Il est souvent nécessaire de nettoyer préalablement vos données lorsque vous les extraire de différentes sources avant de pouvoir les joindre. Cela est dû au fait qu'il peut y avoir des différences dans la manière dont les valeurs sont écrits dans les différentes tables, comme des erreurs d'orthographe, des différences de casse, ou des espaces en trop. Pour

joindre les valeurs, elles doivent correspondre parfaitement. Si des différences existent, R les considère comme des valeurs distinctes.

Pour illustrer ceci, reprenons nos données fictives de patient du premier cours. Vous vous souvenez probablement que nous avions deux dataframes, un appelé `demographique` et l'autre `info_test`. Nous pouvons recréer ces jeux de données mais changer `Alice` en `alice` dans le dataframe `demographique` tout en gardant les autres valeurs identiques.

```
demographique <- tribble(  
  ~nom,      ~age,  
  "alice",    25,  
  "Bob",     32,  
  "Charlie", 45,  
)  
demographique
```

```
info_test <- tribble(  
  ~nom,      ~date_test,      ~resultat,  
  "Alice",   "2023-06-05",   "Negatif",  
  "Bob",     "2023-08-10",   "Positif",  
  "Xavier",  "2023-05-02",   "Negatif",  
)  
info_test
```

Essayons maintenant une jointure interne `inner_join()` sur nos deux jeux de données.

```
inner_join(demographique, info_test, by="nom")
```

Comme nous pouvons le voir, R n'a pas reconnu `Alice` et `alice` comme étant la même personne, donc la seule valeur commune entre les jeux de données était `Bob`. Comment pouvons-nous gérer cela ? Eh bien, il existe plusieurs fonctions que nous pouvons utiliser pour modifier nos chaînes de caractères. Dans ce cas, utiliser `str_to_title()` fonctionnerait pour s'assurer que toutes les valeurs soient identiques. Si nous appliquons cette fonction à notre colonne `nom` dans notre dataframe `demographique`, nous pourrons joindre correctement les tables.

```
demographique <- demographique %>%  
  mutate(nom = str_to_title(nom))  
demographique
```

```
inner_join(demographique, info_test, by="nom")
```

Cela a parfaitement fonctionné ! Nous ne rentrerons pas dans les détails de toutes les différentes fonctions que nous pouvons utiliser pour modifier les chaînes de caractères, puisqu'elles sont couvertes de manière exhaustive dans la leçon sur les chaînes de caractères. L'élément important de cette leçon est que nous allons apprendre à identifier les valeurs discordantes entre nos dataframes.

Les deux jeux de données suivants contiennent des données pour l'Inde, l'Indonésie et les Philippines. Quelles sont les différences entre les valeurs dans les colonnes clés qui devraient être modifiées avant de joindre les jeux de données ?

```

df1 <- tribble(
  ~Pays,          ~Capitale,
  "Inde",        "New Delhi",
  "Indonésie",   "Jakarta",
  "Philippines", "Manille"
)

df2 <- tribble(
  ~Pays,      ~Population,    ~Esperance_de_vie,
  "Inde ",    1393000000,    69.7,
  "indonésie", 273500000,    71.7,
  "Philippines", 113000000,  72.7
)

```

Dans de petits jeux de données comme nos données fictives ci-dessus, il est assez facile de repérer les différences entre les valeurs dans nos colonnes clés. Mais qu'en est-il quand on a un plus grand jeu de données ? Illustrons cela avec deux jeux de données réels sur la tuberculose en Inde.

Notre premier jeu de données contient des données sur la notification des cas de tuberculose en 2022 pour tous les états et territoires de l'Union indienne, issues du [Rapport gouvernemental sur la tuberculose en Inde](#). Nos variables comprennent le nom de l'état/territoire de l'Union, le type de système de santé dans lequel les patients ont été détectés (public ou privé), le nombre cible de patients dont le statut de tuberculose devait être notifié, et le nombre réel de patients atteints de tuberculose dont le statut a été notifié.

```
notification <- read_csv(here("data/notification_TB_Inde.csv"))
```

```
notification
```

```

## # A tibble: 5 × 4
##   Etat                  systeme_sante cible_notifiee
##   <chr>                 <chr>           <dbl>
## 1 Iles Andaman et Nicobar public            520
## 2 Iles Andaman et Nicobar privé             10
## 3 Andhra Pradesh       public            85000
## 4 Andhra Pradesh       privé            30000
## 5 Arunachal Pradesh   public            3450
##   notifiee_relle      <dbl>
##   1                   510
##   2                   24
##   3                 62075
##   4                 30112
##   5                 2722

```

Notre second jeu de données, également issu du même [Rapport sur la tuberculose](#), contient le nom de l'état/territoire de l'Union, le type de système de santé, le nombre de

patients atteints de tuberculose dépistés pour le COVID-19, et le nombre de patients atteints de tuberculose diagnostiqués positifs au COVID-19.

```
covid <- read_csv(here("data/COVID_TB_Inde.csv"))
```

```
covid
```

```
## # A tibble: 5 × 4
##   Etat           systeme_sante covid_test
##   <chr>          <chr>            <dbl>
## 1 Iles Andaman et Nicobar public        322
## 2 Iles Andaman et Nicobar privé         1
## 3 Andhra Pradesh    public       63319
## 4 Andhra Pradesh    privé        26410
## 5 Arunachal Pradesh public        1761
## # covid_diagnosique
## # <dbl>
## 1 0
## 2 0
## 3 97
## 4 17
## 5 0
```

Pour les besoins de cette leçon, nous avons modifié certains des noms d'états/territoires de l'Union dans le jeu de données `covid`. Notre objectif est de les faire correspondre aux noms du jeu de données `notification` afin de pouvoir les joindre. Pour cela, nous devons comparer les valeurs entre eux. Pour de grands jeux de données, si nous souhaitons comparer quelles valeurs sont présentes dans l'un mais pas dans l'autre, nous pouvons utiliser la fonction `setdiff()` en précisant quels dataframes et colonnes nous souhaitons comparer. Commençons par comparer les valeurs de la colonne `state_UT` du dataframe `notification` à celles de la colonne `state_UT` du dataframe `covid`.

```
setdiff(notification$Etat, covid$Etat)
```

```
## [1] "Arunachal Pradesh"
## [2] "Dadra et Nagar Haveli et Daman et Diu"
## [3] "Tamil Nadu"
## [4] "Tripura"
```

Que nous indique cette liste ? En plaçant le jeu de données `notification` en premier, nous demandons à R “quelles valeurs sont présentes dans `notification` mais PAS dans `covid` ?”. Nous pouvons (et devrions !) également inverser l'ordre des jeux de données pour vérifier dans l'autre sens, en demandant “quelles valeurs sont présentes dans `covid` mais PAS dans `notification` ?” Faisons cela et comparons les deux listes.

```
setdiff(covid$Etat, notification$Etat)
```

```
## [1] "ArunachalPradesh"  
## [2] "Dadra & Nagar Haveli & Daman & Diu"  
## [3] "tamil nadu"  
## [4] "Tri pura"
```

Comme nous pouvons le voir, il y a quatre valeurs dans le jeu de données `covid` qui présentent des erreurs d'orthographe ou qui sont écrites de manière différente comparé au jeu de données `notification`. Dans ce cas, la solution la plus simple serait de nettoyer les données de `covid` en utilisant la fonction `case_when()` afin de faire correspondre les deux jeux de données. Nettoyons cela et comparons nos jeux de données à nouveau.

```
covid <- covid %>%  
  mutate(Etat =  
    case_when(Etat == "ArunachalPradesh" ~ "Arunachal Pradesh",  
              Etat == "tamil nadu" ~ "Tamil Nadu",  
              Etat == "Tri pura" ~ "Tripura",  
              Etat == "Dadra & Nagar Haveli & Daman & Diu" ~ "Dadra et  
Nagar Haveli et Daman et Diu",  
              TRUE ~ Etat))  
  
setdiff(notification$Etat, covid$Etat)
```

```
## character(0)
```

```
setdiff(covid$Etat, notification$Etat)
```

```
## character(0)
```



REMINDER À des fins d'illustration, nous avons réécrit les valeurs d'origine de notre jeu de données `covid`. Cependant, dans la pratique, lorsque vous transformez vos variables, il vaut toujours mieux créer une nouvelle variable propre et supprimer les anciennes si vous ne les utilisez plus !

Super ! Comme nous pouvons le voir, il n'y a plus de différences dans les valeurs entre nos jeux de données. Maintenant que nous nous sommes assurés que nos données sont propres, nous pouvons passer à la jointure ! Puisque nous comprenons les bases de la jointure grâce à notre premier cours, nous pouvons aborder des sujets plus complexes.

Le jeu de données suivant, également extrait du [Rapport sur la tuberculose](#), contient des informations sur le nombre de cas de tuberculose pédiatrique et sur le nombre de patients pédiatriques initiés au traitement.

```
enfant <- read_csv(here("data/enfant_TB_Inde.csv"))
```

```
enfant
```

```
## # A tibble: 5 × 4
##   Etat           systeme_sante enfant_notifie
##   <chr>          <chr>                  <dbl>
## 1 Iles Andaman et Nicobar public            18
## 2 Iles Andaman et Nicobar privé             1
## 3 Andhra Pradesh    public            1347
## 4 Andhra Pradesh    privé             1333
## 5 Arunachal Pradesh public            256
## #> enfant_traitement
## #>   <dbl>
## #> 1      19
## #> 2      0
## #> 3    1684
## #> 4    993
## #> 5    282
```

En utilisant la fonction `set_diff()`, comparez les valeurs de jointure du jeu de données `enfant` avec celles du jeu de données `notification` et apportez les modifications nécessaires au jeu de données `enfant` pour que les valeurs correspondent.

Relations un-à-plusieurs

Dans le cours précédent, nous nous sommes intéressés aux jointures un-à-un, où une observation dans un jeu de données correspondait à au maximum une observation dans l'autre jeu de données. Dans une jointure un-à-plusieurs, une observation dans un jeu de données correspond à plusieurs observations dans l'autre jeu de données. L'image ci-dessous illustre ce concept:

Jeux de données 1

Patient_num	Sexe
1001	M
1002	F
1003	M

Jeux de données 2

Patient_num	Temps	Diastolique
1001	pre	92
1001	post	78
1002	pre	96
1002	post	75
1003	pre	89
1003	post	81

Examinons une jointure un-à-plusieurs avec une jointure de type `left_join()`!

```
left_join()
```

Pour illustrer une jointure un-à-plusieurs, reprenons nos données de patients et leurs résultats de tests COVID. Imaginons que dans notre jeu de données, Alice et Xavier se soient fait tester plusieurs fois pour le COVID. Nous pouvons ajouter deux lignes supplémentaires à notre jeu de données `info_test` avec leurs nouvelles informations de test:

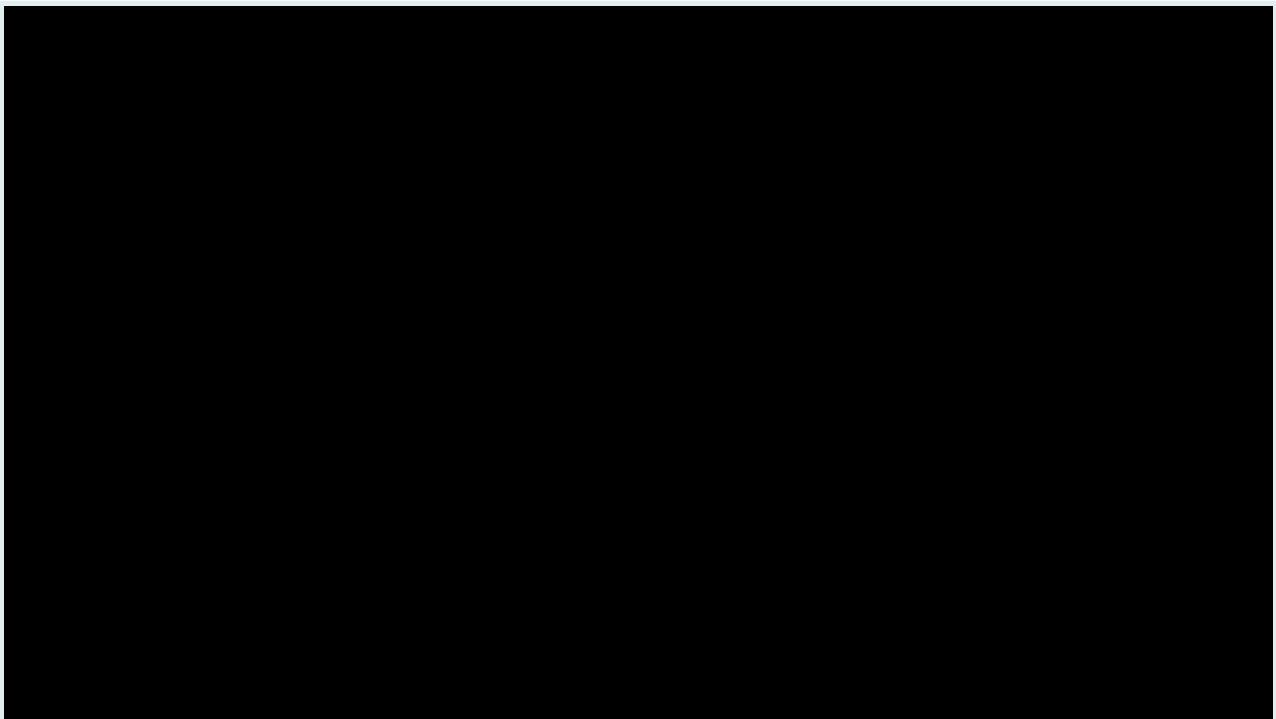
```
info_test_multiples <- tribble(
  ~nom,      ~date_test, ~resultat,
  "Alice",   "2023-06-05", "Negatif",
  "Alice",   "2023-06-10", "Positif",
  "Bob",     "2023-08-10", "Positif",
  "Xavier",  "2023-05-02", "Negatif",
  "Xavier",  "2023-05-12", "Negatif",
)
```

Examinons maintenant ce qui se passe lorsque nous utilisons une jointure de type `left_join()`, avec le jeu de données `demographique` à gauche de l'appel:

```
left_join(demographique, info_test_multiples)
```

```
## Joining with `by = join_by(nom)`
```

Que s'est-il passé ? Eh bien, nous savons qu'Alice était présente dans le jeu de données de gauche, sa ligne a donc été conservée. Mais elle apparaissait deux fois dans le jeu de données de droite, donc ses informations démographiques ont été dupliquées dans le jeu de données final. Xavier n'était pas dans le jeu de données de gauche, il a donc été supprimé. En résumé, lorsqu'une jointure un-à-plusieurs est effectuée, les données du côté "un" sont dupliquées pour chaque ligne correspondante du côté "plusieurs". L'illustration ci-dessous présente ce processus :



Nous pouvons voir le même résultat lorsque l'ordre des jeux de données est inversé, en plaçant `info_test_multiples` à gauche de l'appel.

```
left_join(info_test_multiples, demographique)
```

```
## Joining with `by = join_by(nom)`
```

Encore une fois, les données démographiques d'Alice ont été dupliquées ! Xavier était présent dans le jeu de données de gauche `info_test_multiples` donc ses lignes ont été conservées, mais comme il n'était pas dans le jeu de données `demographique`, les cellules correspondantes sont définies sur `NA`.

Copiez le code ci-dessous pour créer deux petits dataframes :

```

info_patient <- tribble(
  ~patient_num, ~nom,      ~age,
  1,           "Liam",    32,
  2,           "Manny",   28,
  3,           "Nico",    40
)

maladies <- tribble(
  ~patient_num, ~maladie,
  1,            "Diabète",
  1,            "Hypertension",
  2,            "Asthme",
  3,            "Cholestérol Élevé",
  3,            "Arthrite"
)

```

Si vous utilisez une fonction `left_join()` pour joindre ces ensembles de données, combien de lignes y aura-t-il dans le dataframe final ? Essayez de le déterminer, puis effectuez la jointure pour voir si vous aviez raison !

Appliquons cela à nos jeux de données du monde réel. Le premier jeu de données sur lequel nous allons travailler est le jeu de données `notification`. Pour rappel, voici à quoi il ressemble :

```
notification
```

```

## # A tibble: 5 × 4
##   Etat                      systeme_sante cible_notifiee
##   <chr>                     <chr>          <dbl>
## 1 Iles Andaman et Nicobar  public            520
## 2 Iles Andaman et Nicobar  privé             10
## 3 Andhra Pradesh           public            85000
## 4 Andhra Pradesh           privé            30000
## 5 Arunachal Pradesh       public            3450
##   notifiee_relle            <dbl>
##   1                         510
##   2                         24
##   3                        62075
##   4                        30112
##   5                        2722

```

Notre second jeu de données contient 32 des 36 états et territoires de l'union indienne, ainsi que leur catégorie de subdivision et le conseil zonal dans lequel ils sont situés.

```
regions <- read_csv(here("data/regions_FR.csv"))
```

```
regions
```

```

## # A tibble: 5 × 3
##   conseil_zonal      sous_division
##   <chr>                <chr>
## 1 Pas de Conseil Zonal  Territoire de l'Union
## 2 Conseil du Nord-Est    État
## 3 Conseil du Nord-Est    État
## 4 Conseil Zonal de l'Est  État
## 5 Conseil Zonal du Nord  Territoire de l'Union
##   Etat
##   <chr>
## 1 Iles Andaman et Nicobar
## 2 Arunachal Pradesh
## 3 Assam
## 4 Bihar
## 5 Chandigarh

```

Tout d'abord, vérifions s'il y a des différences entre les jeux de données:

```
setdiff(notification$Etat, regions$Etat)
```

```

## [1] "Andhra Pradesh" "Chhattisgarh"    "Ladakh"
## [4] "Tamil Nadu"

```

```
setdiff(regions$Etat, notification$Etat)
```

```

## character(0)

```

Comme nous pouvons le voir, il y a quatre états dans le jeu de données `notification` qui ne sont pas dans le jeu de données `regions`. Ce ne sont pas des erreurs à corriger, nous n'avons simplement pas l'information complète dans notre jeu de données `regions`. Si nous voulons conserver tous les cas de `notification`, nous devrons le placer en position de gauche pour notre jointure. Essayons cela !

```
notif_regions <- notification %>%
  left_join(regions)
```

```

## Joining with `by = join_by(Etat)`

```

Comme prévu, les données du jeu de données `regions` ont été dupliquées pour chaque valeur correspondante du jeu de données `notification`. Pour les états qui ne sont pas dans le jeu de données `regions`, comme l'`Andhra Pradesh`, les cellules correspondantes sont définies sur `NA`.

différences et similitudes avec une jointure interne, `inner_join()`.

En utilisant un `left_join()`, joindre le jeu de données de tuberculose pédiatrique `enfant` avec le jeu de données `regions` en conservant toutes les valeurs du jeu de données `enfant`.

`inner_join()`

Lors de l'utilisation d'une jointure interne `inner_join()` avec une relation un-à-plusieurs, les mêmes principes s'appliquent qu'avec un `left_join()`. Pour illustrer cela, regardons à nouveau nos données de patients COVID et leurs informations de test.

```
demographique
```

```
info_test_multiples
```

Maintenant, voyons ce qui se passe lorsque nous utilisons un `inner_join()` pour joindre ces deux jeux de données.

```
inner_join(demographique, info_test_multiples)
```

```
## Joining with `by = join_by(nom)`
```

Avec un `inner_join()`, les valeurs communes entre les jeux de données sont conservées et celles du côté “un” sont dupliquées pour chaque ligne du côté “plusieurs”.

Puisqu’Alice et Bob sont communs entre les deux jeux de données, ce sont les seuls à être conservés. Et comme Alice apparaît deux fois dans `info_test_multiples`, sa ligne du jeu de données `demographique` est dupliquée !

Essayons cela avec notre jeu de données `covid` sur la tuberculose et notre jeu de données `regions`. Pour rappel, voici nos jeux de données:

```
covid
```

```
regions
```

Comme nous l’avons vu précédemment, le jeu de données `regions` manque 4 états/territoires de l’Union, nous pouvons donc nous attendre à ce qu’ils soient exclus de notre jeu de données final avec un `inner_join()`. Créons un nouveau jeu de données appelé `inner_covid_regions`.

```
inner_covid_regions <- covid %>%  
  inner_join(regions)
```

```
## Joining with `by = join_by(Etat)`
```

```
inner_covid_regions
```

Parfait, c'est exactement ce que nous voulions !

Utilisez la fonction `set_diff()` pour comparer les valeurs entre les jeux de données `enfant` et `regions`. Puis, utilisez un `inner_join()` pour joindre les deux jeux de données. Combien d'observations sont conservées ?

Colonnes clés multiples

Parfois, nous avons plus d'une colonne permettant d'identifier de manière unique les observations que nous souhaitons appairer. Par exemple, imaginons que nous ayons des mesures de pression artérielle systolique et diastolique pour trois patients avant (pre) et après (post) la prise d'un nouveau médicament hypotenseur.

```
tension_arterielle <- tribble(  
  ~nom,      ~temps,    ~systolique,   ~diastolique,  
  "David",   "pre",     139,           87,  
  "David",   "post",    121,           82,  
  "Eamon",   "pre",     137,           86,  
  "Eamon",   "post",    128,           79,  
  "Flavio",  "pre",     137,           81,  
  "Flavio",  "post",    130,           73  
)  
tension_arterielle
```

Maintenant, imaginons que nous ayons un autre jeu de données avec les mêmes 3 patients et leurs taux de créatinine avant et après la prise du médicament. La créatinine est un déchet normalement éliminé par les reins. Si les taux de créatinine dans le sang augmentent, cela peut signifier que les reins ne fonctionnent pas correctement, ce qui peut être un effet secondaire des médicaments hypotenseurs.

```
rein <- tribble(  
  ~nom,      ~temps,    ~creatinine,  
  "David",   "pre",     0.9,  
  "David",   "post",    1.3,  
  "Eamon",   "pre",     0.7,  
  "Eamon",   "post",    0.8,  
  "Flavio",  "pre",     0.6,  
  "Flavio",  "post",    1.4  
)  
rein
```

Nous souhaitons joindre les deux jeux de données de sorte que chaque patient ait deux lignes, une ligne pour sa tension artérielle et sa créatininémie avant la prise du médicament, et une ligne pour sa tension artérielle et sa créatininémie après le médicament. Pour cela, notre premier réflexe serait de joindre sur le nom des patients. Essayons et voyons ce qui se passe :

```
tension_rein_dups <- tension_arterielle %>%
  left_join(rein, by = "nom")
```

```
## Warning in left_join(., rein, by = "nom"): Detected an unexpected many-to-
many relationship between
## `x` and `y`.
## i Row 1 of `x` matches multiple rows in `y`.
## i Row 1 of `y` matches multiple rows in `x`.
## i If a many-to-many relationship is expected, set
##   `relationship = "many-to-many"` to silence this
## warning.
```

```
tension_rein_dups
```

Comme nous pouvons le voir, ce n'est pas du tout ce que nous voulions ! Nous pouvons joindre sur les noms de patients, mais R affiche un message d'avertissement indiquant qu'il s'agit d'une relation « plusieurs-à-plusieurs » car plusieurs lignes dans un jeu de données correspondent à plusieurs lignes dans l'autre jeu de données, ce qui fait que nous obtenons 4 lignes par patient. En règle générale, vous devriez éviter les jointures plusieurs-à-plusieurs! Notez également que comme nous avons deux colonnes appelées `temps` (une dans chaque jeu de données), ces colonnes sont différencierées dans le nouveau jeu de données par `.x` et `.y`.

Ce que nous voulons faire, c'est apparier à la fois le `nom` et le `temps`. Pour cela, nous devons spécifier à R qu'il y a deux colonnes d'appariement. En réalité, c'est très simple ! Tout ce que nous avons à faire est d'utiliser la fonction `c()` et de préciser les deux noms de colonnes.

```
tension_rein <- tension_arterielle %>%
  left_join(rein, by = c("nom", "temps"))
tension_rein
```

C'est parfait ! Appliquons cela maintenant à nos jeux de données réels `notification` et `covid`.

```
notification
```

```
covid
```

Réfléchissons à la forme que nous souhaitons voir avoir pour notre jeu de données final. Nous voulons avoir deux lignes par état, une avec les données de notification de la tuberculose et du COVID pour le secteur public, et une pour le secteur privé. Cela signifie que nous devons apparier sur `state_UT` et `hc_type`. Tout comme pour les données de patients, nous devons spécifier les deux valeurs clés dans la clause `by=` en utilisant `c()`. Essayons !

```
notif_covid <- notification %>%
  left_join(covid, by=c("Etat", "systeme_sante"))
notif_covid
```

Super, c'est exactement ce que nous voulions !

Créez un nouveau jeu de données appelé `TB_final` qui rassemble le jeu de données `notif_covid` avec le jeu de données `enfant`. Puis, joignez ce jeu de données avec le jeu de données `regions` pour obtenir un jeu de données combiné final, en vous assurant qu'aucune donnée de tuberculose n'est perdue.

Contributors

The following team members contributed to this lesson:



KENE DAVID NWOSU

Data analyst, the GRAPH Network

Passionate about world improvement

(make sure to update the contributor list accordingly!)