

ANALYSE DE DONNÉES INTERMÉDIAIRE AVEC

AVEC DES DONNÉES RÉELLES SUR
LE VIH, LA TB & LE PALUDISME



The GRAPH Courses, le Fonds Mondial & l'OMS

Ce livre est une compilation de matériaux de formation créée par The GRAPH Network, soutenue par le Fonds Mondial. Les matériaux visent à développer des compétences globales dans l'analyse des données épidémiologiques.

Travailler avec des chaînes de caractères en R

Introduction
Objectifs d'Apprentissage
Paquets
Définir des Chaînes
Formatage des Chaînes en R avec {stringr}
Changement de Casse
Gestion des Espaces
Mise en Forme du Texte
Habillage du Texte
Application du formatage de chaîne à un ensemble de données
Division des chaînes de caractères avec str_split() et separate()
Utilisation de str_split()
Utilisation de separate()
Séparation des Caractères Spéciaux
Combinaison de Chaînes avec paste()
Sous-référencement de chaînes avec str_sub()
Conclusion
Clés de Réponses

Notes de Leçon | Travailler avec les Chaînes de Caractères en R

Introduction

La maîtrise de la manipulation des chaînes de caractères est une compétence essentielle pour les scientifiques de données. Des tâches telles que le nettoyage de données désordonnées et la mise en forme des sorties dépendent fortement de la capacité à analyser, combiner et modifier des chaînes de caractères. Cette leçon se concentre sur les techniques de travail avec les chaînes de caractères en R, en utilisant les fonctions du package {stringr} dans le tidyverse. Plongeons dedans !

Objectifs d'Apprentissage

- Comprendre le concept de chaînes de caractères et les règles pour les définir en R
- Utiliser des échappements pour inclure des caractères spéciaux comme des guillemets dans les chaînes
- Utiliser les fonctions de {stringr} pour formater les chaînes :
 - Changer la casse avec str_to_lower(), str_to_upper(), str_to_title()
 - Supprimer les espaces superflus avec str_trim() et str_squish()
 - Compléter les chaînes pour une largeur égale avec str_pad()
 - Envelopper le texte à une certaine largeur en utilisant str_wrap()
- Diviser les chaînes en parties en utilisant str_split() et separate()

- Combiner des chaînes ensemble avec `paste()` et `paste0()`
 - Extraire des sous-chaînes des chaînes en utilisant `str_sub()`
-

Paquets

```
# Chargement des paquets requis
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, here, janitor)
```

Définir des Chaînes

Il existe des règles fondamentales pour définir des chaînes de caractères en R.

Les chaînes peuvent être encadrées soit par des guillemets simples soit par des guillemets doubles. Cependant, le type de guillemet utilisé au début doit correspondre à celui utilisé à la fin. Par exemple :

```
string_1 <- "Bonjour" # Utilisation de guillemets doubles
string_2 <- 'Bonjour' # Utilisation de guillemets simples
```

Vous ne pouvez normalement pas inclure de guillemets doubles à l'intérieur d'une chaîne qui commence et se termine par des guillemets doubles. La même règle s'applique aux guillemets simples à l'intérieur d'une chaîne qui commence et se termine par des guillemets simples. Par exemple :

```
will_not_work <- "Guillemets doubles " au sein de guillemets doubles"
will_not_work <- 'Guillemets simples ' au sein de guillemets simples'
```

Mais vous pouvez inclure des guillemets simples à l'intérieur d'une chaîne qui commence et se termine par des guillemets doubles, et vice versa :

```
single_inside_double <- "Guillemets simples ' au sein de guillemets doubles"
```

Alternativement, vous pouvez utiliser le caractère d'échappement \ pour inclure un guillemet simple ou double littéral à l'intérieur d'une chaîne :

```
single_quote <- 'Guillemets simples \' au sein de guillemets doubles'
double_quote <- "Guillemets doubles \" au sein de guillemets doubles"
```

Pour afficher ces chaînes telles qu'elles apparaîtraient dans la sortie, comme sur un graphique, utilisez cat() :

```
cat('Guillemets simples \' au sein de guillemets doubles')

## Guillemets simples ' au sein de guillemets doubles

cat("Guillemets doubles \" au sein de guillemets doubles")

## Guillemets doubles " au sein de guillemets doubles
```

cat() imprime ses arguments sans formatage supplémentaire.

Puisque \ est le caractère d'échappement, vous devez utiliser \\ pour inclure un antislash littéral dans une chaîne :

SIDE NOTE



```
backslash <- "Ceci est un antislash : \\"
cat(backslash)
```

```
## Ceci est un antislash : \
```

Q : Repérage d'Erreurs dans les Définitions de Chaînes

PRACTICE



(in RMD)

Ci-dessous, des tentatives de définition de chaînes de caractères en R, avec deux lignes sur cinq contenant une erreur. Identifiez et corrigez ces erreurs.

```
ex_a <- 'Elle a dit, "Bonjour !" à lui.'
ex_b <- "Elle a dit \"Allons sur la lune\""
ex_c <- "Ils ont été "meilleurs amis" pendant des années."
ex_d <- 'Le journal de Jane\\'
ex_e <- "C'est une journée ensoleillée !
```

Formatage des Chaînes en R avec {stringr}

Le package {stringr} en R fournit des fonctions utiles pour formater les chaînes pour l'analyse et la visualisation. Cela inclut les changements de casse, la gestion des espaces, la standardisation de la longueur et l'habillage du texte.

Changement de Casse

La conversion de la casse est souvent nécessaire pour standardiser les chaînes ou les préparer pour l'affichage. Le package {stringr} fournit plusieurs fonctions de changement de casse :

- str_to_upper() convertit les chaînes en majuscules.

```
str_to_upper("bonjour le monde")
```

```
## [1] "BONJOUR LE MONDE"
```

- str_to_lower() convertit les chaînes en minuscules.

```
str_to_lower("Au revoir")
```

```
## [1] "au revoir"
```

- str_to_title() met en majuscule la première lettre de chaque mot. Idéal pour titrer les noms, sujets, etc.

```
str_to_title("manipulation de chaîne")
```

```
## [1] "Manipulation De Chaîne"
```

Gestion des Espaces

Gérer les espaces rend les chaînes propres et uniformes. Le package {stringr} fournit deux fonctions principales pour cela :

- str_trim() supprime les espaces au début et à la fin.

```
str_trim(" espace coupé ")
```

```
## [1] "espace coupé"
```

- `str_squish()` supprime les espaces au début et à la fin, *et* réduit plusieurs espaces internes à un seul.

```
str_squish(" trop d'espace interne ")
```

```
## [1] "trop d'espace interne"
```

```
# remarquez la différence avec str_trim  
str_trim(" trop d'espace interne ")
```

```
## [1] "trop d'espace interne"
```

Mise en Forme du Texte

`str_pad()` ajoute des espaces à une chaîne pour obtenir une largeur fixe. Par exemple, nous pouvons ajouter des espaces au nombre 7 pour le forcer à avoir 3 caractères :

```
str_pad("7", width = 3, pad = "0") # Ajouter des espaces à gauche pour une longueur de 3 avec 0
```

```
## [1] "007"
```

Le premier argument est la chaîne à mettre en forme. `width` définit la largeur finale de la chaîne et `pad` spécifie le caractère de remplissage.

`side` contrôle si l'ajout d'espaces se fait à gauche ou à droite. L'argument `side` est par défaut “`left`”, donc les espaces seront ajoutés à gauche si non spécifié. Spécifier `side = "right"` ajoute des espaces à droite :

```
str_pad("7", width = 4, side = "right", pad = "_") # Ajouter des espaces à droite pour une longueur de 4 avec _
```

```
## [1] "7__"
```

Où nous pouvons ajouter des espaces des deux côtés :

```
str_pad("7", width = 5, side = "both", pad = "_") # Ajouter des espaces des deux  
côtés pour une longueur de 5 avec _
```

```
## [1] "__7__"
```

Habillement du Texte

L'habillage du texte aide à adapter les chaînes dans des espaces restreints comme les titres de graphiques. La fonction `str_wrap()` habille le texte à une largeur définie.

Par exemple, pour habiller un texte à 10 caractères, nous pouvons écrire :

```
example_string <- "Manipulation de chaînes avec str_wrap peut améliorer la lisibilité  
dans les graphiques."  
wrapped_to_10 <- str_wrap(example_string, width = 10)  
wrapped_to_10
```

```
## [1] "Manipulation\nde  
chaînes\avec\nstr_wrap\npeut\naméliorer\nla\nlisibilité\ndans  
les\ngraphiques."
```

La sortie peut paraître déroutante. Le `\n` indique un saut de ligne, et pour voir la modification correctement, nous devons utiliser la fonction `cat()`, qui est une version spéciale de `print()` :

```
cat(wrapped_to_10)
```

```
## Manipulation  
## de chaînes  
## avec  
## str_wrap  
## peut  
## améliorer  
## la  
## lisibilité  
## dans les  
## graphiques.
```

Notez que la fonction conserve les mots entiers, donc elle ne divisera pas les mots plus longs comme "manipulation".

Définir la largeur à 1 divise essentiellement la chaîne en mots individuels :

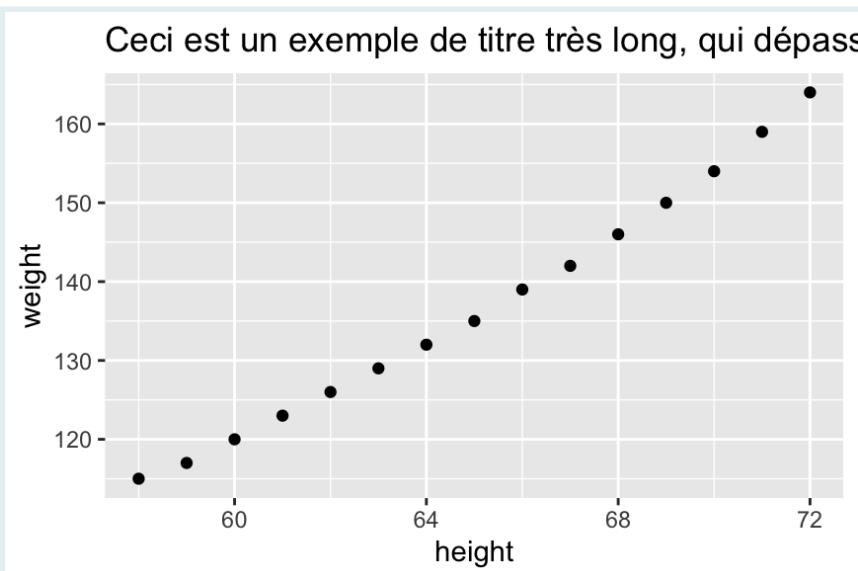
```
cat(str_wrap(example_string, width = 1))
```

```
## Manipulation
## de
## chaînes
## avec
## str_wrap
## peut
## améliorer
## la
## lisibilité
## dans
## les
## graphiques.
```

`str_wrap()` est particulièrement utile dans la création de graphiques avec `ggplot2`. Par exemple, en habillant un long titre pour éviter qu'il ne déborde du graphique :

```
long_title <- "Ceci est un exemple de titre très long, qui dépasserait normalement de
votre ggplot, mais vous pouvez l'habiller avec str_wrap pour l'adapter à une
limite de caractères spécifiée."
```

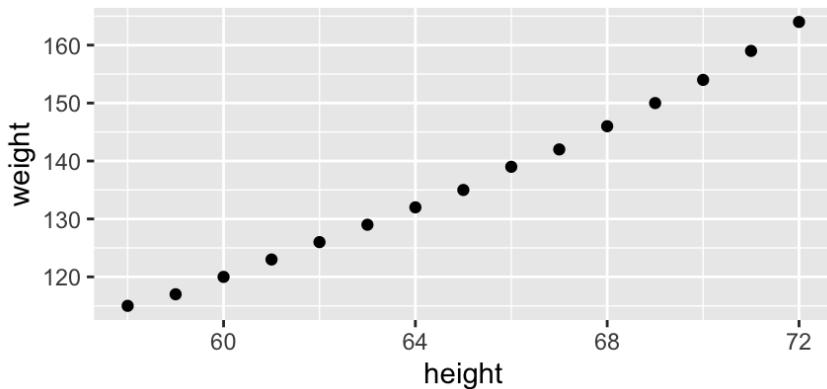
```
# Sans habillage
ggplot(women, aes(height, weight)) +
  geom_point() +
  labs(title = long_title)
```



```
# Avec habillage à 80 caractères
ggplot(women, aes(height, weight)) +
```

```
geom_point() + title = str_wrap(long_title, width = 50))
```

Ceci est un exemple de titre très long, qui dépasserait normalement de votre ggplot, mais vous pouvez l'habiller avec str_wrap pour l'adapter à une limite de caractères spécifiée.



Ainsi, str_wrap() maintient les titres soigneusement à l'intérieur du graphique !

Q : Nettoyage des données de noms de patients

Un jeu de données contient des noms de patients avec un formatage inconsistante et des espaces blancs supplémentaires. Utilisez le package {stringr} pour standardiser ces informations :

```
patient_names <- c(" john doe", "ANNA SMITH ", "Emily Davis")  
# 1. Supprimez les espaces blancs de chaque nom.  
# 2. Convertissez chaque nom en casse de titre pour la cohérence.
```

Q : Standardisation des codes de médicaments



Les codes de médicaments suivants (fictifs) sont formatés de manière inconsistante. Standardisez-les en ajoutant des zéros pour garantir que tous les codes aient 8 caractères de long :

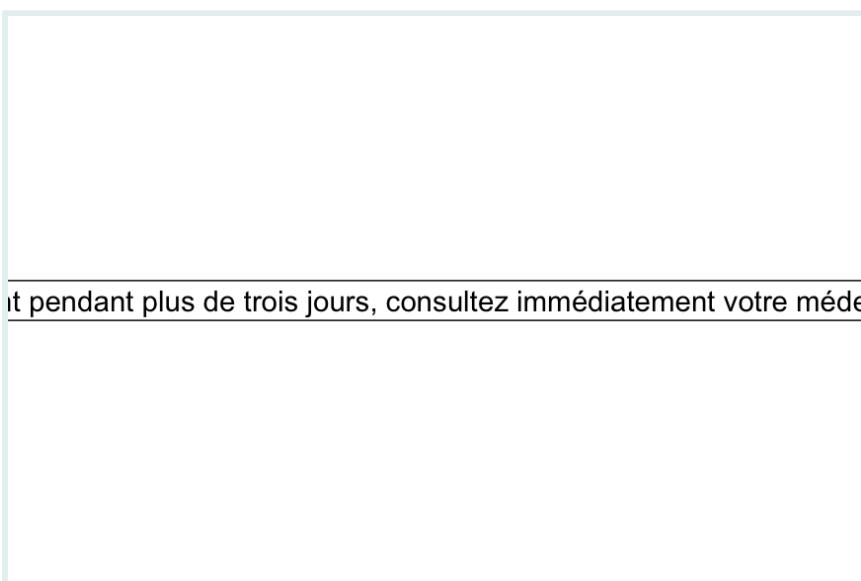
```
drug_codes <- c("12345", "678", "91011")  
# Ajoutez des zéros à chaque code à gauche pour une largeur fixe de 8 caractères.
```

Q : Habillage des instructions médicales

Utilisez `str_wrap()` pour formater ce qui suit pour une meilleure lisibilité :

```
instructions <- "Prenez deux comprimés quotidiennement après les repas. Si les symptômes persistent plus de trois jours, consultez immédiatement votre médecin. Ne prenez pas plus que la dose recommandée. Tenir hors de portée des enfants."
```

```
ggplot(data.frame(x = 1, y = 1), aes(x, y, label = instructions)) +  
  geom_label() +  
  theme_void()
```



it pendant plus de trois jours, consultez immédiatement votre médecin

```
# Maintenant, habillez les instructions à une largeur de 50 caractères puis tracez à nouveau.
```

Application du formatage de chaîne à un ensemble de données

Maintenant, appliquons les fonctions de formatage de chaîne du package `{stringr}` pour nettoyer et standardiser un ensemble de données. Notre focus est sur un ensemble de données issu d'une étude sur les services de soins et de traitement du VIH dans la province de Zambézia, au Mozambique, disponible [ici](#). L'ensemble de données original comportait diverses incohérences de formatage, mais nous avons ajouté des erreurs supplémentaires à des fins éducatives.

D'abord, nous chargeons l'ensemble de données et examinons des variables spécifiques pour des problèmes potentiels.

```
# Charger l'ensemble de données  
hiv_dat_messy_1 <- openxlsx::read.xlsx(here("data/hiv_dat_messy_1.xlsx")) %>%  
  as_tibble()
```

```
# Ces quatre variables contiennent des incohérences de forme  
hiv_dat_messy_1 %>%  
select(district, health_unit, education, regimen)
```

```
## # A tibble: 1,413 × 4  
##   district  health_unit      education    regimen  
##   <chr>     <chr>        <chr>        <chr>  
## 1 "Rural"   District Hospital Maganj... MISSING    AZT+3TC+NVP  
## 2 "Rural"   District Hospital Maganj... secondary  TDF+3TC+EFV  
## 3 "Urban"   24th Of July Health ... MISSING    tdf+3tc+efv  
## 4 "Urban"   24th Of July Health ... MISSING    TDF+3TC+EFV  
## 5 "Urban"   24th Of July Health ... University  tdf+3tc+efv  
## 6 "Urban"   24th Of July Health Faci... Technical  AZT+3TC+NVP  
## 7 "Rural"   District Hospital Maganj... Technical  TDF+3TC+EFV  
## 8 "Urban"   24th Of July Health Faci... Technical  azt+3tc+nvp  
## 9 "Urban"   24th Of July Health Faci... Technical  AZT+3TC+NVP  
## 10 "Urban"  24th Of July Health Faci... Technical  TDF+3TC+EFV  
## # i 1,403 more rows
```

En utilisant la fonction tabyl, nous pouvons identifier et compter les valeurs uniques, révélant les incohérences :

```
# Comptage des valeurs uniques  
hiv_dat_messy_1 %>% tabyl(health_unit)
```

```
##          health_unit    n    percent  
## 24th Of July Health Facility 239 0.16914367  
## 24th Of July Health Facility 249 0.17622081  
## District Hospital Maganja Da Costa 342 0.24203822  
## District Hospital Maganja Da Costa 336 0.23779193  
## Nante Health Facility 119 0.08421798  
## Nante Health Facility 128 0.09058740
```

```
hiv_dat_messy_1 %>% tabyl(education)
```

```
##   education    n    percent  
##   MISSING 776 0.549186129  
##   None 128 0.090587403  
##   Primary 178 0.125973107  
##   Secondary 82 0.058032555  
##   Technical 17 0.012031139  
##   University 4 0.002830856  
##   primary 157 0.111111111  
##   secondary 71 0.050247700
```

```
hiv_dat_messy_1 %>% tabyl(regimen)
```

```

##      regimen   n     percent valid_percent
##  AZT+3TC+EFV  24 0.0169851380  0.0179910045
##  AZT+3TC+NVP 229 0.1620665251  0.1716641679
##  D4T+3TC+ABC   1 0.0007077141  0.0007496252
##  D4T+3TC+EFV   2 0.0014154282  0.0014992504
##  D4T+3TC+NVP  16 0.0113234253  0.0119940030
##  OTHER       1 0.0007077141  0.0007496252
##  TDF+3TC+EFV 404 0.2859164897  0.3028485757
##  TDF+3TC+NVP   3 0.0021231423  0.0022488756
##  azt+3tc+efv  16 0.0113234253  0.0119940030
##  azt+3tc+nvp 231 0.1634819533  0.1731634183
##  d4t+3tc+efv   9 0.0063694268  0.0067466267
##  d4t+3tc+nvp  18 0.0127388535  0.0134932534
##  d4t+4tc+nvp   1 0.0007077141  0.0007496252
##  d4t6+3tc+nvp   2 0.0014154282  0.0014992504
##  other        2 0.0014154282  0.0014992504
##  tdf+3tc+efv 374 0.2646850672  0.2803598201
##  tdf+3tc+nvp   1 0.0007077141  0.0007496252
##      <NA>    79 0.0559094126          NA

```

```
hiv_dat_messy_1 %>% tabyl(district)
```

```

##  district   n     percent
##  Rural    234 0.16560510
##  Urban    118 0.08351026
##  Rural    691 0.48903043
##  Urban    370 0.26185421

```

Une autre fonction utile pour visualiser ces problèmes est `tbl_summary` du package `{gtsummary}` :

```
hiv_dat_messy_1 %>%
  select(district, health_unit, education, regimen) %>%
  tbl_summary()
```

Characteristic	N = 1,413¹
district	
Rural	234 (17%)
Urban	118 (8.4%)
Rural	691 (49%)
Urban	370 (26%)
health_unit	
24th Of July Health Facility	239 (17%)
24th Of July Health Facility	249 (18%)
District Hospital Maganja Da Costa	342 (24%)
District Hospital Maganja Da Costa	336 (24%)
Nante Health Facility	119 (8.4%)
Nante Health Facility	128 (9.1%)
education	
MISSING	776 (55%)
None	128 (9.1%)
primary	157 (11%)
Primary	178 (13%)
secondary	71 (5.0%)
Secondary	82 (5.8%)
Technical	17 (1.2%)
University	4 (0.3%)
regimen	
azt+3tc+efv	16 (1.2%)
AZT+3TC+EFV	24 (1.8%)

Characteristic	N = 1,413 ¹
AZT+3TC+NVP	229 (17%)
D4T+3TC+ABC	1 (<0.1%)
d4t+3tc+efv	9 (0.7%)
D4T+3TC+EFV	2 (0.1%)
d4t+3tc+nvp	18 (1.3%)
D4T+3TC+NVP	16 (1.2%)
d4t+4tc+nvp	1 (<0.1%)
d4t6+3tc+nvp	2 (0.1%)
other	2 (0.1%)
OTHER	1 (<0.1%)
tdf+3tc+efv	374 (28%)
TDF+3TC+EFV	404 (30%)
tdf+3tc+nvp	1 (<0.1%)
TDF+3TC+NVP	3 (0.2%)
Unknown	79

¹ n (%)

La sortie montre clairement des incohérences dans la casse, l'espacement et le format, donc nous devons les standardiser.

Ensuite, nous abordons ces problèmes de manière systématique :

```
hiv_dat_clean_1 <- hiv_dat_messy_1 %>%
  mutate(
    district = str_to_title(str_trim(district)), # Standardiser les noms de district
    health_unit = str_squish(health_unit),        # Supprimer les espaces supplémentaires
    education = str_to_title(education),          # Standardiser les niveaux d'éducation
    regimen = str_to_upper(regimen)              # Consistance dans la colonne régime
  )
```

Et nous pouvons vérifier l'efficacité de ces changements en réexécutant la fonction `tbl_summary()` :

```
hiv_dat_clean_1 %>%
  select(district, health_unit, education, regimen) %>%
  tbl_summary()
```

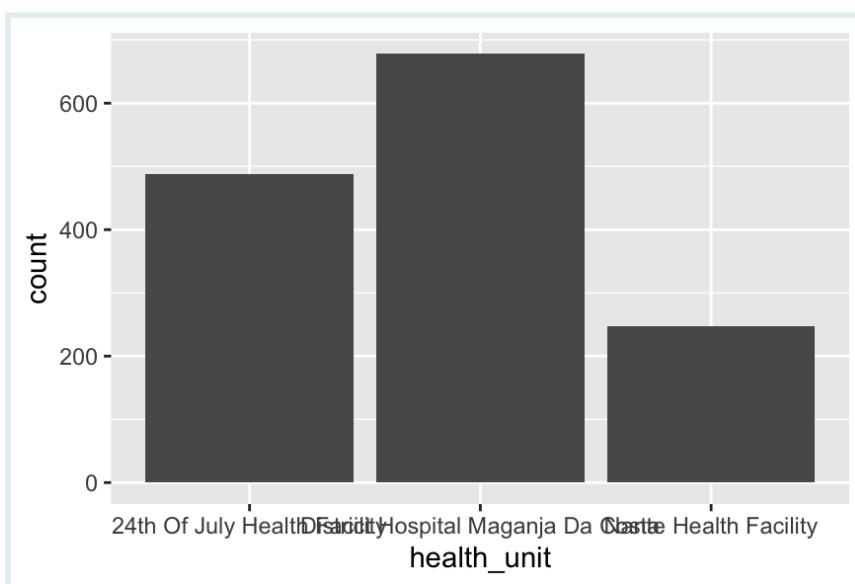
Characteristic	N = 1,413¹
district	
Rural	925 (65%)
Urban	488 (35%)
health_unit	
24th Of July Health Facility	488 (35%)
District Hospital Maganja Da Costa	678 (48%)
Nante Health Facility	247 (17%)
education	
Missing	776 (55%)
None	128 (9.1%)
Primary	335 (24%)
Secondary	153 (11%)
Technical	17 (1.2%)
University	4 (0.3%)
regimen	
AZT+3TC+EFV	40 (3.0%)
AZT+3TC+NVP	460 (34%)
D4T+3TC+ABC	1 (<0.1%)
D4T+3TC+EFV	11 (0.8%)
D4T+3TC+NVP	34 (2.5%)
D4T+4TC+NVP	1 (<0.1%)
D4T6+3TC+NVP	2 (0.1%)
OTHER	3 (0.2%)
TDF+3TC+EFV	778 (58%)

Characteristic N = 1,413 ¹	
Unknown	79
¹ n (%)	

Super !

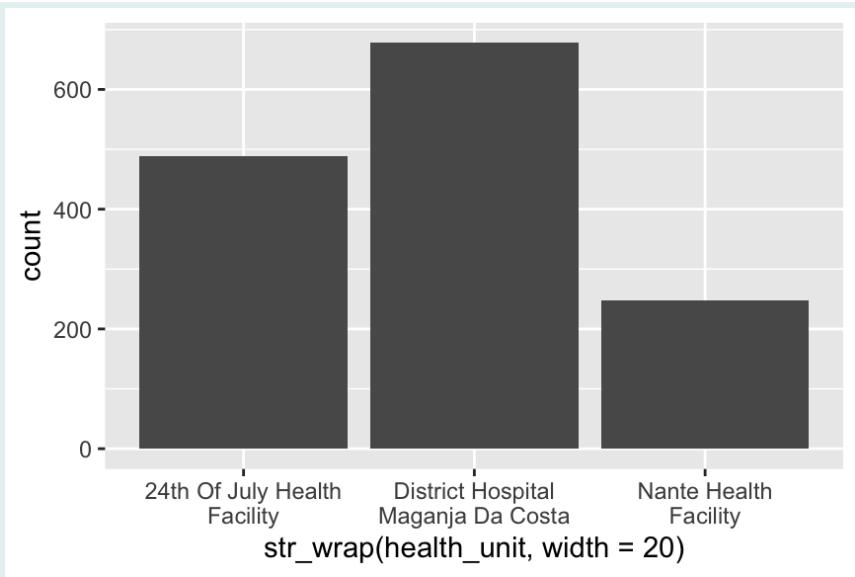
Enfin, essayons de tracer des comptages de la variable `health_unit`. Pour le style de tracé ci-dessous, nous rencontrons un problème avec des étiquettes longues :

```
ggplot(hiv_dat_clean_1, aes(x = health_unit)) +
  geom_bar()
```



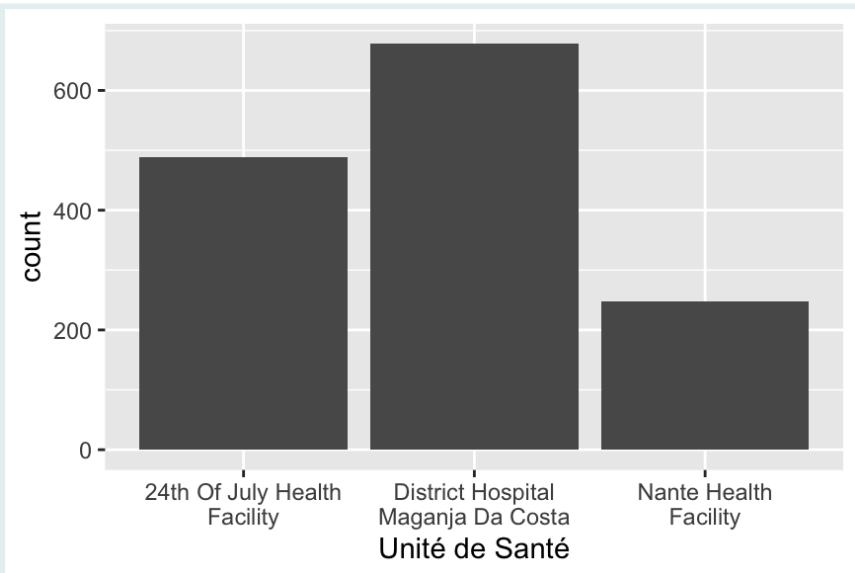
Pour résoudre cela, nous pouvons ajuster les étiquettes en utilisant `str_wrap()` :

```
hiv_dat_clean_1 %>%
  ggplot(aes(x = str_wrap(health_unit, width = 20))) +
  geom_bar()
```



Beaucoup plus propre, bien que nous devrions probablement corriger le titre de l'axe

```
hiv_dat_clean_1 %>%
  ggplot(aes(x = str_wrap(health_unit, width = 20))) +
  geom_bar() +
  labs(x = "Unité de Santé")
```



Maintenant, essayez votre main sur des opérations de nettoyage similaires dans les questions de pratique ci-dessous.

PRACTICE



Q : Formatage d'un ensemble de données sur la tuberculose

Dans cet exercice, vous nettoierez un ensemble de données, `lima_messy`, provenant d'une étude sur l'adhésion au traitement de la tuberculose à Lima, au Pérou. Plus de détails sur l'étude et l'ensemble de données sont disponibles [ici](#).

Commencez par importer l'ensemble de données :

```
lima_messy_1 <- openxlsx::read.xlsx(here("data/lima_messy_1.xlsx"))
%>%
  as_tibble()
lima_messy_1
```



```
## # A tibble: 1,293 × 18
##   id      age       sex marital_status
##   <chr>    <chr>     <chr> <chr>
## 1 pe-1008 38 and older M   Single
## 2 lm-1009 38 and older M   Married / cohabitating
## 3 pe-1010 27 to 37   m   Married / cohabitating
## 4 lm-1011 27 to 37   m   Married / cohabitating
## 5 pe-1012 38 and older m   Married / cohabitating
## 6 lm-1013 27 to 37   M   Single
## 7 pe-1014 27 To 37    m   Married / cohabitating
## 8 lm-1015 22 To 26    m   Single
## 9 pe-1016 27 to 37    m   Single
## 10 lm-1017 22 to 26   m   Single
## # i 1,283 more rows
## # i 14 more variables: poverty_level <chr>, ...
```

Votre tâche est de nettoyer les variables `marital_status`, `sex` et `age` dans `lima_messy`. Après le processus de nettoyage, générez un tableau récapitulatif en utilisant la fonction `tbl_summary()`. Visez à ce que votre sortie s'aligne sur cette structure :

Caractéristique N = 1,293	
marital_status	
Divorcé / Séparé	93 (7.2%)
Marié / Cohabitant	486 (38%)
Célibataire	677 (52%)
Veuf	37 (2.9%)
sex	
F	503 (39%)
M	790 (61%)

Caractéristique N = 1,293

age	
21 ans et moins	338 (26%)
22 à 26 ans	345 (27%)
27 à 37 ans	303 (23%)
38 ans et plus	307 (24%)

Mettez en œuvre le nettoyage et résumez :

PRACTICE

```
# Créer un nouvel objet pour les données nettoyées
• lima_clean <- lima_messy %>%
  mutate(
  (in RMD) # Nettoyer marital_status

    # Nettoyer sex

    # Nettoyer age

  )

# Vérifier le nettoyage
lima_clean %>%
  select(marital_status, sex, age) %>%
 tbl_summary()
```

Q : Habillage des étiquettes d'axe dans un graphique

À l'aide du jeu de données nettoyé lima_clean de la tâche précédente, créez un diagramme en barres pour afficher le nombre de participants par statut_marital. Ensuite, habillez les étiquettes de l'axe des x pour qu'elles n'aient pas plus de 15 caractères par ligne, afin d'améliorer la lisibilité.

```
# Créez votre diagramme en barres avec du texte habillé ici :
```

Division des chaînes de caractères avec str_split() et separate()

Diviser des chaînes de caractères est une tâche courante dans la manipulation de données. Le tidyverse offre des fonctions efficaces pour cette tâche, notamment stringr::str_split() et tidyr::separate().

Utilisation de str_split()

La fonction `str_split()` est utile pour diviser des chaînes en parties. Par exemple :

```
exemple_chaine <- "diviser-cette-chaine"  
str_split(exemple_chaine, pattern = "-")
```

```
## [[1]]  
## [1] "diviser" "cette"    "chaine"
```

Ce code divise `exemple_chaine` à chaque trait d'union.

Cependant, appliquer `str_split()` directement à un dataframe peut être plus complexe.

Essayons-le avec le jeu de données IRS du Malawi comme étude de cas. Vous devriez déjà être familier avec ce jeu de données d'une leçon précédente. Il est disponible [ici](#). Pour l'instant, nous nous concentrerons sur la colonne `start_date_long` :

```
irs <- read_csv(here("data/Iollovo_data.csv"))  
irs_dates_1 <- irs %>% select(village, start_date_long)  
irs_dates_1
```

```
## # A tibble: 112 × 2  
##   village           start_date_long  
##   <chr>             <chr>  
## 1 Mess              April 07 2014  
## 2 Nkombedzi         April 22 2014  
## 3 B Compound        May 13 2014  
## 4 D Compound        May 13 2014  
## 5 Post Office       May 13 2014  
## 6 Mangulenje        May 15 2014  
## 7 Mangulenje Senior May 27 2014  
## 8 Old School         May 27 2014  
## 9 Mwanza            May 28 2014  
## 10 Alumenda          June 18 2014  
## # i 102 more rows
```

Supposons que nous voulions diviser la variable `start_date_long` pour extraire le jour, le mois et l'année. Nous pouvons écrire :

```
irs_dates_1 %>%  
  mutate(start_date_parts = str_split(start_date_long, " "))
```

```

## # A tibble: 112 × 3
##   village      start_date_long start_date_parts
##   <chr>        <chr>          <list>
## 1 Mess         April 07 2014  <chr [3]>
## 2 Nkombedzi    April 22 2014  <chr [3]>
## 3 B Compound   May 13 2014   <chr [3]>
## 4 D Compound   May 13 2014   <chr [3]>
## 5 Post Office  May 13 2014   <chr [3]>
## 6 Mangulenje   May 15 2014   <chr [3]>
## 7 Mangulenje Senior May 27 2014 <chr [3]>
## 8 Old School   May 27 2014   <chr [3]>
## 9 Mwanza       May 28 2014   <chr [3]>
## 10 Alumenda    June 18 2014  <chr [3]>
## # i 102 more rows

```

Cela résulte en une colonne de liste, qui peut être difficile à utiliser. Pour la rendre plus lisible, nous pouvons utiliser `unnest_wider()` :

```

irs_dates_1 %>%
  mutate(start_date_parts = str_split(start_date_long, " ")) %>%
  unnest_wider(start_date_parts, names_sep = "_")

```

```

## # A tibble: 112 × 5
##   village      start_date_long start_date_parts_1
##   <chr>        <chr>          <chr>
## 1 Mess         April 07 2014  April
## 2 Nkombedzi    April 22 2014  April
## 3 B Compound   May 13 2014   May
## 4 D Compound   May 13 2014   May
## 5 Post Office  May 13 2014   May
## 6 Mangulenje   May 15 2014   May
## 7 Mangulenje Senior May 27 2014 May
## 8 Old School   May 27 2014   May
## 9 Mwanza       May 28 2014   May
## 10 Alumenda    June 18 2014  June
## # i 102 more rows
## # i 2 more variables: start_date_parts_2 <chr>, ...

```

Ça fonctionne ! Nos parties de date sont maintenant séparées. Cependant, cette approche est assez encombrante. Une meilleure solution pour diviser les composants est la fonction `separate()`.

Utilisation de `separate()`

Essayons la même tâche avec `separate()` :

```

irs_dates_1 %>%
  separate(start_date_long, into = c("mois", "jour", "année"), sep = " ")

```

```

## # A tibble: 112 × 4
##   village      mois  jour  année
##   <chr>        <chr> <chr> <chr>
## 1 Mess         April 07   2014
## 2 Nkombedzi   April 22   2014
## 3 B Compound  May   13   2014
## 4 D Compound  May   13   2014
## 5 Post Office May   13   2014
## 6 Mangulenje  May   15   2014
## 7 Mangulenje Senior May  27   2014
## 8 Old School   May   27   2014
## 9 Mwanza       May   28   2014
## 10 Alumenda    June  18   2014
## # i 102 more rows

```

Bien plus simple !

Cette fonction nécessite de spécifier :

- La colonne à diviser.
- `into` - Noms des nouvelles colonnes.
- `sep` - Le caractère séparateur.

Pour conserver la colonne originale, utilisez `remove = FALSE` :

```

irs_dates_1 %>%
  separate(start_date_long, into = c("mois", "jour", "année"), sep = " ", remove =
  FALSE)

```

```

## # A tibble: 112 × 5
##   village      start_date_long  mois  jour  année
##   <chr>        <chr>        <chr> <chr> <chr>
## 1 Mess         April 07 2014   April 07   2014
## 2 Nkombedzi   April 22 2014   April 22   2014
## 3 B Compound  May 13 2014    May   13   2014
## 4 D Compound  May 13 2014    May   13   2014
## 5 Post Office May 13 2014    May   13   2014
## 6 Mangulenje  May 15 2014    May   15   2014
## 7 Mangulenje Senior May 27 2014  May   27   2014
## 8 Old School   May 27 2014    May   27   2014
## 9 Mwanza       May 28 2014    May   28   2014
## 10 Alumenda    June 18 2014   June  18   2014
## # i 102 more rows

```

SIDE NOTE



Alternativement, le package `lubridate` offre des fonctions pour extraire les composants des dates :

```

  irs_dates_1 %>%
    mutate(start_date_long = mdy(start_date_long)) %>%
    mutate(jour = day(start_date_long),
          mois = month(start_date_long, label = TRUE),
          année = year(start_date_long))

```

SIDE NOTE



```

## # A tibble: 112 × 5
##   village      start_date_long  jour mois  année
##   <chr>        <date>        <int> <ord> <dbl>
## 1 Mess         2014-04-07     7   Apr  2014
## 2 Nkombedzi   2014-04-22     22  Apr  2014
## 3 B Compound  2014-05-13     13  May  2014
## 4 D Compound  2014-05-13     13  May  2014
## 5 Post Office 2014-05-13     13  May  2014
## 6 Mangulenje  2014-05-15     15  May  2014
## 7 Mangulenje Senior 2014-05-27 27   May  2014
## 8 Old School   2014-05-27     27  May  2014
## 9 Mwanza       2014-05-28     28  May  2014
## 10 Alumenda    2014-06-18    18   Jun  2014
## # i 102 more rows

```

Lorsque certaines lignes manquent de toutes les parties nécessaires, `separate()` émettra un avertissement. Démontrons cela en supprimant artificiellement toutes les instances du mot “April” de nos dates :

```

irs_dates_with_problem <-
  irs_dates_1 %>%
    mutate(start_date_missing = str_replace(start_date_long, "April ", ""))
irs_dates_with_problem

```

```

## # A tibble: 112 × 3
##   village      start_date_long start_date_missing
##   <chr>        <chr>           <chr>
## 1 Mess         April 07 2014   07 2014
## 2 Nkombedzi   April 22 2014   22 2014
## 3 B Compound  May 13 2014    May 13 2014
## 4 D Compound  May 13 2014    May 13 2014
## 5 Post Office May 13 2014    May 13 2014
## 6 Mangulenje May 15 2014    May 15 2014
## 7 Mangulenje Senior May 27 2014 May 27 2014
## 8 Old School   May 27 2014    May 27 2014
## 9 Mwanza       May 28 2014    May 28 2014
## 10 Alumenda    June 18 2014   June 18 2014
## # i 102 more rows

```

Maintenant, essayons de diviser les parties de la date :

```

irs_dates_with_problem %>%
  separate(start_date_missing, into = c
("mois", "jour", "année"), sep = " ")

```

Warning: Expected 3 pieces. Missing pieces filled with `NA` in 3 rows [1, 2, 12].

```

## # A tibble: 112 × 5
##   village      start_date_long mois   jour   année
##   <chr>        <chr>       <chr> <chr> <chr>
## 1 Mess         April 07 2014  07    2014  <NA>
## 2 Nkombedzi   April 22 2014  22    2014  <NA>
## 3 B Compound  May 13 2014   May   13    2014
## 4 D Compound  May 13 2014   May   13    2014
## 5 Post Office May 13 2014   May   13    2014
## 6 Mangulenje  May 15 2014   May   15    2014
## 7 Mangulenje Senior May 27 2014  May   27    2014
## 8 Old School   May 27 2014   May   27    2014
## 9 Mwanza       May 28 2014   May   28    2014
## 10 Alumenda   June 18 2014   June  18    2014
## # i 102 more rows

```

Comme vous pouvez le voir, les lignes manquant de parties produiront des avertissements. Gérez ces avertissements avec soin, car ils peuvent conduire à des données inexactes. Dans ce cas, nous avons maintenant l'information du jour et du mois pour ces lignes dans les mauvaises colonnes.

Q : Division des chaînes de tranches d'âge

Considérez le jeu de données esoph_ca, du package {medicaldata}, qui implique une étude cas-témoin sur le cancer de l'œsophage en France.

```
medicaldata::esoph_ca %>% as_tibble()
```

```

## # A tibble: 88 × 5
##   agegp alcgp   tobgp   ncases ncontrols
##   <ord> <ord>   <ord>     <dbl>      <dbl>
## 1 25–34 0–39g/day 0–9g/day 0        40
## 2 25–34 0–39g/day 10–19   0        10
## 3 25–34 0–39g/day 20–29   0        6
## 4 25–34 0–39g/day 30+     0        5
## 5 25–34 40–79    0–9g/day 0        27
## 6 25–34 40–79    10–19   0        7
## 7 25–34 40–79    20–29   0        4
## 8 25–34 40–79    30+     0        7
## 9 25–34 80–119   0–9g/day 0        2

```

```
## 10 25-34 80-119    10-19      0      1
## # i 78 more rows
```

Divisez les tranches d'âge dans la colonne agegp en deux colonnes distinctes : agegp_inferieur et agegp_superieur.

Après avoir utilisé la fonction `separate()`, le groupe d'âge “75+” nécessitera un traitement spécial. Utilisez `readr::parse_number()` ou une autre méthode pour convertir la limite d'âge inférieure (“75+”) en nombre.

```
medicaldata::esoph_ca %>%
  separate(_____) %>%
  # convertir 75+ en nombre
  mutate(_____)
```

Séparation des Caractères Spéciaux

Pour utiliser la fonction `separate()` sur des caractères spéciaux comme le point (.), nous devons les échapper avec un double antislash (\\\).

Considérez le scénario où les dates sont formatées avec des points :

```
irs_with_period <- irs_dates_1 %>%
  mutate(start_date_long = format(lubridate::mdy(start_date_long), "%d.%m.%Y"))
irs_with_period
```

```
## # A tibble: 112 x 2
##   village           start_date_long
##   <chr>             <chr>
## 1 Mess              07.04.2014
## 2 Nkombedzi         22.04.2014
## 3 B Compound        13.05.2014
## 4 D Compound        13.05.2014
## 5 Post Office       13.05.2014
## 6 Mangulenje        15.05.2014
## 7 Mangulenje Senior 27.05.2014
## 8 Old School         27.05.2014
## 9 Mwanza            28.05.2014
## 10 Alumenda          18.06.2014
## # i 102 more rows
```

Tenter de séparer ce format de date directement avec `sep = ".."` ne fonctionnera pas :

```
irs_with_period %>%
  separate(start_date_long, into = c("day", "month", "year"), sep = ".")
```

```

## # A tibble: 112 × 4
##   village      day month year
##   <chr>     <chr> <chr> <chr>
## 1 Mess        ...  ...
## 2 Nkombedzi  ...  ...
## 3 B Compound ...  ...
## 4 D Compound ...  ...
## 5 Post Office ...  ...
## 6 Mangulenje ...  ...
## 7 Mangulenje Senior ...  ...
## 8 Old School  ...  ...
## 9 Mwanza      ...  ...
## 10 Alumenda   ...  ...
## # i 102 more rows

```

Cela ne fonctionne pas comme prévu car, dans les expressions régulières (regex), le point est un caractère spécial. Nous en apprendrons davantage à ce sujet en temps voulu. La bonne approche consiste à échapper le point en utilisant un double antislash (\) :

```

irs_with_period %>%
  separate(start_date_long, into = c("day", "month", "year"), sep = "\\.")

```

```

## # A tibble: 112 × 4
##   village      day month year
##   <chr>     <chr> <chr> <chr>
## 1 Mess        07  04  2014
## 2 Nkombedzi  22  04  2014
## 3 B Compound 13  05  2014
## 4 D Compound 13  05  2014
## 5 Post Office 13  05  2014
## 6 Mangulenje 15  05  2014
## 7 Mangulenje Senior 27  05  2014
## 8 Old School  27  05  2014
## 9 Mwanza      28  05  2014
## 10 Alumenda   18  06  2014
## # i 102 more rows

```

Maintenant, la fonction comprend qu'elle doit diviser la chaîne à chaque point littéral.

De même, lors de l'utilisation d'autres caractères spéciaux comme +, * ou ?, nous devons également les précédé d'un double antislash (\) dans l'argument sep.

SIDE NOTE



Qu'est-ce qu'un Caractère Spécial ?

SIDE NOTE

Dans les expressions régulières, qui aident à trouver des motifs dans le texte, les caractères spéciaux ont des rôles spécifiques. Par exemple, un point (.) est un caractère générique qui peut représenter n'importe quel caractère. Ainsi, dans une recherche, "do.t" pourrait correspondre à "dolt," "dost," ou "doct" De même, le signe plus (+) est utilisé pour indiquer une ou plusieurs occurrences du caractère précédent. Par exemple, "ho+se" correspondrait à "hose" ou "hooose" mais pas à "hse." Lorsque nous avons besoin d'utiliser ces caractères dans leurs rôles ordinaires, nous utilisons un double antislash (\ \) devant eux, comme "\ \ ." ou "\ \ +." Nous en apprendrons plus sur ces caractères spéciaux dans une leçon future.

Q : Séparation des Caractères Spéciaux

Votre prochaine tâche concerne le jeu de données `hiv_dat_clean_1`. Concentrez-vous sur la colonne `regimen`, qui liste les régimes de médicaments séparés par un signe +. Votre objectif est de diviser cette colonne en trois nouvelles colonnes : `drug_1`, `drug_2` et `drug_3` en utilisant la fonction `separate()`. Faites très attention à la façon dont vous gérez le séparateur +. Voici la colonne :

PRACTICE


(in RMD)

```
hiv_dat_clean_1 %>%
  select(regimen)
```

```
## # A tibble: 1,413 × 1
##   regimen
##   <chr>
##   1 AZT+3TC+NVP
##   2 TDF+3TC+EFV
##   3 TDF+3TC+EFV
##   4 TDF+3TC+EFV
##   5 TDF+3TC+EFV
##   6 AZT+3TC+NVP
##   7 TDF+3TC+EFV
##   8 AZT+3TC+NVP
##   9 AZT+3TC+NVP
##  10 TDF+3TC+EFV
## # i 1,403 more rows
```

Combinaison de Chaînes avec paste()

La fonction `paste()` dans R concatène ou joint ensemble des chaînes de caractères. Cela vous permet de combiner plusieurs chaînes en une seule.

Pour combiner deux chaînes simples :

```
string1 <- "Hello"  
string2 <- "World"  
paste(string1, string2)
```

```
## [1] "Hello World"
```

Le séparateur par défaut est un espace, donc cela renvoie “Hello World”.

Démontrons comment utiliser cela sur un ensemble de données, avec les données de date de l’IRS. D’abord, nous séparerons la date de début en colonnes individuelles :

```
irs_dates_separated <- # stocker pour une utilisation ultérieure  
  irs_dates_1 %>%  
  separate(start_date_long, into = c("month", "day", "year"), sep = " ", remove =  
    FALSE)  
irs_dates_separated
```

```
## # A tibble: 112 × 5  
##   village      start_date_long month day   year  
##   <chr>        <chr>          <chr> <chr> <chr>  
## 1 Mess         April 07 2014   April 07  2014  
## 2 Nkombedzi   April 22 2014   April 22  2014  
## 3 B Compound  May 13 2014    May 13  2014  
## 4 D Compound  May 13 2014    May 13  2014  
## 5 Post Office May 13 2014    May 13  2014  
## 6 Mangulenje  May 15 2014    May 15  2014  
## 7 Mangulenje Senior May 27 2014  May 27  2014  
## 8 Old School   May 27 2014    May 27  2014  
## 9 Mwanza       May 28 2014    May 28  2014  
## 10 Alumenda    June 18 2014   June 18  2014  
## # i 102 more rows
```

Ensuite, nous pouvons recombiner jour, mois et année avec `paste()` :

```
irs_dates_separated %>%  
  select(day, month, year) %>%  
  mutate(start_date_long_2 = paste(day, month, year))
```

```

## # A tibble: 112 × 4
##   day   month year start_date_long_2
##   <chr> <chr> <chr> <chr>
## 1 07   April 2014 07 April 2014
## 2 22   April 2014 22 April 2014
## 3 13   May   2014 13 May  2014
## 4 13   May   2014 13 May  2014
## 5 13   May   2014 13 May  2014
## 6 15   May   2014 15 May  2014
## 7 27   May   2014 27 May  2014
## 8 27   May   2014 27 May  2014
## 9 28   May   2014 28 May  2014
## 10 18  June  2014 18 June 2014
## # i 102 more rows

```

L'argument `sep` spécifie le séparateur entre les éléments. Pour un séparateur différent, comme un trait d'union, nous pouvons é

crire :

```

irs_dates_separated %>%
  mutate(start_date_long_2 = paste(day, month, year, sep = "-"))

```

```

## # A tibble: 112 × 6
##   village      start_date_long month day   year
##   <chr>        <chr>          <chr> <chr> <chr>
## 1 Mess         April 07 2014   April 07  2014
## 2 Nkombedzi   April 22 2014   April 22  2014
## 3 B Compound  May 13 2014    May 13   2014
## 4 D Compound  May 13 2014    May 13   2014
## 5 Post Office May 13 2014    May 13   2014
## 6 Mangulenje  May 15 2014    May 15   2014
## 7 Mangulenje Senior May 27 2014  May 27   2014
## 8 Old School   May 27 2014    May 27   2014
## 9 Mwanza       May 28 2014    May 28   2014
## 10 Alumenda    June 18 2014   June 18   2014
## # i 102 more rows
## # i 1 more variable: start_date_long_2 <chr>

```

Pour concaténer sans espaces, nous pouvons définir `sep = ""` :

```

irs_dates_separated %>%
  select(day, month, year) %>%
  mutate(start_date_long_2 = paste(day, month, year, sep = ""))

```

```

## # A tibble: 112 × 4
##   day   month year start_date_long_2
##   <chr> <chr> <chr> <chr>

```

```

## 1 07 April 2014 07April2014
## 2 22 April 2014 22April2014
## 3 13 May 2014 13May2014
## 4 13 May 2014 13May2014
## 5 13 May 2014 13May2014
## 6 15 May 2014 15May2014
## 7 27 May 2014 27May2014
## 8 27 May 2014 27May2014
## 9 28 May 2014 28May2014
## 10 18 June 2014 18June2014
## # i 102 more rows

```

Où nous pouvons utiliser la fonction `paste0()`, qui est équivalente à `paste(..., sep = "")` :

```

irs_dates_separated %>%
  select(day, month, year) %>%
  mutate(start_date_long_2 = paste0(day, month, year))

```

```

## # A tibble: 112 × 4
##       day   month year start_date_long_2
##   <chr> <chr> <chr> <chr>
## 1 07    April  2014 07April2014
## 2 22    April  2014 22April2014
## 3 13    May    2014 13May2014
## 4 13    May    2014 13May2014
## 5 13    May    2014 13May2014
## 6 15    May    2014 15May2014
## 7 27    May    2014 27May2014
## 8 27    May    2014 27May2014
## 9 28    May    2014 28May2014
## 10 18   June   2014 18June2014
## # i 102 more rows

```

Essayons de combiner `paste()` avec d'autres fonctions de chaîne pour résoudre un problème de données réaliste. Considérez la colonne ID dans le jeu de données `hiv_dat_messy_1` :

```

hiv_dat_messy_1 %>%
  select(patient_id)

## # A tibble: 1,413 × 1
##       patient_id
##   <chr>
## 1 pd-10037
## 2 pd-10537
## 3 pd-5489
## 4 id-5523
## 5 pd-4942
## 6 pd-4742

```

```

## 7 pd-10879
## 8 id-2885
## 9 pd-4861
## 10 pd-5180
## # i 1,403 more rows

```

Imaginez que nous voulions standardiser ces ID pour qu'ils aient le même nombre de caractères. C'est souvent une exigence pour les ID (pensez aux numéros de téléphone, par exemple).

Pour cela, nous pouvons utiliser `separate()` pour diviser les ID en parties, puis utiliser `paste()` pour les recombiner dans un format standardisé.

```

hiv_dat_messy_1 %>%
  select(patient_id) %>% # pour la visibilité
  separate(patient_id, into = c("prefix", "patient_num"), sep = "-", remove = F) %>%
  mutate(patient_num = str_pad(patient_num, width = 5, side = "left", pad = "0")) %>%
  mutate(patient_id_padded = paste(prefix, patient_num, sep = "-"))

```

```

## # A tibble: 1,413 × 4
##       patient_id prefix patient_num patient_id_padded
##   <chr>        <chr>    <chr>          <chr>
## 1 pd-10037     pd      10037      pd-10037
## 2 pd-10537     pd      10537      pd-10537
## 3 pd-5489      pd      05489      pd-05489
## 4 id-5523      id      05523      id-05523
## 5 pd-4942      pd      04942      pd-04942
## 6 pd-4742      pd      04742      pd-04742
## 7 pd-10879     pd      10879      pd-10879
## 8 id-2885      id      02885      id-02885
## 9 pd-4861      pd      04861      pd-04861
## 10 pd-5180      pd      05180      pd-05180
## # i 1,403 more rows

```

Dans cet exemple, `patient_id` est divisé en un préfixe et un numéro. Le numéro est ensuite complété par des zéros pour assurer une longueur cohérente, et enfin, les deux parties sont concaténées à nouveau en utilisant `paste()` avec un trait d'union comme séparateur. Ce processus standardise le format des ID des patients.

Excellent travail !

PRACTICE



(in RMD)

Q : Standardisation des ID dans le Jeu de Données `lima_messy_1`

Dans le jeu de données `lima_messy_1`, les ID ne sont pas complétés par des zéros, ce qui les rend difficiles à trier.

Par exemple, l'ID pe-998 est en haut de la liste après un tri par ordre décroissant, ce qui n'est pas ce que nous voulons.

```
lima_messy_1 %>%
  select(id) %>%
  arrange(desc(id)) # trier par ordre décroissant (les ID les plus
                     élevés devraient être en haut)

## # A tibble: 1,293 × 1
##   id
##   <chr>
## 1 pe-998
## 2 pe-996
## 3 pe-951
## 4 pe-900
## 5 pe-2347
## 6 pe-2337
## 7 pe-2335
## 8 pe-2333
## 9 pe-2331
## 10 pe-2329
## # i 1,283 more rows
```



Essayez de résoudre ce problème en utilisant une procédure similaire à celle utilisée pour hiv_dat_messy_1.

Votre Tâche :

- Séparer l'ID en parties.
- Compléter la partie numérique pour la standardisation.
- Recombiner les parties en utilisant paste().
- Retrier les ID par ordre décroissant. Le plus haut ID devrait se terminer par 2347

```
lima_messy_1 %>%
```



Q : Création de déclarations récapitulatives

l'ensemble de données `irs`. La déclaration doit décrire la couverture de pulvérisation pour chaque village.

Sortie souhaitée : “Pour le village X, la couverture de pulvérisation était de Y % à la date Z.”

PRACTICE



Votre tâche : - Sélectionnez les colonnes nécessaires de l'ensemble de données `irs`. - Utilisez `paste()` pour créer la déclaration récapitulative.

```
irs %>%
  select(village, start_date_default, coverage_p) %>%
```

REMINDER



Au fur et à mesure que nous avançons dans cette leçon, rappelez-vous que l'auto-complétion de RStudio peut vous aider à trouver des fonctions dans le package `stringr`.

Tapez simplement `str_` et une liste de fonctions `stringr` apparaîtra. Toutes les fonctions `stringr` commencent par `str_`.

Ainsi, au lieu d'essayer de toutes les mémoriser, vous pouvez utiliser l'auto-complétion comme référence si nécessaire.

Sous-référencement de chaînes avec `str_sub`

`str_sub` vous permet d'extraire des parties d'une chaîne de caractères en fonction des positions des caractères. La syntaxe de base est `str_sub(chaine, debut, fin)`.

Exemple : Extraction des 2 premiers caractères des identifiants de patients :

```
patient_ids <- c("ID12345-abc", "ID67890-def")
str_sub(patient_ids, 1, 2) # Retourne "ID", "ID"

## [1] "ID" "ID"
```

Ou les 5 premiers :

```
str_sub(patient_ids, 1, 5) # Retourne "ID123", "ID678"
```

```
## [1] "ID123" "ID678"
```

Les valeurs négatives comptent à rebours depuis la fin de la chaîne. Cela est utile pour extraire des suffixes.

Par exemple, pour obtenir les 4 derniers caractères des identifiants de patients.

```
str_sub(patient_ids, -4, -1) # Retourne "-abc", "-def"
```

```
## [1] "-abc" "-def"
```

Assurez-vous de faire une pause et de comprendre ce qui s'est passé ci-dessus.

Lorsque les indices sont en dehors de la longueur de la chaîne, str_sub le gère avec grâce sans erreurs :

```
str_sub(patient_ids, 1, 30) # Retourne en toute sécurité la chaîne complète lorsque  
# la plage dépasse la longueur de la chaîne
```

```
## [1] "ID12345-abc" "ID67890-def"
```

Dans un dataframme, nous pouvons utiliser str_sub dans mutate(). Par exemple, ci-dessous, nous extrayons l'année et le mois de la colonne start_date_default et créons une nouvelle colonne appelée year_month :

```
irs %>%  
  select(start_date_default) %>%  
  mutate(year_month = str_sub(start_date_default, start = 1, end = 7))
```

```
## # A tibble: 112 × 2  
##   start_date_default year_month  
##   <date>            <chr>  
## 1 2014-04-07        2014-04  
## 2 2014-04-22        2014-04  
## 3 2014-05-13        2014-05  
## 4 2014-05-13        2014-05  
## 5 2014-05-13        2014-05  
## 6 2014-05-15        2014-05  
## 7 2014-05-27        2014-05  
## 8 2014-05-27        2014-05  
## 9 2014-05-28        2014-05
```

```
## 10 2014-06-18          2014-06  
## # i 102 more rows
```

Q : Extraction de sous-chaînes d'ID

PRACTICE



(in RMD)

```
hiv_dat_messy_1 %>%  
  select(patient_id) %>%  
  # votre code ici :
```

Conclusion

Félicitations pour avoir atteint la fin de cette leçon ! Vous avez appris sur les chaînes de caractères en R et diverses fonctions pour les manipuler efficacement.

Le tableau ci-dessous offre un rapide récapitulatif des fonctions clés que nous avons abordées. Rappelez-vous, vous n'avez pas besoin de mémoriser toutes ces fonctions. Savoir qu'elles existent et comment les rechercher (comme utiliser Google) est plus que suffisant pour des applications pratiques.

Fonction	Description	Exemple	Résultat de l'exemple
<code>str_to_upper()</code>	Convertir les caractères en majuscules	<code>str_to_upper("hiv")</code>	"HIV"
<code>str_to_lower()</code>	Convertir les caractères en minuscules	<code>str_to_lower("HIV")</code>	"hiv"
<code>str_to_title()</code>	Convertir en majuscules le premier caractère de chaque mot	<code>str_to_title("hiv awareness")</code>	"Hiv Awareness"
<code>str_trim()</code>	Supprimer les espaces au début et à la fin	<code>str_trim(" hiv ")</code>	"hiv"
<code>str_squish()</code>	Supprimer les espaces au début et à la fin et réduire les espaces internes <code>str_squish(" hiv cases ")</code> "hiv cases"		

	Formater une chaîne à une largeur donnée (pour formater la sortie) str_wrap("HIV awareness", width = 5) "HIV"	
str_wrap()	Séparer les éléments d'un vecteur de caractères str_split("Hello- World", "-")	"Hello", "World")
paste()	Concaténer des vecteurs après les avoir convertis en caractères paste("Hello", "World") "Hello World"	
str_sub()	Extraire et remplacer des sous-chaînes d'un vecteur de caractères str_sub("HelloWorld", 1, 4) "Hell"	
separate()	Séparer une colonne de caractères en plusieurs colonnes	<pre>separate(tibble(a = "Hello-World"), a, b c into = c("b", "c"), Hello World sep = "-")</pre>

Notez que bien que ces fonctions couvrent des tâches courantes telles que la standardisation des chaînes, la division et la jonction des chaînes, cette introduction n'effleure que la surface de ce qui est possible avec le package `{stringr}`. Si vous travaillez avec beaucoup de données textuelles brutes, vous voudrez peut-être explorer davantage sur le site web [stringr](#).

Clés de Réponses

Q : Identification d'Erreurs dans les Définitions de Chaînes

1. **ex_a**: Correct.
2. **ex_b**: Correct.
3. **ex_c**: Erreur. Version corrigée : ex_c <- "They've been \"best friends\" for years." 4

. **ex_d**: Erreur. Version corrigée : ex_d <- 'Jane\'s diary' 5. **ex_e**: Erreur. Guillemet de fermeture manquant. Version corrigée : ex_e <- "It's a sunny day!"

Q : Nettoyage des Données de Noms de Patients

```
patient_names <- c(" john doe", "ANNA SMITH ", "Emily Davis")  
patient_names <- str_trim(patient_names) # Supprimer les espaces  
patient_names <- str_to_title(patient_names) # Convertir en casse de titre
```

Q : Standardisation des Codes de Médicaments

```
drug_codes <- c("12345", "678", "91011")  
# Remplir chaque code avec des zéros à gauche pour une largeur fixe de 8 caractères.  
drug_codes_padded <- str_pad(drug_codes, 8, pad = "0")
```

Q : Formatage des Instructions Médicales

```
instructions <- "Prenez deux comprimés quotidiennement après les repas. Si les symptômes persistent plus de trois jours, consultez immédiatement votre médecin. Ne prenez pas plus que la dose recommandée. Tenir hors de portée des enfants."  
# Formater les instructions  
wrapped_instructions <- str_wrap(instructions, width = 50)  
  
ggplot(data.frame(x = 1, y = 1), aes(x, y, label = wrapped_instructions)) +  
  geom_label() +  
  theme_void()
```

Q : Formatage d'un Jeu de Données sur la Tuberculose

Les étapes pour nettoyer le jeu de données lima_messy comprendraient :

```
lima_clean <- lima_messy %>%  
  mutate(  
    marital_status = str_squish(str_to_title(marital_status)), # Nettoyer et standardiser marital_status  
    sex = str_squish(str_to_upper(sex)), # Nettoyer et standardiser sex  
    age = str_squish(str_to_lower(age)) # Nettoyer et standardiser age  
)  
  
lima_clean %>%  
  select(marital_status, sex, age) %>%  
 tbl_summary()
```

Ensuite, utilisez la fonction `tbl_summary()` pour créer le tableau récapitulatif.

Q : Formatage des Étiquettes d'Axe dans un Graphique

```
# En supposant que lima_clean est déjà créé et contient marital_status
ggplot(lima_clean, aes(x = str_wrap(marital_status, width = 15))) +
  geom_bar() +
  labs(x = "État Civil")
```

Q : Séparation des Chaînes de Plages d'Âge

```
esoph_ca %>%
  select(agegp) %>% # pour illustration
  separate(agegp, into = c("agegp_lower", "agegp_upper"), sep = "-") %>%
  mutate(agegp_lower = readr::parse_number(agegp_lower))
```

Q : Création de Déclarations Sommaires

```
irs %>%
  select(village, start_date_default, coverage_p) %>%
  mutate(summary_statement = paste0("Pour le village ", village, ", la couverture de
    pulvérisation était de ", coverage_p, "% le ", start_date_default))
```

Q : Extraction de Sous-chaînes d'ID

```
hiv_dat_messy_1 %>%
  select(patient_id) %>%
  mutate(numeric_part = str_sub(patient_id, 4))
```

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

Les factors dans R

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.csv"))
```

```
## # 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
## # i 21 more variables: birth_month <chr>, birth_day <dbl>,
## #   death_year <dbl>, death_month <chr>, death_day <dbl>, ...
```

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

- **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.

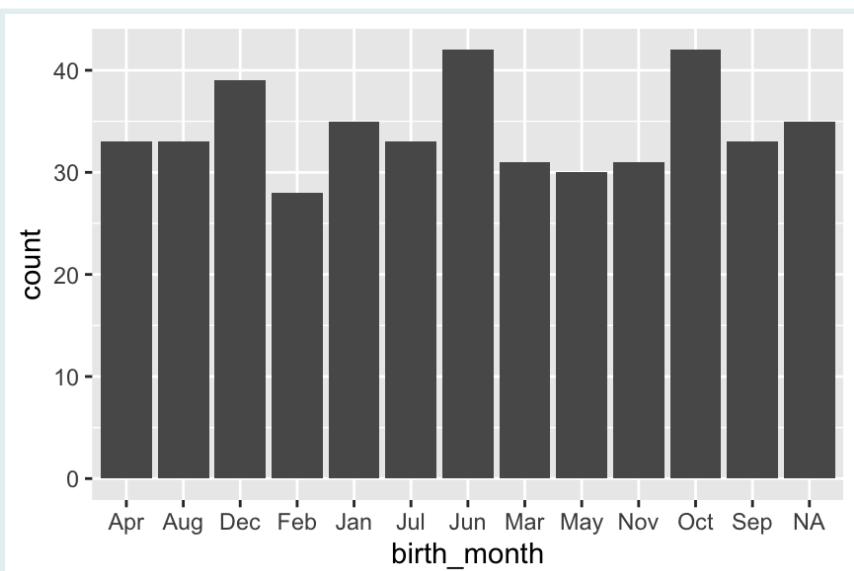
Ce dernier point, le contrôle de l'ordre des niveaux de facteurs, sera notre objectif principal.

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)
```

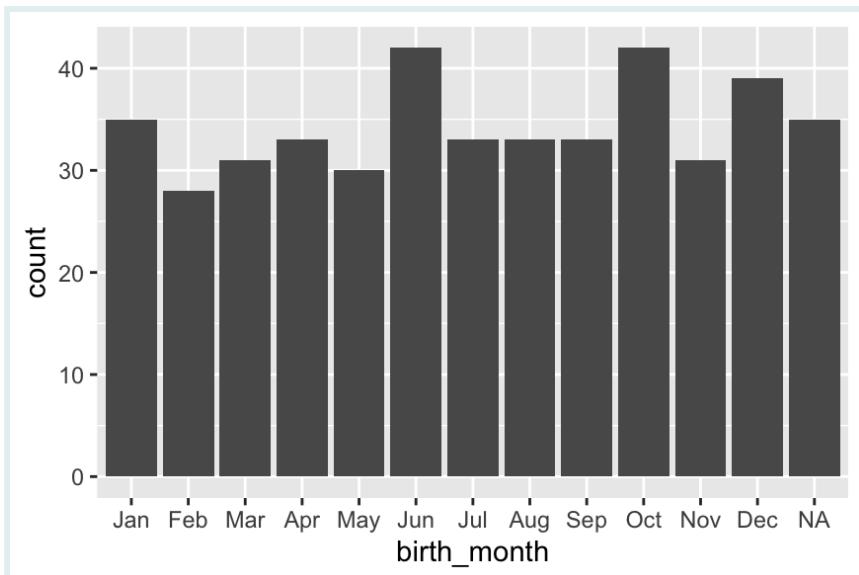
```
## [1] "factor"
```

```
# 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



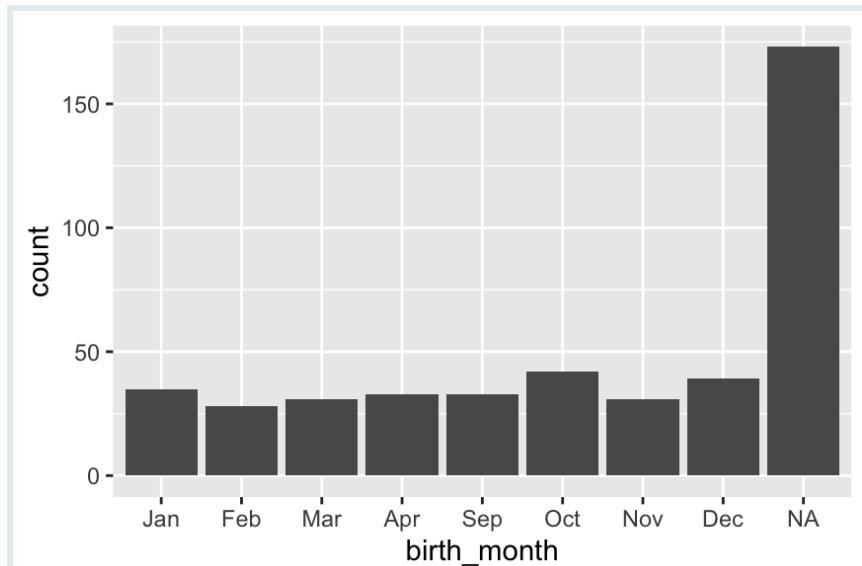
```
hiv_mort_missing_months <-
  hiv_mort %>%
  mutate(birth_month = factor(x = birth_month,
                             levels = c("Jan", "Feb", "Mar",
"Apr",
                               # missing months
"Sep", "Oct", "Nov",
"Dec")))
```

Nous finissons avec beaucoup de valeurs NA :

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



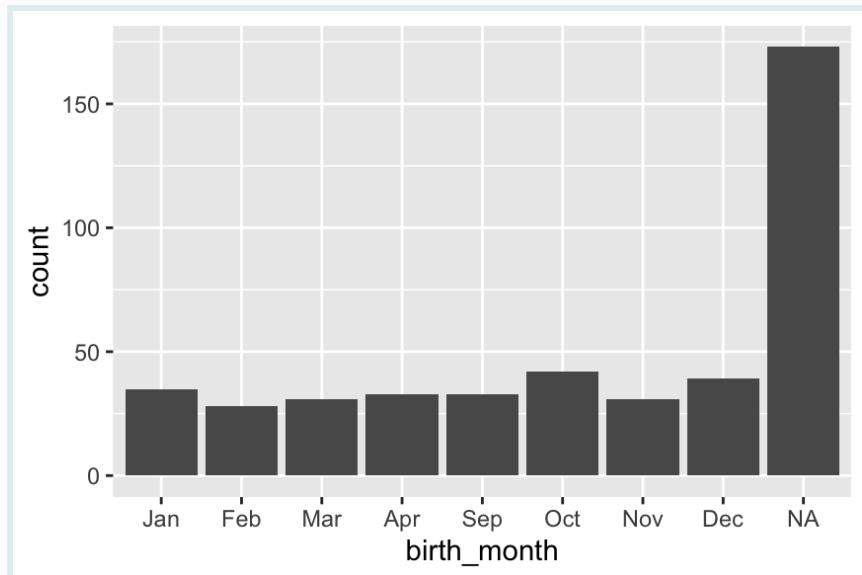
WATCH OUT



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",
"Aog", # typos
"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" "May"
## [9] "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`

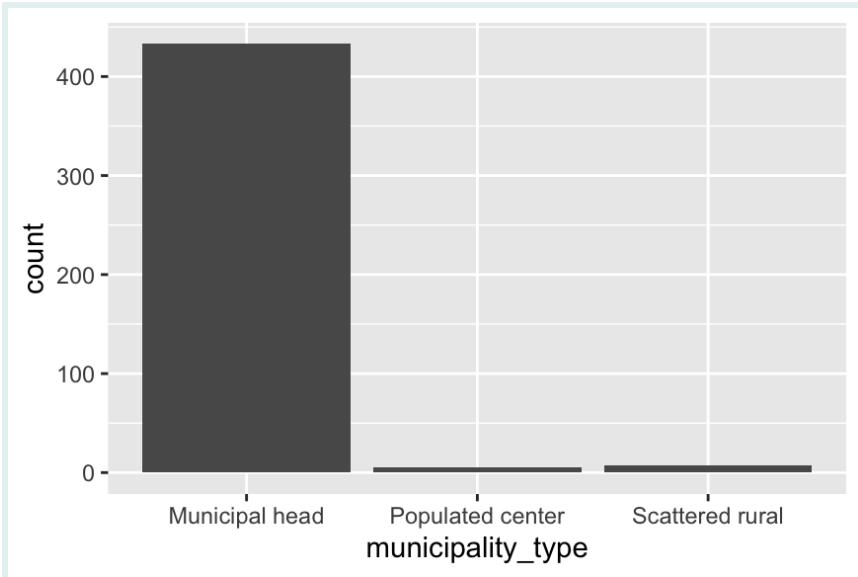
Les facteurs sont très utiles, mais ils peuvent parfois être un peu fastidieux à manipuler uniquement avec les fonctions de base R. Heureusement, le package `forcats`, membre du tidyverse, propose un ensemble de fonctions qui simplifient grandement la manipulation des facteurs. Nous allons examiner quatre 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 ?

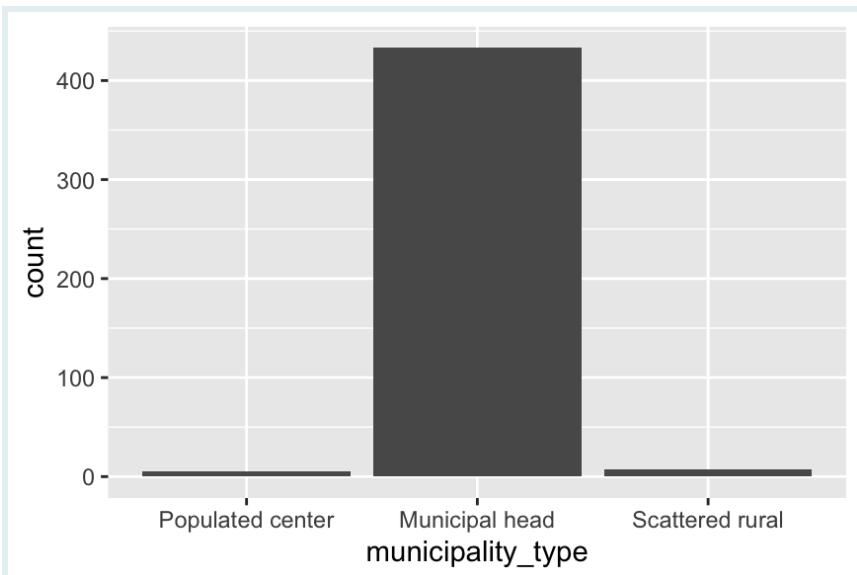
Cela peut être réalisé en utilisant `fct_relevel()`. Voici comment :

```
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 :

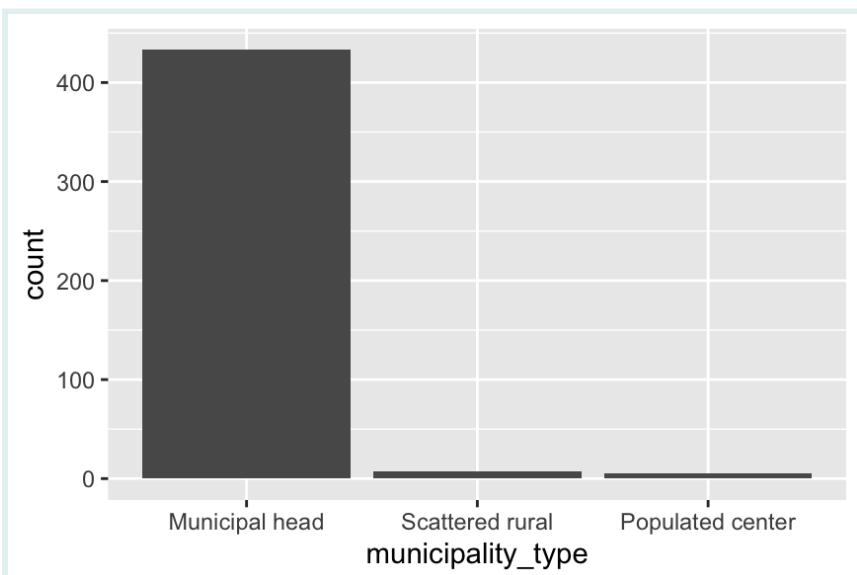
```
ggplot(hiv_mort_pop_center_first) +
  geom_bar(aes(x = municipality_type))
```



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))
```

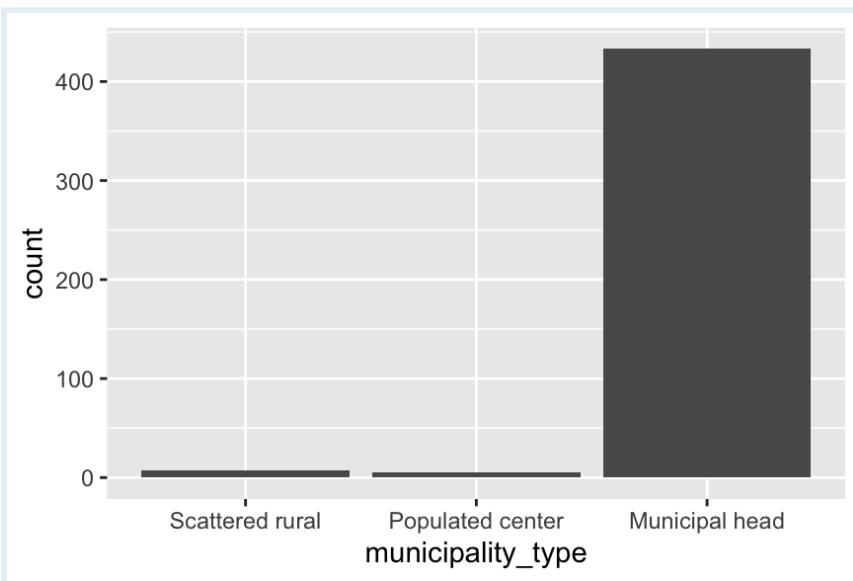


La syntaxe est : spécifier le facteur, le niveau à déplacer, et utiliser l'argument `after` pour définir à quelle position le placer après.

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 med_age_death  
##   <chr>                <int>           <dbl>           <dbl>  
## 1 Aguadas                  2             42              42  
## 2 Anserma                 15            37.4            37.5  
## 3 Aranzazu                 2             37.5            37.5  
## 4 Belalcázar                4             38.8            41  
## 5 Chinchiná                 62            43.6            42.5  
## 6 Filadelfia                 5             42.6            43  
## 7 La Dorada                 46            41.0            41  
## 8 La Merced                 3              27              28  
## 9 Manizales                 199            41.0            41  
## 10 Manzanares                3             38.3            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 :

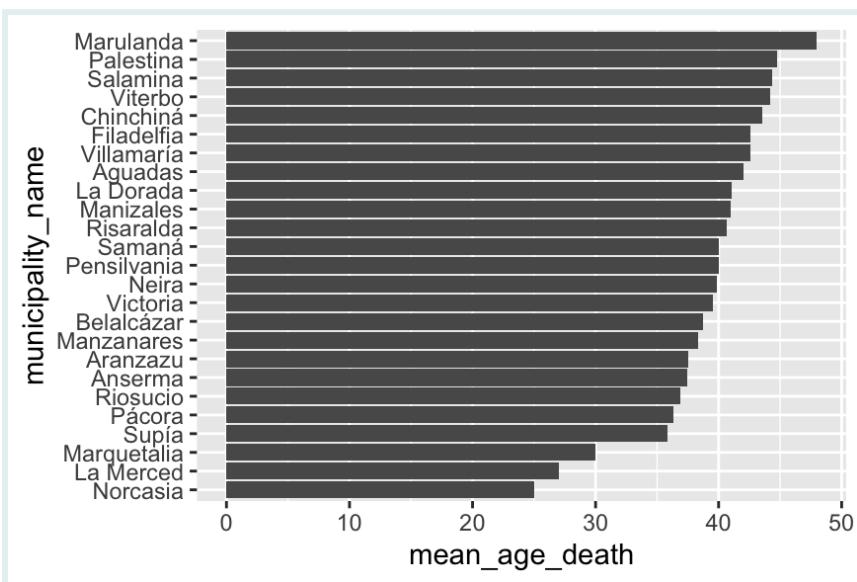
```
summary_per_muni_reordered <-  
  summary_per_muni %>%  
  mutate(municipality_name = fct_reorder(.f = municipality_name,  
                                         .x = mean_age_death))
```

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))
```



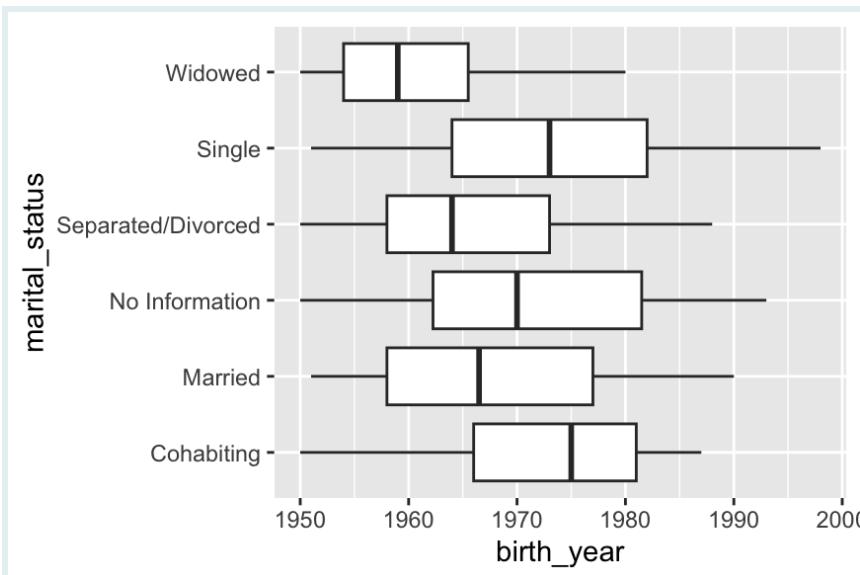
Q: Utiliser fct_reorder

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).

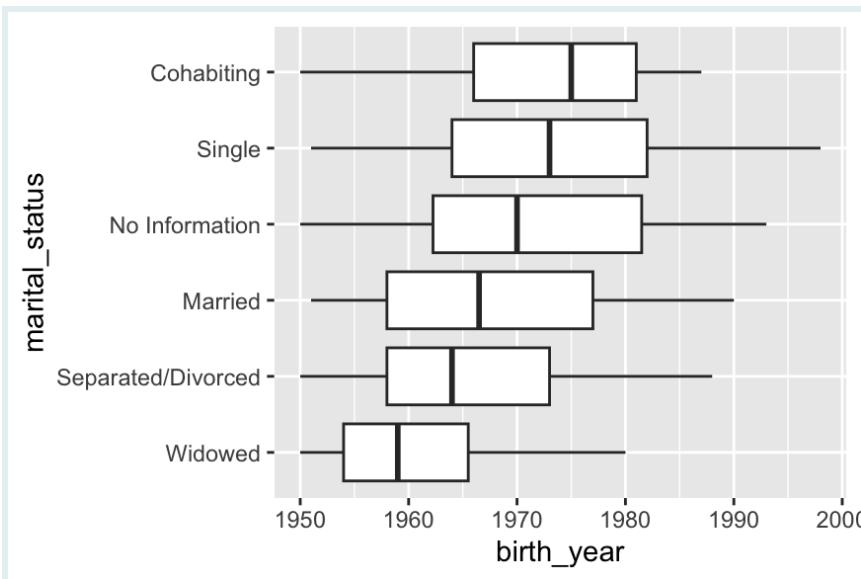
C'est là que intervient l'argument .fun de fct_reorder(). L'argument .fun nous permet de spécifier une fonction de résumé qui sera utilisée pour calculer le nouvel ordre des niveaux :

```
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

PRACTICE



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 change
levels(hiv_mort_muni_recode$municipality_type)

```

```
## [1] "City"           "Populated center" "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
##   <chr>             <int>
```

```
## <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.

SIDE NOTE 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"
```

Si nous créons ensuite un tableau ou un graphique, notre ordre est perturbé et revient à l'ordre alphabétique, avec "other" comme premier niveau :

```
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"))
```

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

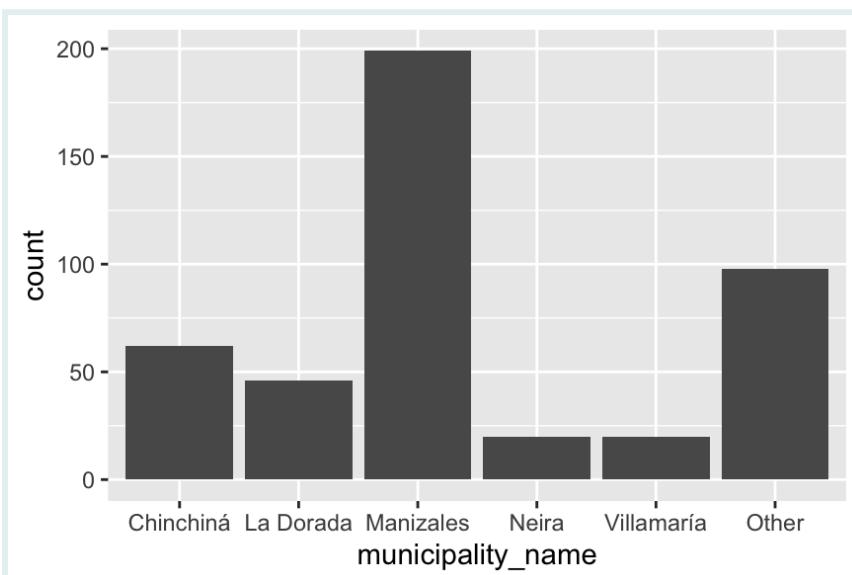
Parfois, nous avons trop de niveaux pour un tableau d'affichage ou un graphique, et nous voulons regrouper les niveaux les moins fréquents dans une seule catégorie, généralement appelée ‘Other’.

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()
```

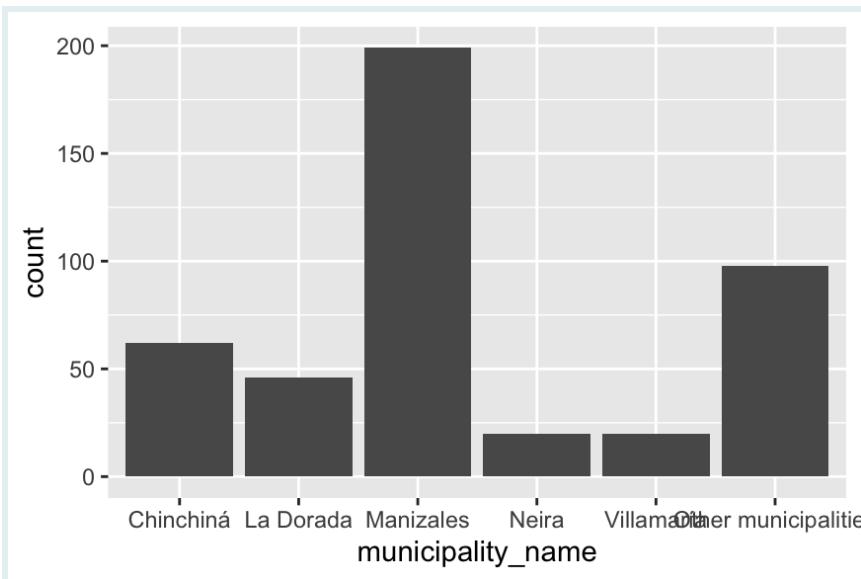


Dans l'utilisation ci-dessus, le paramètre `n = 5` signifie que les cinq municipalités les plus fréquentes sont conservées, et le reste est regroupé dans 'Other'.

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

- b. 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

Dates 1: Reconnaître et Savoir Formatter des Dates

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
## # i 5 more variables: date_debut_defaut <date>,
## #   date_fin_defaut <date>, date_debut_typique <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
## # i 3 more variables: date_adm_abrege <chr>,
## #   date_adm_mixte <chr>, date_sortie_defaut <date>
```

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 :

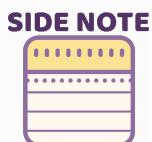
```
as.Date("12/31/2000")
as.Date("31 dec, 2000")
```

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
## # ... i 102 more rows
## # ... i 2 more variables: date_debut_longue <chr>, ...
```

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
↳ <date>↳          ↳ <date>↳          <chr>
1 2014-04-07        2014-04-17        07/04/2014
2 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 0007-04-20
## 2 0022-04-20
## 3 0013-05-20
## 4 0013-05-20
## 5 0013-05-20
```

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 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
```

```
## # i 5 more variables: date_debut_defaut <date>,
## #   date_fin_defaut <date>, date_debut_typique <date>, ...
```

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")
```

```
## [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)
```

```
## # 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 jan 2000", "07/30/2001"), orders=c("dmy", "mdy"))
```

```
## [1] "2000-01-01 UTC" "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 jan 2000", "07/30/2001"), orders=c("mdy", "dmy"))
```

```
## [1] "2000-01-01 UTC" "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")))
```

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")))
```

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 reconvertisant 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("Juin 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:



AMANDA MCKINLEY

R Developer and Instructor, the GRAPH Network



KENE DAVID NWOSU

Data analyst, the GRAPH Network

Passionate about world improvement



GUY WAFFEU

R Instructor and Public Health Physician

Committed to improving the quality of data analysis

Dates 2 : Intervalles, Composantes et Arrondissement

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
Conclusion
Corrigé

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!

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)
```

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).

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

```
## # 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  
## # i 5 more variables: start_date_default <date>,  
## #   end_date_default <date>, start_date_typical <chr>, ...
```

Le deuxième jeu de données contient des données mensuelles de 2015 à 2019 comparant l'incidence moyenne du paludisme pour 1000 personnes dans les villages ayant reçu des pulvérisations intradomiciliaires d'insecticide rémanent (PIIR) par rapport aux villages qui n'en ont pas reçu.

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

```
## # 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
```

La colonne `ir_case` montre l'incidence du paludisme dans les villages avec PIIR et `ir_control` montre l'incidence dans les villages sans PIIR. La colonne `date` contient le mois et un jour aléatoire. Le jeu de données inclut également les températures minimales et maximales mensuelles (`avg_min` et `avg_max`).

Le jeu de données final contient 1460 lignes de données météorologiques quotidiennes pour la région d'Iollovo de 2015 à 2019.

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

```
## # 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   1.0
## 5 2015-01-05    20.0    32.2  53.0
```

Chaque ligne représente un seul jour et inclut des mesures de température minimale (`min_temp`) en degrés Celsius, de température maximale (`max_temp`) en degrés Celsius, et de précipitations (`rain`) en millimètres.

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`. Cela s'appelle parfois l'opérateur d'intervalle. 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
```

Cela signifie qu'il y avait six périodes de 5 jours dans cet intervalle. ::: r-practice

Lubridate weeks

Utilisez la fonction `weeks()` à la place de `days()` dans la méthode lubridate pour calculer la différence de temps en semaines entre les deux dates ci-dessous :

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

:::

Comparaison

Alors, quelle méthode est la meilleure ? Lubridate offre plus de flexibilité et de précision lorsqu'on travaille avec des dates dans R. Regardons un exemple simple pour comprendre pourquoi.

Commençons par définir deux dates espacées de 6 ans :

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

Si nous voulons calculer combien d'années se sont écoulées entre ces dates, comment procéderions-nous avec R de base ? Nous pourrions d'abord soustraire les deux dates, `date_2 - date_1`, puis diviser par un nombre moyen de jours, comme 365,25 (en tenant compte des années bissextiles) :

```
(date_2 - date_1)/365.25
```

```
## Time difference of 6.001369 days
```

Le résultat est proche de 6 ans mais pas précis!

SIDE NOTE



Le résultat est toujours donné en "jours". Vous pouvez facilement convertir la différence de temps en une valeur numérique :

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

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

Diviser par 365 ou 366 donnera également des résultats imprécis :

```
(date_2 - date_1)/365
```

```
## Time difference of 6.005479 days
```

```
(date_2 - date_1)/366
```

```
## Time difference of 5.989071 days
```

Ce que vous devez faire est de prendre en compte qu'il y a seulement deux années bissextiles (deux jours supplémentaires) entre ces dates et de soustraire d'abord ces deux jours :

```
(date_2 - date_1 - 2)/365
```

```
## Time difference of 6 days
```

Mais ce sera une chose pénible à faire en pratique lorsqu'on travaille avec des données réelles. Avec les intervalles de lubridate, le processus est plus simple, car les années

bissextils sont prises en compte pour vous :

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

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

La différence est légère, mais lubridate est clairement le gagnant dans cette situation.

Bien que nous ne les couvrions pas dans ce cours, {lubridate} est également excellent pour gérer les irrégularités temporelles telles que les fuseaux horaires et les changements d'heure d'été.

Lubridate intervals

Pouvez-vous appliquer la fonction d'intervalle de lubridate à notre ensemble de données IRS ? Créez une nouvelle colonne appelée “spraying_time” et, en utilisant l'opérateur %--% de lubridate, calculez le nombre de jours entre start_date_default et end_date_default.

::: side- note Lubridate distingue techniquement entre les périodes et les durées. Vous pouvez en savoir plus [ici](#) :::

Note: I've kept the code unchanged since code is language-independent.

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 :

```
irs %>%
  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`.

```
irs %>%
  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()`.

```
irs %>%
  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
```

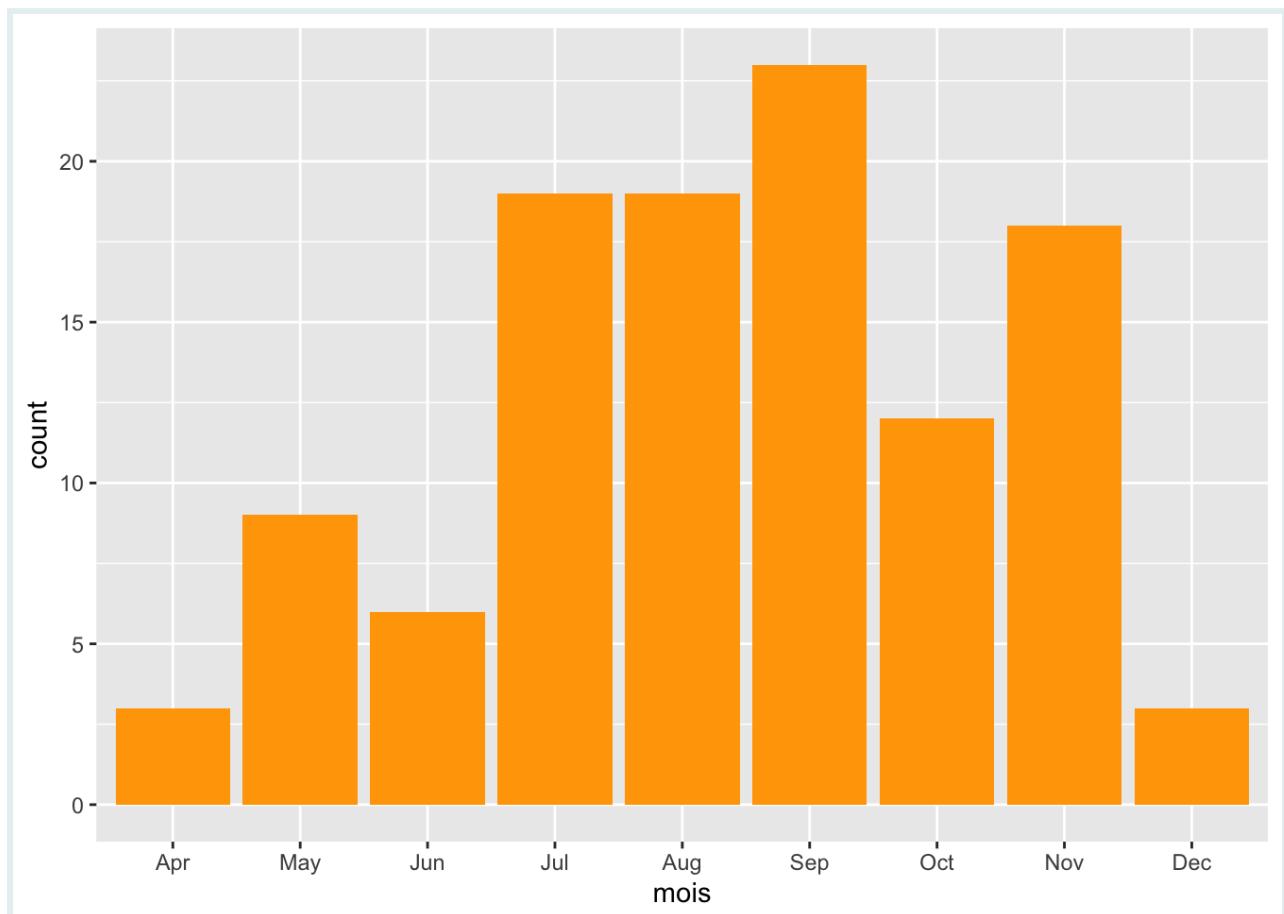
Extraction des jours de la semaine

Créez une nouvelle variable appelée `jour_semaine_debut` 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 des raisons pour lesquelles vous pourriez vouloir extraire des composants de date spécifiques est lorsque vous voulez visualiser vos données.

Par exemple, disons que nous voulons visualiser les mois où les pulvérisations commencent, nous pouvons le faire en créant une nouvelle variable `mois` avec la fonction `month()`, et en traçant un graphique à barres avec `geom_bar`.

```
irs %>%
  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, aucune n'ayant lieu durant les trois premiers mois de l'année. Les auteurs de l'article dont ces données sont issues ont déclaré que les campagnes de pulvérisation visaient à se terminer juste avant la saison des pluies (novembre-avril) au Malawi. Cela correspond au schéma observé.

Visualisation des mois de fin d'arrosage

Utilisez le data frame `irs` pour créer un nouveau graphique montrant les mois de fin des campagnes de pulvérisation. Comparez ce graphique à celui du début des campagnes. Est-ce que les motifs sont similaires ?

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 <- as.Date("2012-03-17")
floor_date(ma_date, 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 <- as.Date("2020-01-03")
ceiling_date(ma_date, 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.

Pratique de l'arrondi des dates

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 :

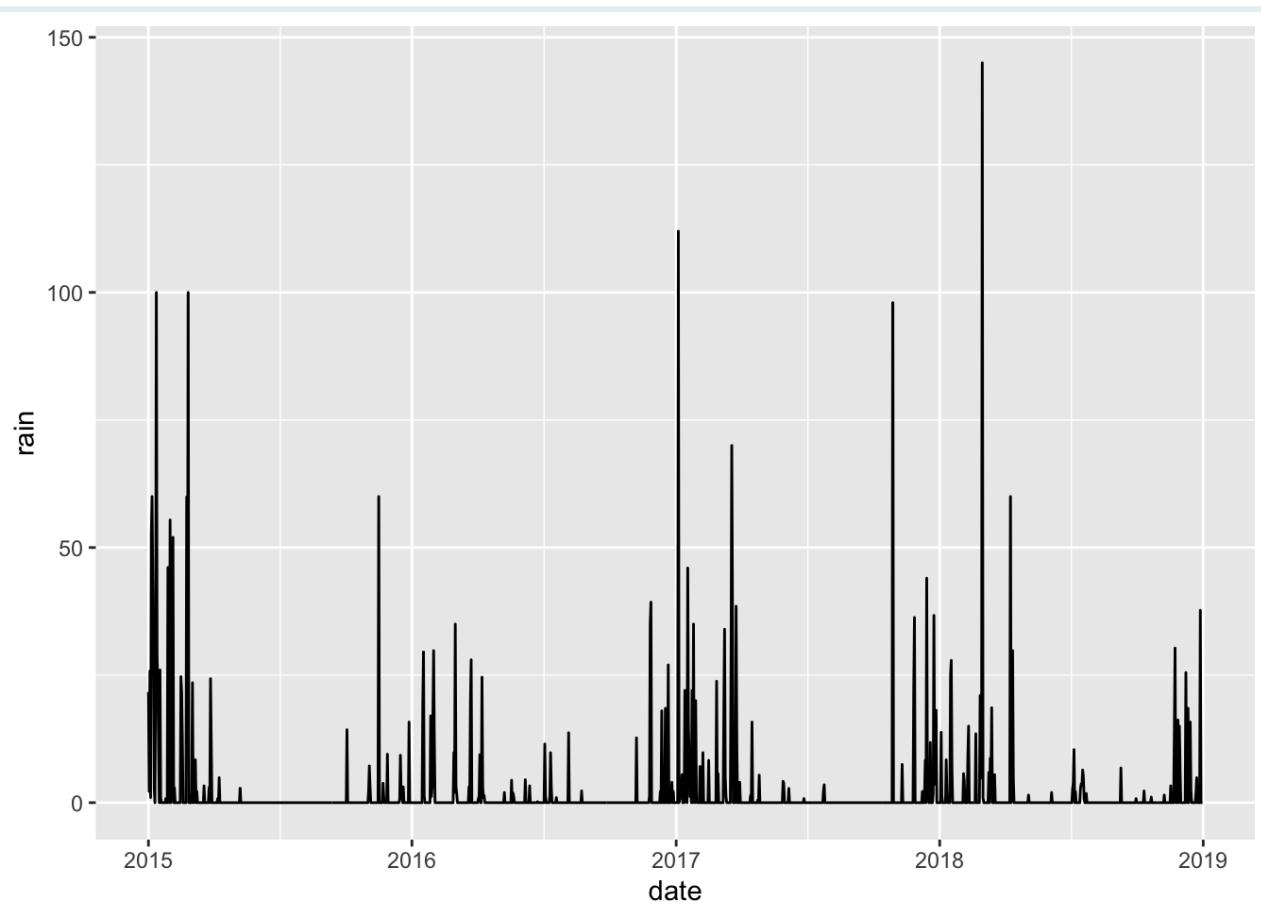
```
date_arroundie <- as.Date("2001-11-29")
floor_date(date_arroundie, unit="year")
```

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.

```
meteo
```

Comme nous pouvons le voir, nos données météorologiques sont enregistrées quotidiennement, mais ce niveau de détail n'est pas idéal pour étudier comment les tendances météorologiques affectent la transmission du paludisme, qui suit un modèle saisonnier. Les données météorologiques quotidiennes peuvent être assez bruitées étant donné la variation significative d'un jour à l'autre :

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



En plus d'être visuellement en désordre, il est un peu difficile de voir les motifs saisonniers. L'agrégation mensuelle est une approche plus efficace pour capturer les variations saisonnières.

Si nous voulions tracer la moyenne des précipitations mensuelles, notre première tentative pourrait être d'utiliser la fonction `str_sub()` pour extraire les sept premiers caractères de notre date (le composant mois et année).

```
meteo %>%
  mutate(mois_annee=str_sub(date, 1, 7))
```

Ensuite, nous regroupons par `mois_annee` pour calculer la moyenne des précipitations pour chaque mois :

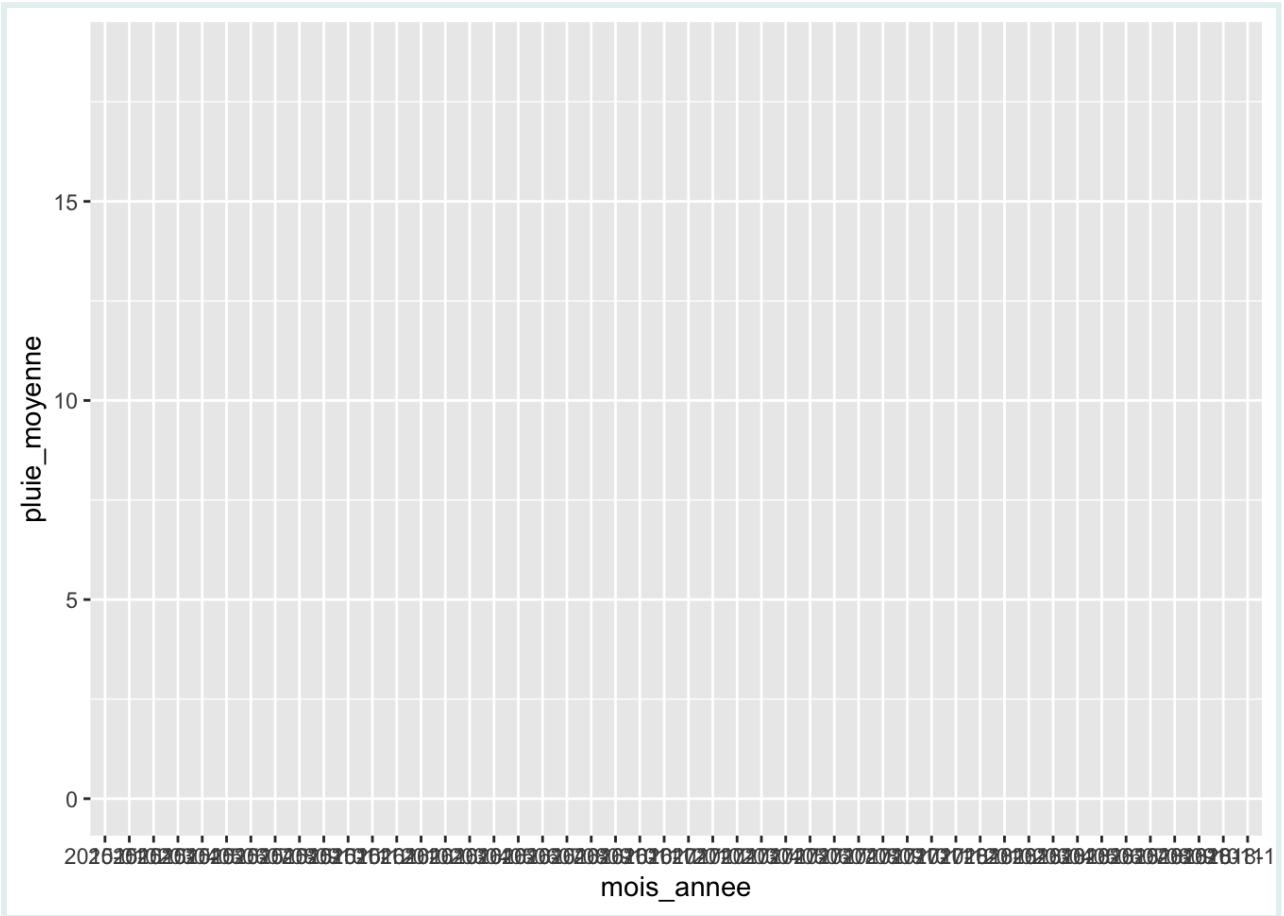
```
resume_meteo_1 <-
  weather %>%
  mutate(mois_annee=str_sub(date, 1, 7)) %>%
  # regrouper et résumer
  group_by(mois_annee) %>%
  summarise(pluie_moyenne=mean(rain))
resume_meteo_1
```

```
## # A tibble: 48 × 2
##   mois_annee    pluie_moyenne
##   <chr>          <dbl>
## 1 2015-01        18.6
## 2 2015-02        10.4
## 3 2015-03        2.69
## 4 2015-04        0.19
## 5 2015-05        0.0968
## 6 2015-06        0
## 7 2015-07        0
## 8 2015-08        0
## 9 2015-09        0
## 10 2015-10       0.461
## # i 38 more rows
```

Cependant, nous rencontrons un problème lors de la tentative de tracer ces données. Notre variable `mois_annee` est maintenant un caractère, et non une date. Cela signifie qu'elle n'est pas continue. Tracer un graphique linéaire avec une variable non continue ne fonctionne pas :

```
resume_meteo_1 %>%
  ggplot() +
  geom_line(aes(mois_annee, pluie_moyenne))
```

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



La meilleure façon de procéder est d'abord d'arrondir nos dates au mois en utilisant la fonction `floor_date()`, puis de regrouper nos données par notre nouvelle variable `mois_annee`, et ensuite de calculer la moyenne mensuelle. Essayons-le maintenant.

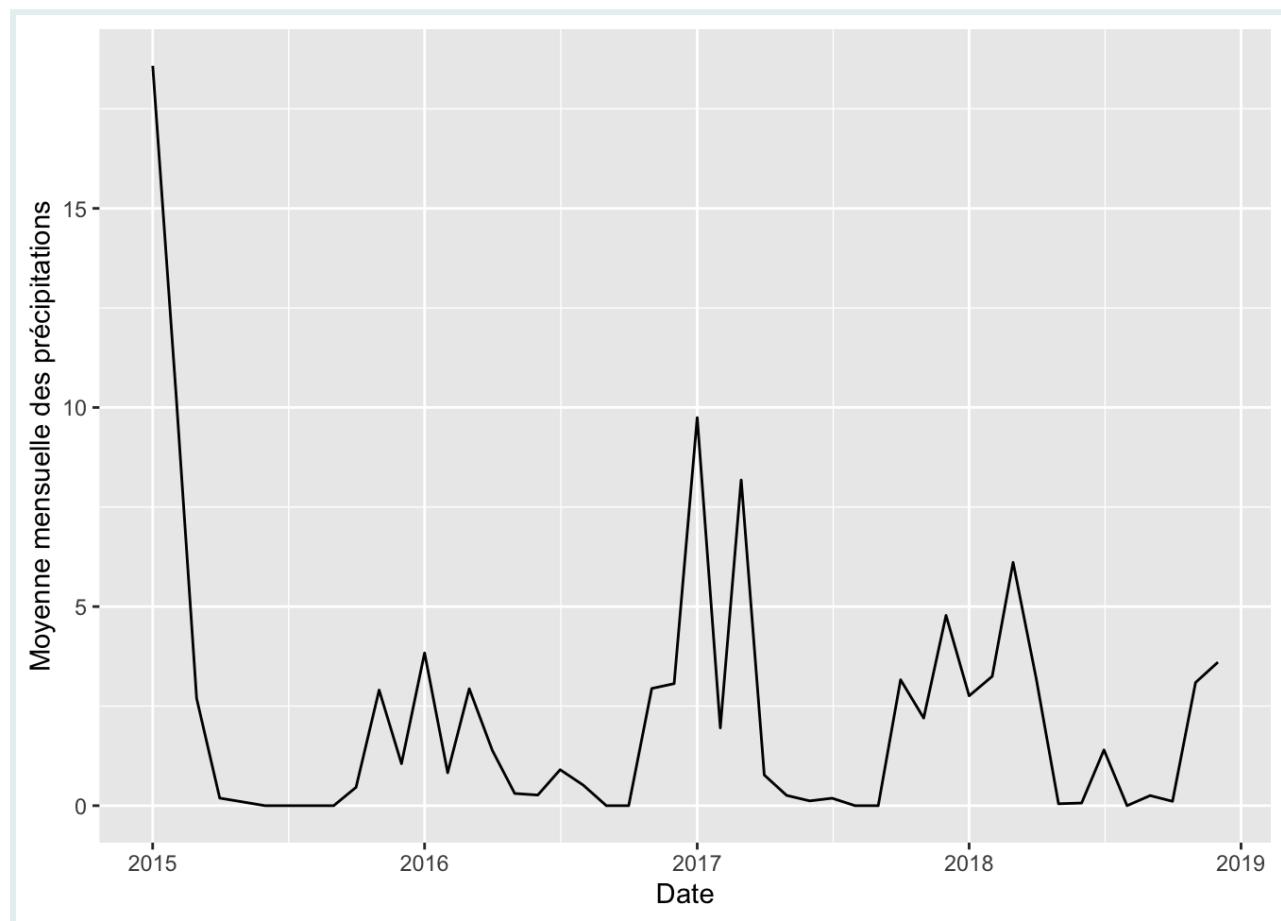
```
resume_meteo_2 <- weather %>%
  mutate(mois_annee=floor_date(date, unit="month")) %>%
  group_by(mois_annee) %>%
  summarise(pluie_moyenne=mean(rain))
resume_meteo_2
```

```
## # A tibble: 48 × 2
##   mois_annee    pluie_moyenne
##   <date>           <dbl>
## 1 2015-01-01     18.6
## 2 2015-02-01     10.4
## 3 2015-03-01      2.69
## 4 2015-04-01      0.19
## 5 2015-05-01     0.0968
## 6 2015-06-01      0
## 7 2015-07-01      0
## 8 2015-08-01      0
## 9 2015-09-01      0
```

```
## 10 2015-10-01      0.461
## # i 38 more rows
```

Maintenant, nous pouvons tracer nos données et nous aurons un graphique de la moyenne des précipitations mensuelles sur la période de 4 ans.

```
resume_meteo_2 %>%
  ggplot() +
  geom_line(aes(mois_annee, pluie_moyenne)) +
  labs(x="Date", y="Moyenne mensuelle des précipitations")
```



Cela semble bien meilleur ! Nous obtenons maintenant une image beaucoup plus claire des tendances saisonnières et des variations annuelles.

Tracer les températures minimales et maximales mensuelles moyennes

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

Conclusion

Cette leçon a couvert les compétences fondamentales pour travailler avec des dates en R - calculer des intervalles, extraire des composantes, arrondir et créer des visualisations de séries temporelles. Avec ces blocs de construction clés maintenant maîtrisés, vous pouvez désormais commencer à manipuler des données de date pour découvrir et analyser des modèles au fil du temps.

Corrigé

Lubridate weeks

```
oct_31 <- as.Date("2023-10-31")
jul_20 <- as.Date("2023-07-20")
difference_temps <- oct_31 %--% jul_20
difference_temps/weeks(1)
```

```
## [1] -14.71429
```

Intervalles avec Lubridate

```
irs %>%
  mutate(temp_arrosage = interval(start_date_default,
  end_date_default)/days(1)) %>%
  select(temp_arrosage)
```

```
## # A tibble: 112 × 1
##       temp_arrosage
##   <dbl>
## 1      10
## 2       5
## 3       0
## 4       0
## 5       0
## 6      11
## 7       0
## 8       0
## 9      19
## 10      9
## # ... with 102 more rows
```

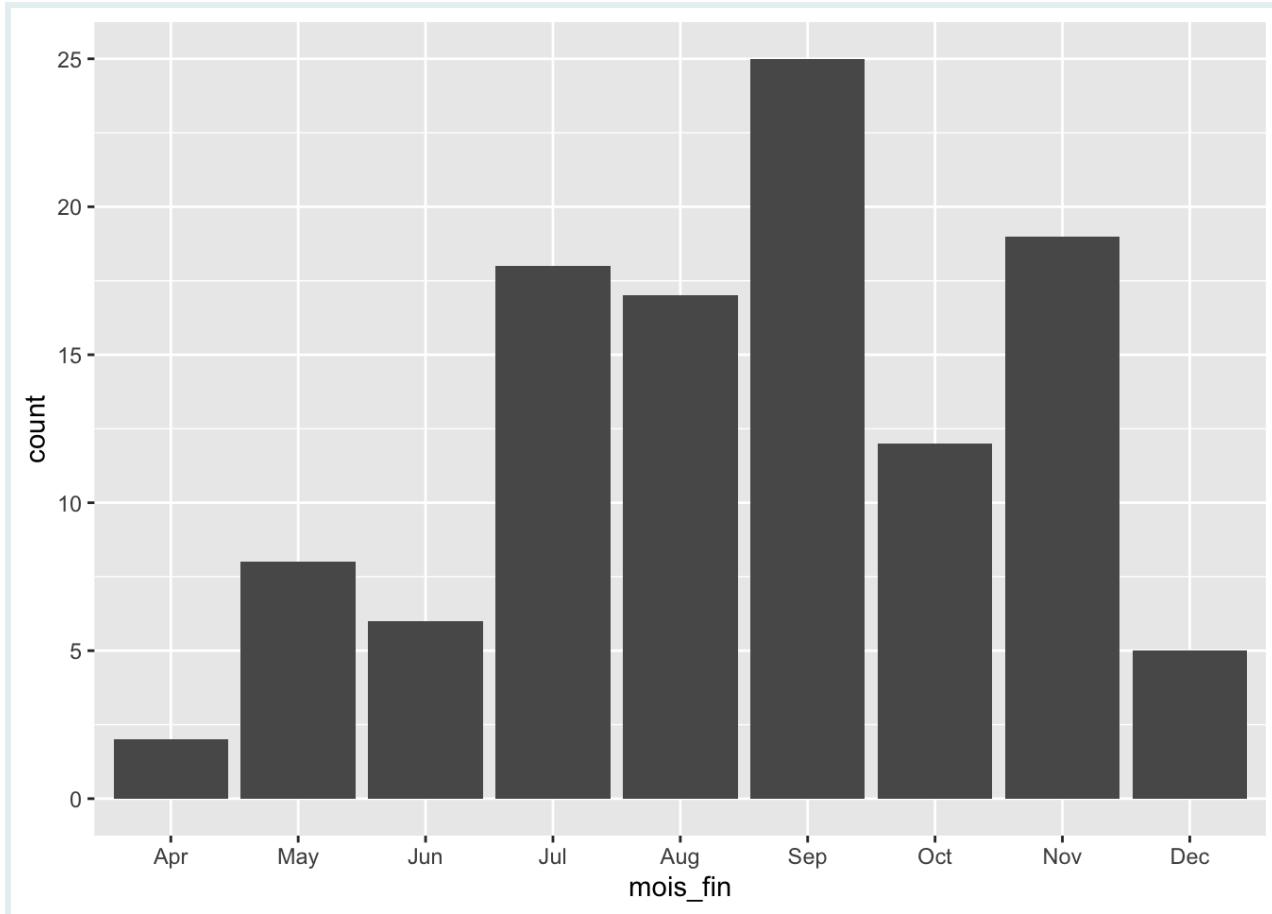
Extraction des jours de la semaine

```
irs %>%
  mutate(jour_semaine_debut = wday(start_date_default, label = TRUE)) %>%
  select(jour_semaine_debut)
```

```
## # A tibble: 112 × 1
##   jour_semaine_debut
##   <ord>
##     1 Mon
##     2 Tue
##     3 Tue
##     4 Tue
##     5 Tue
##     6 Thu
##     7 Tue
##     8 Tue
##     9 Wed
##    10 Wed
## # i 102 more rows
```

Visualisation des mois de fin d'arrosage

```
irs %>%
  mutate(mois_fin = month(end_date_default, label = TRUE)) %>%
  ggplot(aes(x = mois_fin)) +
  geom_bar()
```



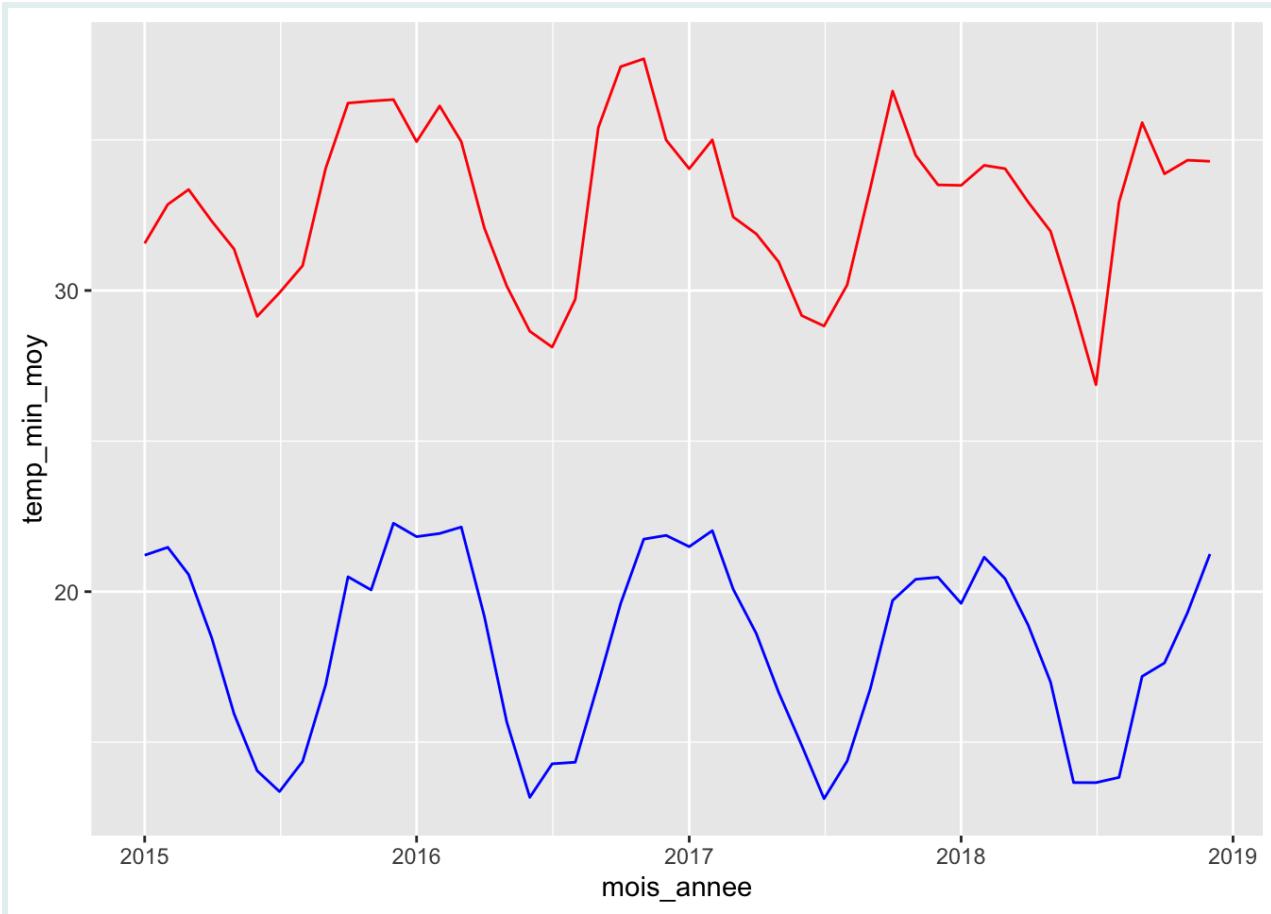
Pratique de l'arrondi des dates

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

```
## [1] "2001-01-01"
```

Tracer les températures minimales et maximales mensuelles moyennes

```
weather %>%
  mutate(mois_annee = floor_date(date, unit="month")) %>%
  group_by(mois_annee) %>%
  summarise(temp_min_moy = mean(min_temp),
            temp_max_moy = mean(max_temp)) %>%
  ggplot() +
  geom_line(aes(x = mois_annee, y = temp_min_moy), color = "blue") +
  geom_line(aes(x = mois_annee, y = temp_max_moy), color = "red")
```



Contributors

The following team members contributed to this lesson:



AMANDA MCKINLEY

R Developer and Instructor, the GRAPH Network



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement



GUY WAFFEU

R Instructor and Public Health Physician
Committed to improving the quality of data analysis

Joindre des tables de données

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
Answer Key

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`: "left", "right", "inner" et "full".
- 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,
               here,
               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, `demographique`, contient les noms et les âges de trois patients :

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

```
## # 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(demographique, 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 !

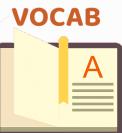
Un troisième problème se pose lorsqu'une entité apparaît plus d'une fois dans un ensemble de données. Peut-être qu'Alice a fait plusieurs tests de TB :

```
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 !

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

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

SIDE NOTE



```
## 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 rencontrerons une erreur :

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

```
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 datafram 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 datafram x avec destinataire_test du datafram y parce qu'ils représentent les mêmes données.”

Q : `left_join()` entre patients et controles {.unlisted}

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.

PRACTICE



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

PRACTICE

(in RMD)

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

PRACTICE

(in RMD)

```
# 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

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 :

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

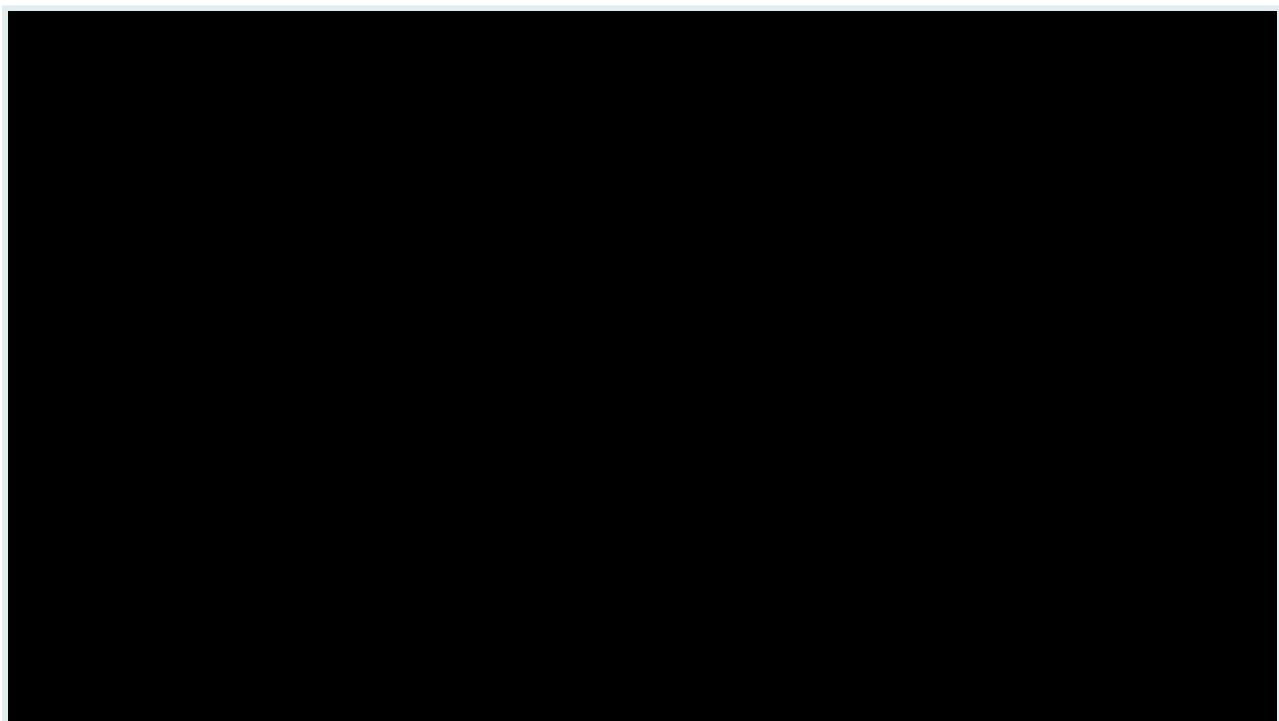
Si nous effectuons un `left_join()` en utilisant `demographique` comme ensemble de données de gauche (`x = demographique`) et `info_test_xavier` comme ensemble de données de droite (`y = info_test_xavier`), à quoi devrions-nous nous attendre ? Rappelons que Charlie n'est présent que dans l'ensemble de 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 = info_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 *info_test_xavier*, il est toujours inclus dans la sortie. (Mais bien sûr, comme ses informations de test ne sont pas disponibles dans *info_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 *info_test_xavier* est l'ensemble de données de gauche et *demographique* celui de droite :

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

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

VOCAB



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.



Q : `left_join()` entre diagnostics et démographies

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",  
  4,           "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::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::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.



Q : left_join() entre cas de tuberculose et continents

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 `info_test_xavier` originaux :

```
demographique
```

```
info_test_xavier
```

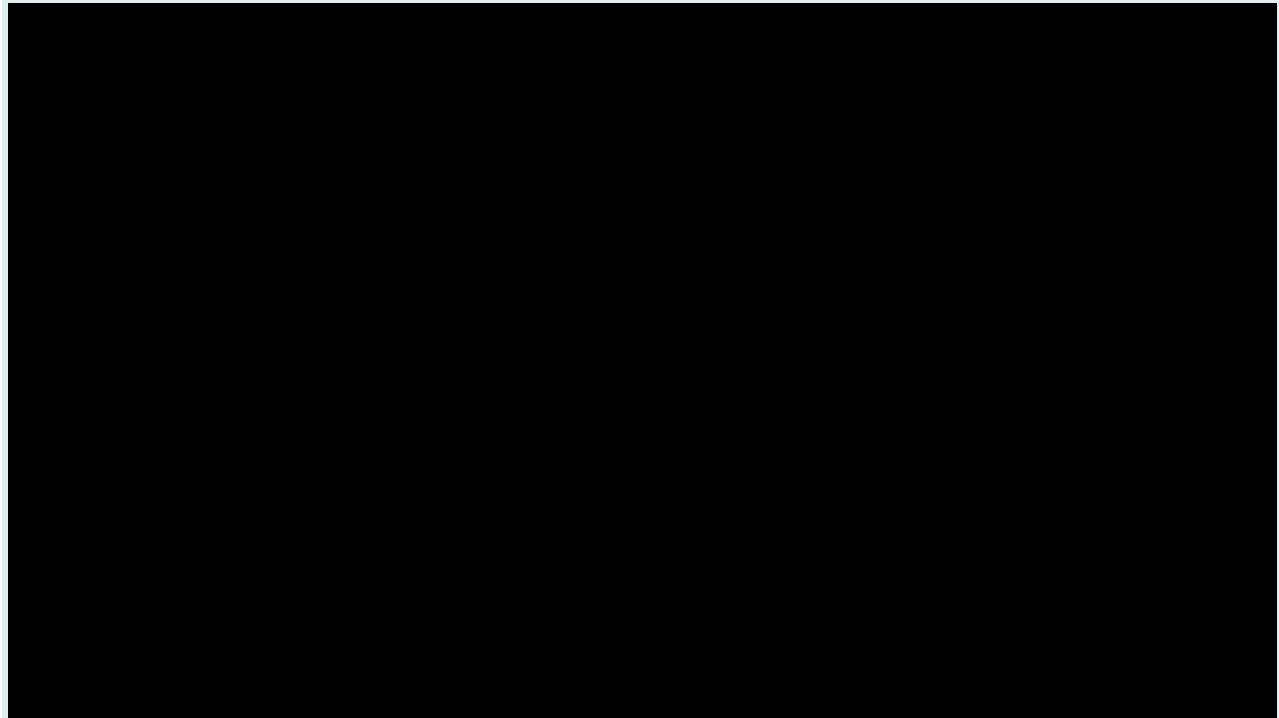
Essayons maintenant `right_join()`, avec `demographique` comme dataframe de droite :

```
right_join(x = info_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 `info_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, démographique  
right_join(x = info_test_xavier, y = démographique)
```

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

```
# ici, LEFT_join privilégie le df de GAUCHE, encore une fois démographique  
left_join(x = démographique, y = info_test_xavier)
```

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



SIDE NOTE

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 :

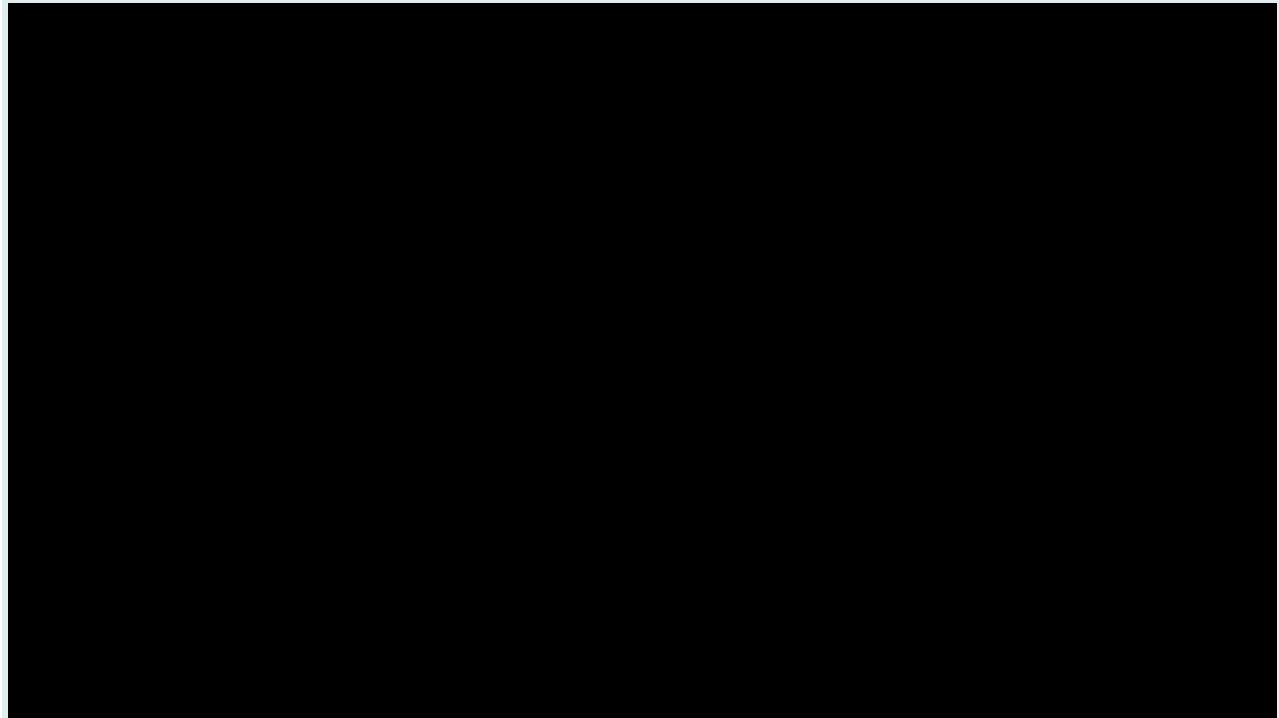
```
demographique
```

```
info_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, info_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(info_test_xavier, demographique, by="nom")
```

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

PRACTICE



Q : inner_join() entre pathogènes

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.

Retournons à nos données sur les dépenses de santé et l'incidence de la tuberculose 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 des données à la fois 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
```

Super!

Remarquez qu'il n'y a maintenant que 44 lignes dans le résultat, car les trois pays sans informations de dépenses correspondantes dans `dep_sante_2019` ont été exclus.

En plus de `left_join()`, le `inner_join()` est l'un des jointures les plus courantes lors du travail avec des données, donc il est probable que vous le rencontrerez souvent. 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 voulez uniquement des enregistrements correspondants dans votre ensemble de données final ou vous pourriez finir par perdre beaucoup de données accidentellement ! En contraste, `full_join()` est la jointure la plus inclusive, jetons un coup d'œil dans la section suivante.

Q : `inner_join()` d'une seule ligne

Le bloc de code ci-dessous filtre le jeu de données `dep_sante_2019` aux 70 pays ayant les dépenses les plus élevées :



```
exp_elevees <-  
  dep_sante_2019 %>%  
  arrange(-expend_usd) %>%  
  head(70)
```

Utilisez un `inner_join()` pour joindre ce jeu de données `exp_elevees` avec le jeu de données d'incidence de la tuberculose en Afrique, `tb_2019_afrique`.

Si vous faites cela correctement, il n'y aura qu'une seule ligne retournée. Pourquoi ?

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 :

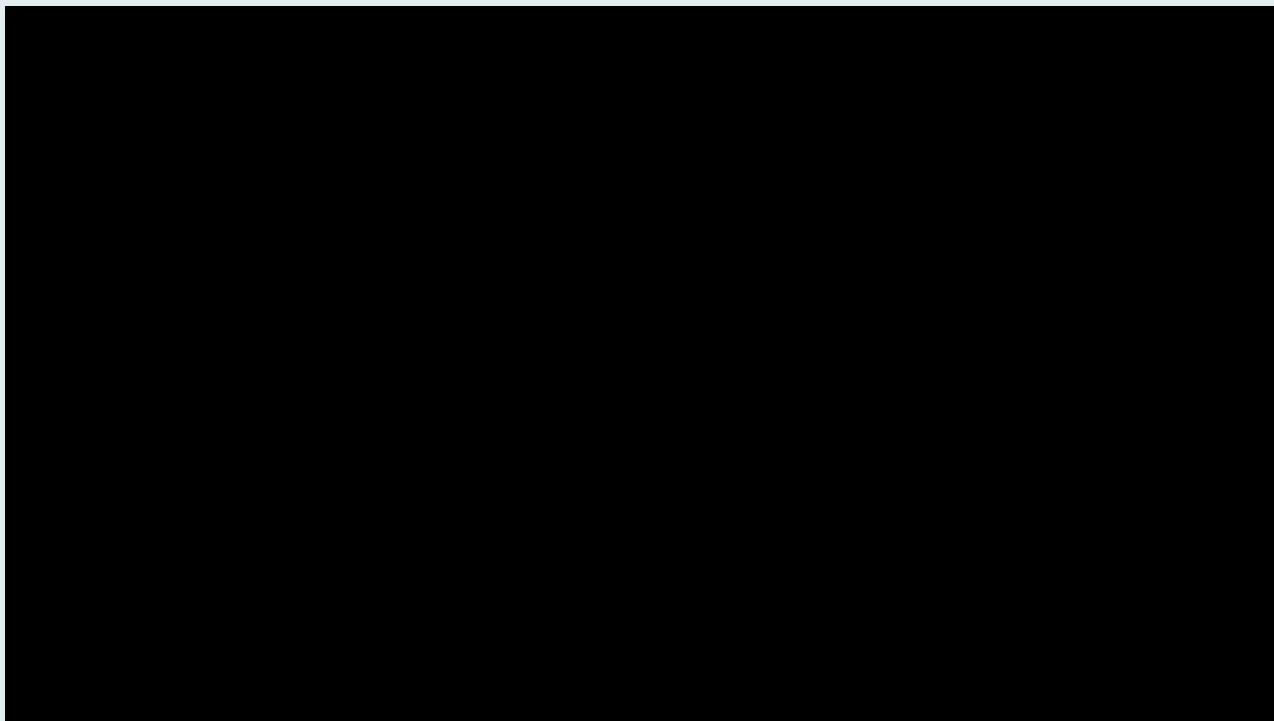
```
demographique
```

```
info_test_xavier
```

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

```
full_join(demographique, info_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(info_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.

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 %>%
  full_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.



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_dep_tb` en utilisant un `full_join` !

```
full_dep_tb <- tb_2019_afrique %>%
  full_join(dep_sante_2019)
```

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

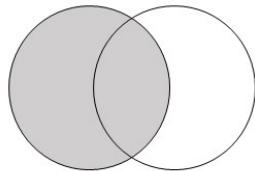
```
full_dep_tb
```

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

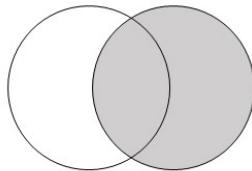
Résumé

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 !

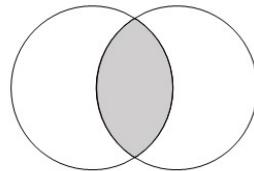
`left_join()`



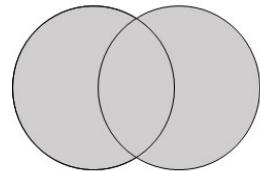
`right_join()`



`inner_join()`



`full_join()`



Answer Key

Q : `left_join()` entre patients et controles

```
left_join(x=patients, y=controles)
```

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

```
## # A tibble: 3 × 4
##   id_patient nom     age date_controle
```

```
##      <dbl> <chr> <dbl> <chr>
## 1      1 John     32 2023-01-20
## 2      2 Joy      28 2023-02-20
## 3      3 Khan     40 2023-05-15
```

Q : `left_join()` avec l'argument “by”

```
left_join(x=details_patient, y=registres_vaccination,
by=c("numero_id"="code_patient"))
```

```
## # A tibble: 3 × 5
##   numero_id nom_complet adresse    type_vaccin
##   <chr>      <chr>       <chr>      <chr>
## 1 A001       Alice       123 Elm St COVID-19
## 2 B002       Bob        456 Maple Dr Grippe
## 3 C003       Charlie    789 Oak Blvd Hépatite B
## # i 1 more variable: date_vaccination <chr>
```

Q : `left_join()` entre diagnostics et démographies

```
left_join(x=demographique_patient, y=dx_maladie)
```

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

## # A tibble: 5 × 6
##   id_patient nom      age genre maladie date_diagnostic
##   <dbl> <chr>    <dbl> <chr> <chr>   <chr>
## 1 1 Fred      28 Femme Influenza 2023-01-15
## 2 2 Genevieve 45 Femme <NA>      <NA>
## 3 3 Henry     32 Homme <NA>      <NA>
## 4 5 Irene     55 Femme <NA>      <NA>
## 5 8 Jules     40 Homme Influenza 2023-02-20
```

Q : `left_join()` entre cas de tuberculose et continents

```
left_join(x=cas_tb_enfants, y=pays_continents,
by=c(country="country.name.fr"))
```

```
## # A tibble: 5 × 4
##   country      cas_tb_smear_0_14 continent region
##   <chr>          <dbl> <chr>      <chr>
## 1 Afghanistan  588 Asia       South Asia
```

```
## 4 American Samoa NA <NA> <NA>
## 5 Andorra 0 <NA> <NA>
```

Q : inner_join() entre pathogènes

```
inner_join(total_inf, resultats)
```

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

```
## # A tibble: 2 × 4
##   pathogene    total_infections n_hosp n_deces
##   <chr>          <dbl>     <dbl>    <dbl>
## 1 Campylobacter      9751     1938      42
## 2 Listeria            136      128       30
```

Q : inner_join() d'une seule ligne

```
inner_join(exp_elevees, tb_2019_afrique)
```

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

```
## # A tibble: 1 × 4
##   country    expend_usd cases conf_int_95
##   <chr>        <dbl>  <dbl>   <chr>
## 1 Seychelles      572     15 [13 – 18]
```

Il n'y a qu'un seul pays en commun entre les deux ensembles de données.

Q : full_join() avec des données sur le paludisme

```
full_join(inc_paludisme, deces_paludisme)
```

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

```
## # A tibble: 5 × 3
##   année inc_100k deces_100k
##   <dbl>    <dbl>     <dbl>
## 1 2010      69.5      NA
## 2 2011      66.5     12.9
## 3 2014      59.8      NA
```

## 4	2016	58.7	9.4
## 5	2017	59.2	NA

Contributors

The following team members contributed to this lesson:



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement



AMANDA MCKINLEY

R Developer and Instructor, the GRAPH Network

Joindre des tables de données (leçon 2)

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 :

```
uire(pacman)) install.packages("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(
  ~age,
  "Alice", 25,
  "Bob", 32,
  "Charlie", 45,
  "David"
)
demographique
```

```
info_test <- tribble(
  ~date_test, ~resultat,
  "2023-06-05", "Negatif",
  "2023-08-10", "Positif",
  "2023-05-02", "Negatif",
  "2023-07-20", "Positif"
)
info_test
```

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

```
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
```

```
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 ?

```
tibble(  
  ~Capitale,  
,   "New Delhi",  
nésie",  "Jakarta",  
ippines", "Manille"  
  
tibble(  
  ~Population,    ~Esperance_de_vie,  
,      1393000000,  69.7,  
nésie",  273500000, 71.7,  
ippines", 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.

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

```
ation
```

```
## # 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  
## # i 1 more variable: notifiee_relle <dbl>
```

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.

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

```
## # A tibble: 5 × 4
##   Etat           systeme_sante covid_test covid_diagnositi...
##   <chr>          <chr>        <dbl>        <dbl>
## 1 Iles Andaman e... public         322         0
## 2 Iles Andaman e... privé          1         0
## 3 Andhra Pradesh public       63319        97
## 4 Andhra Pradesh privé        26410        17
## 5 ArunachalPrade... public      1761         0
## # i abbreviated name: `covid_diagnosique`
```

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`.

```
notification$Etat, covid$Etat)
```

```
## [1] "Arunachal Pradesh"                      "Dadra et Nagar Haveli et"
Daman et Diu" "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.

```
covid$Etat, notification$Etat)
```

```
## [1] "ArunachalPradesh"                      "Dadra & Nagar Haveli & Daman &
Diu" "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 comparer 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 %>%
  e(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))

notification$Etat, covid$Etat)
```

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

```
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.

```
← read_csv(here("data/enfant_TB_Inde.csv"))
```

```
## # A tibble: 5 × 4
##   Etat                      systeme_sante enfant_notifie
##   <chr>                     <chr>                <dbl>
```

```

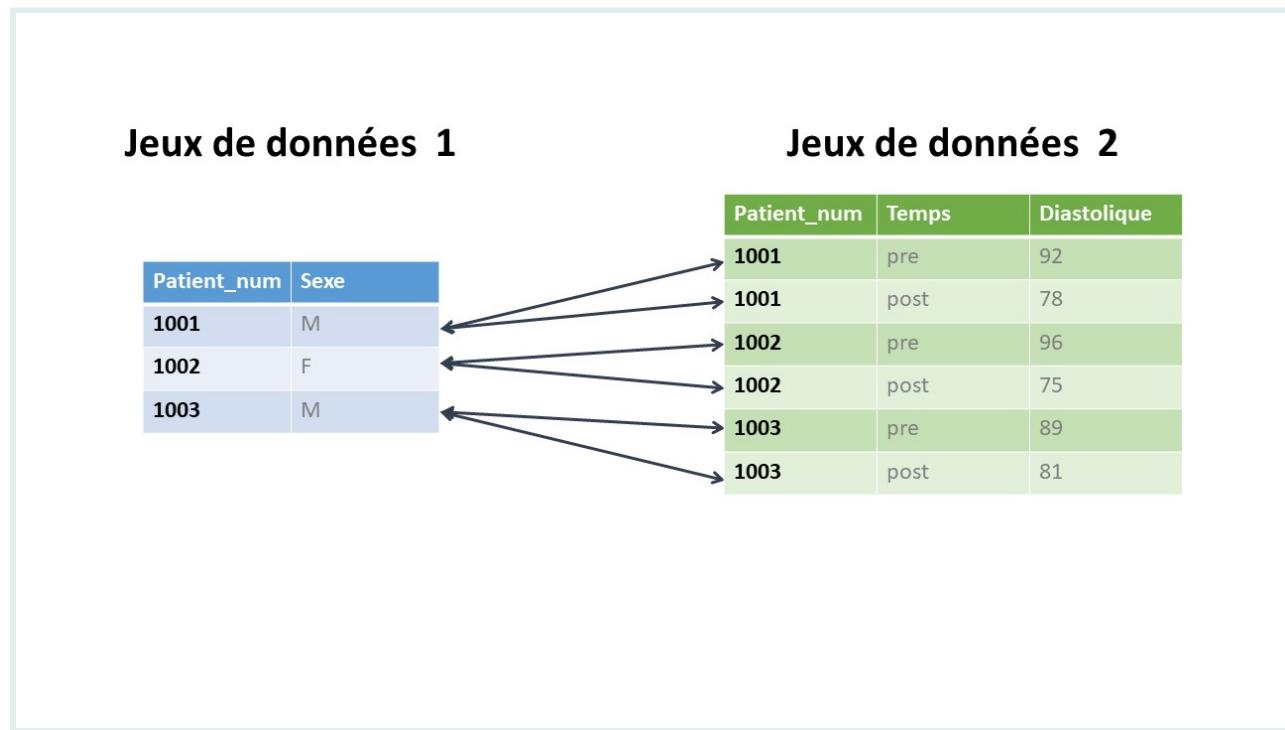
## 1 Iles Andaman et Nicobar public           18
## 2 Iles Andaman et Nicobar privé            1
## 3 Andhra Pradesh public                   1347
## 4 Andhra Pradesh privé                  1333
## 5 Arunachal Pradesh public              256
## # i 1 more variable: enfant_traitement <dbl>

```

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:



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:

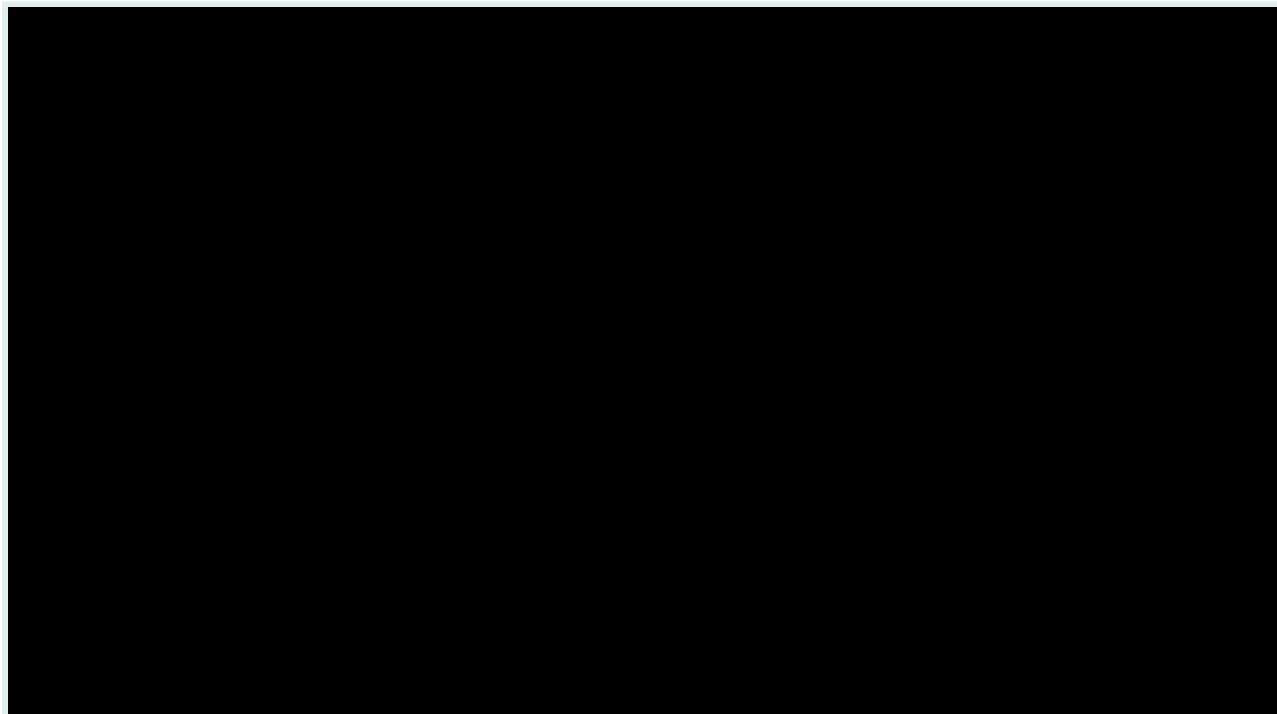
```
st_multiples <- tribble(  
  ~date_test, ~resultat,  
  "2023-06-05", "Negatif",  
  "2023-06-10", "Positif",  
  "2023-08-10", "Positif",  
  "2023-05-02", "Negatif",  
  "2023-05-12", "Negatif",
```

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:

```
ln(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.

```
ln(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 :

```
cient <- tribble(
  .by = "nom",
  ~ent_num, ~nom,      ~age,
  "Liam",        32,
  "Manny",       28,
  "Nico",        40

; <- tribble(
  .by = "maladie",
  ~ent_num, ~maladie,
  "Diabète",
  "Hypertension",
  "Asthme",
  "Cholestérol Élevé",
  "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:

```
ation
```

```
## # 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
## # i 1 more variable: notifiee_relle <dbl>
```

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.

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

```
## # A tibble: 5 × 3
##   conseil_zonal    sous_division     Etat
##   <chr>            <chr>           <chr>
## 1 Pas de Conseil Zonal Territoire de l'Un... Iles Andaman e...
## 2 Conseil du Nord-Est État             Arunachal Prad...
## 3 Conseil du Nord-Est État             Assam
## 4 Conseil Zonal de l'Est  État             Bihar
## 5 Conseil Zonal du Nord Territoire de l'Un... Chandigarh
```

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

```
notification$Etat, regions$Etat)
```

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

```
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 !

```
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.

Parfait ! Nous savons maintenant comment utiliser une jointure de type `left_join()` lors de la jointure de jeux de données avec une correspondance un-à-plusieurs. Regardons les 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.

```
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:

```
regions
```



```
covid
```

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`.

```
covid_regions <- covid %>%
  join(regions)
```

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

```
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 apparier. 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.

```
_arterielle <- tribble(
  ~temps, ~systolique, ~diastolique,
  "pre",      139,      87,
  "post",     121,      82,
  "pre",      137,      86,
  "post",     128,      79,
  "pre",      137,      81,
  "post",     130,      73
)_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.

```

tribble(
  ~temps, ~creatinine,
  "l", "pre",      0.9,
  "l", "post",     1.3,
  "l", "pre",      0.7,
  "l", "post",     0.8,
  "lo", "pre",     0.6,
  "lo", "post",    1.4

```

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 :

```

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.

```

```

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.

```

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

```

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

```
ation
```

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 appuyer 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 !

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

Notes de leçon : Introduction aux fonctions et aux conditionnelles

Introduction
Objectifs d'apprentissage
Packages
Bases d'une fonction
Quand écrire une fonction en R
Fonctions avec Plusieurs Arguments
Passage d'Arguments aux Fonctions Internes
L'Argument Ellipse,
Compréhension de la Portée en R
Introduction aux Conditionnels : <code>if, else if et else</code>
Vérification des arguments avec des conditionnels
Conditionnels Vectorisés
Où stocker vos fonctions
Conclusion !
Corrigés

Introduction

Les deux composants principaux du langage R sont les objets et les fonctions. Les objets sont les structures de données que nous utilisons pour stocker des informations, et les fonctions sont les outils que nous utilisons pour manipuler ces objets. Citant [John Chambers](#), qui a joué un rôle clé dans le développement du langage R, tout ce qui "existe" dans un environnement R est un objet, et tout ce qui "se passe" est une fonction.

Jusqu'à présent, vous avez principalement utilisé des fonctions écrites par d'autres. Dans cette leçon, vous apprendrez à écrire vos propres fonctions en R.

Écrire des fonctions vous permet d'automatiser des tâches répétitives, d'améliorer l'efficacité et de réduire les erreurs dans votre code.

Dans cette leçon, nous apprendrons les fondamentaux des fonctions avec des exemples simples. Puis, dans une leçon future, nous écrirons des fonctions plus complexes qui peuvent automatiser de grandes parties de votre flux de travail d'analyse de données.

Objectifs d'apprentissage

À la fin de cette leçon, vous serez capable de :

1. Créer et utiliser vos propres fonctions en R.
2. Concevoir des arguments de fonction et définir des valeurs par défaut.
3. Utiliser une logique conditionnelle telle que `if, else if, et else` au sein des fonctions.
4. Vérifier et valider les arguments de fonction pour prévenir les erreurs.
5. Gérer la portée des fonctions et comprendre les variables locales vs. globales.

6. Gérer des données vectorisées dans les fonctions.
 7. Organiser et stocker vos fonctions personnalisées pour une réutilisation facile.
-

Packages

Exécutez le code suivant pour installer et charger les packages nécessaires pour cette leçon :

```
if (!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, here, NHSRdatasets, medicaldata, outbreaks,
reactable)
```

Bases d'une fonction

Commençons par créer une fonction très simple. Considérez la fonction suivante qui convertit les pounds (une unité de poids) en kilogrammes (une autre unité de poids) :

```
pounds_en_kg <- function(pounds) {
  return(pounds * 0.4536)
}
```

Si vous exécutez ce code, vous créerez une fonction nommée `pounds_en_kg`, qui peut être utilisée directement dans un script ou dans la console :

```
pounds_en_kg(150)
```

```
## [1] 68.04
```

Décortiquons la structure de cette première fonction étape par étape.

Tout d'abord, une fonction est créée en utilisant l'instruction `function`, suivie d'une paire de parenthèses et d'une paire d'accolades.

```
function() {  
}
```

À l'intérieur des parenthèses, nous indiquons les **arguments** de la fonction. Notre fonction ne prend qu'un seul argument, que nous avons décidé de nommer `pounds`. C'est la valeur que nous voulons convertir de pounds en kilogrammes.

```
function(pounds) {  
}
```

Bien sûr, nous aurions pu nommer cet argument comme nous le voulions.

L'élément suivant, à l'intérieur des accolades, est le **corps** de la fonction. C'est là que nous écrivons le code que nous voulons exécuter lorsque la fonction est appelée.

```
function(pounds) {  
  pounds * 0.4536  
}
```

Maintenant, nous voulons que notre fonction retourne ce qui est calculé à l'intérieur de son corps. Cela est réalisé via l'instruction `return`.

```
function(pounds) {  
  return(pounds * 0.4536)  
}
```

Parfois, vous pouvez omettre l'instruction `return` et simplement écrire l'expression à retourner à la fin de la fonction, car R retournera automatiquement la dernière expression évaluée dans la fonction :

```
function(pounds) {  
  pounds * 0.4536 # R retournera automatiquement cette expression  
}
```

Cependant, il est conseillé d'inclure toujours l'instruction `return`, car cela rend le code plus lisible.

Nous pourrions aussi vouloir d'abord assigner le résultat à un objet puis le retourner :

```
function(pounds) {  
  kg <- pounds * 0.4536  
  return(kg)  
}
```

C'est un peu plus long, mais cela rend la fonction plus claire.

Enfin, pour que notre fonction puisse être appelée et utilisée, nous devons lui donner un nom. Cela revient à stocker une valeur dans un objet. Ici, nous la stockons dans un objet nommé `pounds_to_kg`.

```
pounds_to_kg <- function(pounds) {  
  kg <- pounds * 0.4536  
  return(kg)  
}
```

Avec notre fonction, nous avons donc créé un nouvel objet dans notre environnement appelé `pounds_to_kg`, de classe `function`.

```
class(pounds_to_kg)
```

```
## [1] "function"
```

Nous pouvons maintenant l'utiliser ainsi avec un argument nommé :

```
pounds_to_kg(pounds = 150)
```

```
## [1] 68.04
```

Ou sans un argument nommé :

```
pounds_to_kg(150)
```

```
## [1] 68.04
```

La fonction peut également être utilisée avec un vecteur de valeurs :

```
my_vec <- c(150, 200, 250)
pounds_to_kg(my_vec)
```

```
## [1] 68.04 90.72 113.40
```

Et voilà ! Vous venez de créer votre première fonction en R.

Vous pouvez voir le code source de n'importe quelle fonction en tapant son nom sans parenthèses :

```
pounds_to_kg
```

```
## function(pounds) {
##   kg <- pounds * 0.4536
##   return(kg)
## }
## <bytecode: 0x298ac89d8>
```

Pour voir cela comme un script R, vous pouvez utiliser la fonction `View` :

```
View(pounds_to_kg)
```

Cela ouvrira un nouvel onglet dans RStudio avec le code source de la fonction.

Cette méthode fonctionne pour n'importe quelle fonction, pas seulement celles que vous créez. Pour un exemple à quoi ressemble une *vraie* fonction dans la pratique, essayez de

[View\(reactable\)](#)

Fonction Age Mois

Créez une fonction simple appelée `years_to_months` qui transforme l'âge en années en âge en mois.

Essayez-la avec `years_to_months(12)`

```
# Votre code ici  
years_to_months <- ...
```

Écrivons maintenant une fonction un peu plus complexe, pour un peu plus de pratique. La fonction que nous allons écrire convertira une température en Fahrenheit (utilisée aux États-Unis) en température en Celsius. La formule pour cette conversion est :

$$C = \frac{5}{9} \times (F - 32)$$

Et voici la fonction :

```
fahrenheit_to_celsius <- function(fahrenheit) {  
  celsius <- (5 / 9) * (fahrenheit - 32)  
  return(celsius)  
}  
  
fahrenheit_to_celsius(32) # point de congélation de l'eau. Devrait être 0
```

```
## [1] 0
```

Testons la fonction sur une colonne du jeu de données `airquality`, qui est l'un des jeux de données intégrés dans R :

```
airquality %>%  
  select(Temp) %>%  
  mutate(Temp = fahrenheit_to_celsius(Temp)) %>%  
  head()
```

```
##      Temp  
## 1 19.44444  
## 2 22.22222  
## 3 23.33333  
## 4 16.66667  
## 5 13.33333  
## 6 18.88889
```

Super !

Quand écrire une fonction en R

Dans R, de nombreuses opérations peuvent être complétées en utilisant des fonctions existantes ou en combinant quelques-unes. Cependant, il y a des occasions où il est avantageux de créer sa propre fonction :

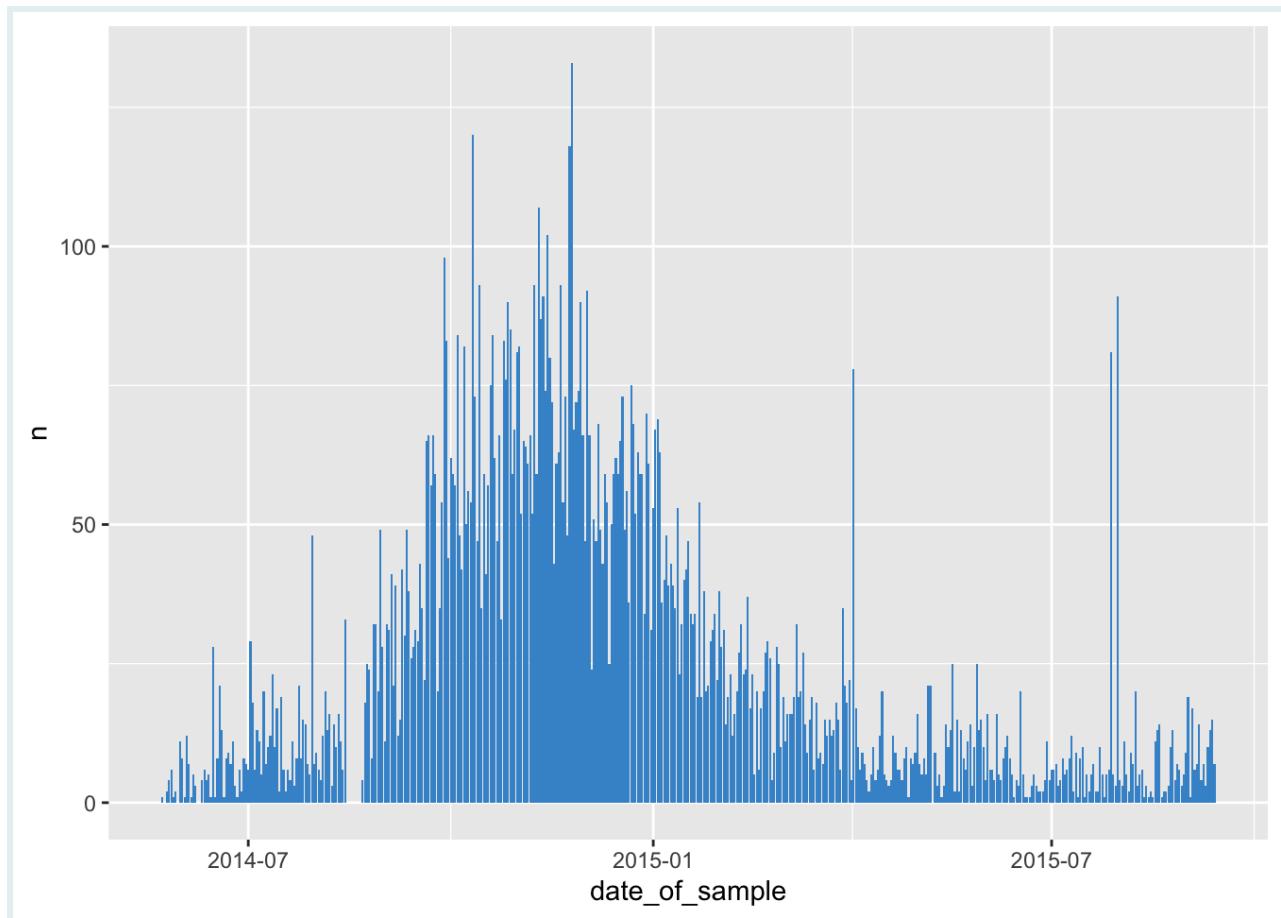
- **Réutilisabilité** : Si vous vous retrouvez à écrire le même code à plusieurs reprises, il peut être bénéfique de l'encapsuler dans une fonction. Par exemple, si vous convertissez fréquemment des températures de Fahrenheit en Celsius, créer une fonction `fahrenheit_to_celsius` rendrait votre code plus épuré et améliorerait l'efficacité.
- **Lisibilité** : Les fonctions peuvent améliorer la lisibilité du code, en particulier lorsqu'elles ont des noms descriptifs. Avec des fonctions simples comme `fahrenheit_to_celsius`, les avantages ne sont pas toujours évidents. Cependant, au fur et à mesure que les fonctions deviennent plus complexes, l'importance des noms descriptifs devient de plus en plus cruciale.
- **Partage** : Les fonctions facilitent le partage du code. Elles peuvent être distribuées soit dans le cadre d'un package, soit comme des scripts autonomes. Bien que la création d'un package soit plus complexe et au-delà du cadre de ce cours, partager des fonctions plus simples est assez direct. Nous parlerons des options pour cela plus tard dans la leçon.

::: note latérale **Fonctions pour les jeux de données et les Graphiques**

Les fonctions les plus utiles que vous écrirez probablement concerteront la manipulation de jeux de données et de graphiques. Voici une fonction qui prend une liste linéaire de cas et retourne un graphique de la courbe épidémique :

```
# Fonction pour tracer une courbe épidémique
tracer_courbe_epidemie <- function(donnees, colonne_date) {
  donnees %>%
    count({{ colonne_date }}) %>%
    complete({{ colonne_date }}) := seq(min({{ colonne_date }}),
                                         max({{ colonne_date }}), by="day")) %>%
    ggplot(aes(x = {{ colonne_date }}, y = n)) +
    geom_col(fill = "#4395D1")
}

# Exemple d'utilisation
tracer_courbe_epidemie(outbreaks::ebola_sierraleone_2014, date_of_sample)
```



Cette leçon abordera des fonctions plus complexes plus tard. Pour l'instant, nous nous concentrerons sur les bases de la rédaction de fonctions, en utilisant comme exemples des fonctions simples de manipulation de vecteurs. :::

Fonction de Conversion Celsius en Fahrenheit

Créez une fonction nommée `celsius_en_fahrenheit` qui convertit la température de Celsius en Fahrenheit. Voici la formule pour cette conversion :

$$\text{Fahrenheit} = \text{Celsius} \times 1.8 + 32$$

```
# Votre code ici
celsius_en_fahrenheit <- ...
```

Ensuite, testez votre fonction sur la colonne `temp` du jeu de données intégré `beaver1` dans R :

```
beaver1 %>%
  select(temp) %>%
  head()
```

Fonctions avec Plusieurs Arguments

La plupart des fonctions prennent plusieurs arguments plutôt qu'un seul. Examinons un exemple de fonction qui prend trois arguments :

```
calculate_calories <- function(carb_grams, protein_grams, fat_grams) {  
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)  
  return(result)  
}  
  
calculate_calories(carb_grams = 50, protein_grams = 25, fat_grams = 10)  
  
## [1] 390
```

La fonction `calculate_calories` calcul le total des calories basé sur les grammes de glucides, protéines, et lipides. On estime que les glucides et les protéines ont 4 calories par gramme, tandis que les lipides ont 9 calories par gramme.

Si vous essayez d'utiliser la fonction sans fournir tous les arguments, cela entraînera une erreur.

```
calculate_calories(carb_grams = 50, protein_grams = 25)
```

Erreur dans `calculate_calories(carb_grams = 50, protein_grams = 25)` :
l'argument "fat_grams" est manquant, avec aucune valeur par défaut

Vous pouvez définir des **valeurs par défaut** pour les arguments de votre fonction. Si un argument est **appelé** sans qu'une **valeur ne lui soit attribuée**, alors cet argument prend sa valeur par défaut.

Voici un exemple où `fat_grams` se voit attribuer une valeur par défaut de 0.

```
calculate_calories <- function(carb_grams, protein_grams, fat_grams = 0) {  
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)  
  return(result)  
}  
  
calculate_calories(50, 25)
```

```
## [1] 300
```

Dans cette version révisée, `carb_grams` et `protein_grams` sont des arguments obligatoires, mais nous pourrions rendre tous les arguments optionnels en leur donnant tous des valeurs par défaut :

```
calculate_calories <- function(carb_grams = 0, protein_grams = 0, fat_grams = 0) {  
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)  
  return(result)  
}
```

Maintenant, nous pouvons appeler la fonction sans arguments :

```
calculate_calories()
```

```
## [1] 0
```

Nous pouvons également l'appeler avec certains arguments :

```
calculate_calories(carb_grams= 50, protein_grams = 25)
```

```
## [1] 300
```

Et cela fonctionne comme prévu.

Fonction de Calcul de l'IMC

Créez une fonction nommée calc_imc qui calcule l'Indice de Masse Corporelle (IMC) pour une ou plusieurs personnes. Gardez à l'esprit que l'IMC est calculé comme le poids en kg divisé par le carré de la taille en mètres. Par conséquent, cette fonction nécessite deux arguments obligatoires : le poids et la taille.

```
# Votre code ici  
calc_imc <- ...
```

Ensuite, appliquez votre fonction au jeu de données medicaldata::smartpill pour calculer l'IMC de chaque personne :

```
medicaldata::smartpill %>%  
  as_tibble() %>%  
  select(Weight, Height) %>%  
  mutate(IMC = calc_imc(Weight, Height))
```

Passage d'Arguments aux Fonctions Internes

Lors de l'écriture de fonctions en R, vous pourriez avoir besoin d'utiliser des fonctions existantes au sein de votre fonction personnalisée. Par exemple, considérons notre fonction familière qui convertit les livres en kilogrammes :

```
pounds_en_kg <- function(pounds) {  
  kg <- pounds * 0.4536  
  return(kg)  
}
```

Il pourrait être utile de pouvoir arrondir le résultat à un nombre spécifié de décimales sans appeler une fonction séparée.

Pour cela, nous pouvons intégrer directement la fonction `round` dans notre fonction personnalisée. La fonction `round` a deux arguments : `x`, le nombre à arrondir, et `digits`, le nombre de décimales à arrondir :

```
round(x = 1.2345, digits = 2)
```

```
## [1] 1.23
```

Maintenant, nous pouvons ajouter un argument à notre fonction appelé `arrondir_a` qui sera passé à l'argument `digits` de la fonction `round`.

```
pounds_en_kg <- function(pounds, arrondir_a = 2) {  
  kg <- pounds * 0.4536  
  kg_arrondi <- round(x = kg, digits = arrondir_a)  
  return(kg_arrondi)  
}
```

Dans la fonction ci-dessus, nous avons ajouté un argument appelé `arrondir_a` avec une valeur par défaut de 3.

Maintenant, lorsque vous passez une valeur à l'argument `arrondir_a`, elle sera utilisée par la fonction `round`.

```
pounds_en_kg(10) # sans argument passé à arrondir_a, la valeur par défaut de 2  
est utilisée
```

```
## [1] 4.54
```

```
pounds_en_kg(10, arrondir_a = 1)
```

```
## [1] 4.5
```

```
pounds_en_kg(10, arrondir_a = 3)
```

```
## [1] 4.536
```

L'Argument Ellipse, ...

Parfois, il y a de nombreux arguments à passer à une fonction interne. Par exemple, considérez la fonction `format()` en R, qui a de nombreux arguments :

```
format(x = 12364.2345,  
       big.mark = " ", # séparateur de milliers  
       decimal.mark = ",", # point décimal à la française !  
       nsmall = 2, # nombre de chiffres après la virgule  
       scientific = FALSE # utiliser la notation scientifique ?  
     )
```

```
## [1] "12 364,23"
```

Vous pouvez voir tous les arguments en tapant `?format` dans la console.

Si nous voulons que notre fonction puisse passer tous ces arguments à la fonction `format`, nous utiliserons l'argument ellipse, Voici un exemple :

```
pounds_en_kg <- function(pounds, ...) {  
  kg <- pounds * 0.4536  
  kg_formate <- format(x = kg, ...)  
  return(kg_formate)  
}
```

Maintenant, lorsque nous passons des arguments à la fonction `livres_en_kg`, ils seront transmis à la fonction `format`, même si nous ne les avons pas explicitement définis dans notre fonction.

```
pounds_en_kg(10000.234)
```

```
## [1] "4536.106"
```

```
pounds_en_kg(10000.234, big.mark = " ", decimal.mark = ",")
```

```
## [1] "4 536,106"
```

Super !

Pratique avec l'Argument ...

Considérez notre fonction `calculer_calories()`.

```
calculate_calories <- function(carb_grams, protein_grams, fat_grams) {  
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)  
  return(result)  
}
```

Améliorez cette fonction pour accepter les arguments de formatage en utilisant le mécanisme

Compréhension de la Portée en R

La portée se réfère à la visibilité des variables et objets dans différentes parties de votre code R. Il est important de comprendre la portée lors de l'écriture de fonctions.

Les objets créés à l'intérieur d'une fonction ont une **portée locale** au sein de cette fonction (par opposition à une **portée globale**) et ne sont pas accessibles en dehors de la fonction. Illustrons cela avec la fonction pounds_en_kg :

Imaginez que vous voulez convertir un poids en pounds en kilogrammes et vous écrivez une fonction pour cela :

```
pounds_en_kg <- function(pounds) {  
  kg <- pounds * 0.4536  
}
```

Vous pourriez être tenté d'essayer d'accéder à la variable kg en dehors de la fonction, mais vous obtiendrez une erreur :

```
pounds_en_kg(50)  
kg
```

Erreur : objet 'kg' introuvable

C'est parce que kg est une variable locale à l'intérieur de la fonction livres_en_kg et n'est pas accessible dans l'environnement global.

Pour utiliser une valeur générée à l'intérieur d'une fonction, nous devons nous assurer qu'elle est retournée par la fonction :

```
pounds_en_kg <- function(pounds) {  
  kg <- pounds * 0.4536  
  return(kg)  
}
```

Et ensuite, nous pouvons stocker le résultat de la fonction dans une variable globale :

```
kg <- pounds_en_kg(50)
```

Maintenant, nous pouvons accéder à l'objet kg :

```
kg
```

```
## [1] 22.68
```

L'Opérateur de Superassignation, <—

Bien que nous ayons dit que les objets créés au sein d'une fonction ont une portée locale, il est en réalité **possible** de créer des variables globales depuis l'intérieur d'une fonction en utilisant l'opérateur spécial <<—.

Considérez l'exemple suivant :

```
test <- function() {  
  nouvel_objet <<- 15  
}
```

Maintenant, si nous exécutons la fonction, nouvel_objet sera créé dans l'environnement global, avec une valeur de 15 :

```
test()  
nouvel_objet
```

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

Bien que cela soit techniquement possible, cela n'est généralement pas recommandé (surtout pour les non-experts) en raison des effets secondaires potentiels et des défis de maintenance.

Introduction aux Conditionnels : if, else if et else

Les conditionnels, qui sont utilisés pour contrôler le flux d'exécution du code, sont une partie essentielle de la programmation, en particulier lors de l'écriture de fonctions. En R, les conditionnels sont mis en œuvre à l'aide des instructions `if`, `else` et `else if`.

Lorsque nous utilisons `if`, nous spécifions que nous voulons que certaines parties du code s'exécutent uniquement si une condition spécifique est vraie.

Voici la structure d'une instruction `if` :

```
if (condition) {  
  # code à exécuter si la condition est vraie  
}
```

Remarquez que cela ressemble à la structure d'une fonction.

Maintenant, voyons une instruction `if` en action, pour convertir une température de Celsius en Fahrenheit :

```
celsius <- 20  
convertir_en <- "fahrenheit"  
  
if (convertir_en == "fahrenheit") {  
  fahrenheit <- (celsius * 9/5) + 32  
  print(fahrenheit)  
}
```

```
## [1] 68
```

Dans ce fragment de code, si la variable `convertir_en` est égale à "fahrenheit", la conversion est réalisée et le résultat est imprimé.

Voyons maintenant ce qui se passe lorsque `convertir_en` est défini à une valeur autre que "fahrenheit" :

```
convertir_en <- "kelvin"  
  
if (convertir_en == "fahrenheit") {  
  fahrenheit <- (celsius * 9/5) + 32  
  print(fahrenheit)  
}
```

Dans ce cas, rien n'est imprimé car la condition n'est pas satisfaite.

Pour gérer les situations où `convertir_en` n'est pas "fahrenheit", nous pouvons ajouter une instruction `else` :

```
convertir_en <- "kelvin"  
  
if (convertir_en == "fahrenheit") {  
  fahrenheit <- (celsius * 9/5) + 32  
  print(fahrenheit)  
} else {  
  print(celsius)  
}
```

```
## [1] 20
```

Ici, si la variable `convertir_en` ne correspond pas à "fahrenheit", le code dans le bloc `else` s'exécute, imprimant la valeur originale en Celsius.

Pour vérifier plusieurs conditions spécifiques, nous pouvons utiliser `else if` :

```
convertir_en <- "kelvin"

if (convertir_en == "fahrenheit") {
  fahrenheit <- (celsius * 9/5) + 32
  print(fahrenheit)
} else if (convertir_en == "kelvin") {
  temp_kelvin <- celsius + 273.15
  print(temp_kelvin)
} else {
  print(celsius)
}
```

```
## [1] 293.15
```

Ici, le code gère trois possibilités :

- Convertir en Fahrenheit si `convertir_en` est "fahrenheit".
- Si `convertir_en` n'est PAS "fahrenheit", vérifie si c'est "kelvin". Si c'est le cas, convertir en Kelvin.
- Si `convertir_en` n'est ni "fahrenheit" ni "kelvin", imprimer la valeur originale en Celsius.

Notez que vous pouvez avoir autant d'instructions `else if` que nécessaire, tandis que vous ne pouvez avoir qu'une seule instruction `else` attachée à une instruction `if`.

Finalement, nous pouvons encapsuler cette logique dans une fonction. Nous allons assigner le résultat à l'intérieur de chaque conditionnel à une variable appelée `sortie` et retourner `sortie` à la fin de la fonction :

```
convertir_celsius <- function(celsius, convertir_en) {
  if (convertir_en == "fahrenheit") {
    sortie <- (celsius * 9/5) + 32
  } else if (convertir_en == "kelvin") {
    sortie <- celsius + 273.15
  } else {
    sortie <- celsius
  }
  return(sortie)
}
```

Testons la fonction :

```
convertir_celsius(20, "fahrenheit")
```

```
## [1] 68
```

```
convertir_celsius(20, "kelvin")
```

```
## [1] 293.15
```

Un problème avec la fonction actuelle est que si une valeur invalide est passée à convert_to, nous n'obtenons aucun message informatif à ce sujet :

```
convertir_celsius(20, "celsius")
```

```
## [1] 20
```

```
convertir_celsius(20, "foo")
```

```
## [1] 20
```

C'est un problème courant avec les fonctions qui utilisent des conditionnels. Nous discuterons de la manière de gérer cela dans la section suivante.

Débogage d'une fonction avec une logique conditionnelle

Une fonction nommée check_negatives est conçue pour analyser un vecteur de nombres en R et imprimer un message indiquant si le vecteur contient des nombres négatifs. Cependant, la fonction contient actuellement des erreurs de syntaxe.

```
check_negatives <- function(numbers) {  
  x <- numbers  
  
  if any(x < 0) {  
    print("x contient des nombres négatifs")  
  } else {  
    print("x ne contient pas de nombres négatifs")  
}
```

Identifiez et corrigez les erreurs de syntaxe dans la fonction check_negatives. Après avoir corrigé la fonction, testez-la avec les vecteurs suivants pour vous assurer qu'elle fonctionne correctement : 1. c(8, 3, -2, 5) 2. c(10, 20, 30, 40)

Vérification des arguments avec des conditionnels

Lors de l'écriture de fonctions en R, il est souvent utile de s'assurer que les entrées fournies sont sensées et dans le domaine attendu. Sans vérifications appropriées, une fonction peut retourner des résultats incorrects ou échouer silencieusement, ce qui peut être source de confusion et de perte de temps pour le débogage. C'est là que la vérification des arguments intervient.

Considérez le scénario suivant avec notre fonction de conversion de température `convertir_celsius()`, qui convertit une température de Celsius en fahrenheit ou kelvin

```
convertir_celsius (30, "centigrade")
```

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

Dans ce cas, l'utilisateur essaie de convertir une température de Kelvin en "centigrade". Mais notre fonction échoue silencieusement, retournant la valeur Celsius d'origine au lieu d'un message d'erreur. Cela est dû au fait que l'argument `convertir_en` n'est pas vérifié pour sa validité.

Pour améliorer notre fonction, nous pouvons introduire une vérification des arguments pour valider `convertir_de` et `convertir_en`. La fonction `stop()` en R nous permet de terminer l'exécution d'une fonction et d'imprimer un message d'erreur. Voici comment nous pouvons utiliser `stop()` pour vérifier des valeurs valides de `convertir_de` et `convertir_en` :

```
# Tester stop() en dehors d'une fonction, pour l'intégrer plus tard dans
# conv_temp()
convertir_en <- "mauvaise échelle"

if (!(convertir_en %in% c("fahrenheit", "kelvin"))) {
  stop("convertir_en doit être fahrenheit ou kelvin")
}
```

Erreur : convertir_en doit être celsius, fahrenheit, ou kelvin

Intégrons cela dans notre fonction `convertir_celsius()` :

```

convertir_celsius <- function(celsius, convert_en) {
  if (!(convert_en %in% c("fahrenheit", "kelvin"))) {
    stop("convert_en doit être fahrenheit ou kelvin")
  }

  if (convert_en == "fahrenheit") {
    out <- (celsius * 9/5) + 32
  } else if (convert_en == "kelvin") {
    out <- celsius + 273.15
  } else {
    out <- celsius
  }
  return(out)
}

```

Notez que dans cette fonction mise à jour, il n'est plus nécessaire d'avoir une instruction `else`, car la fonction `stop()` mettra fin à l'exécution de la fonction si `convert_en` n'est pas l'une des trois valeurs valides. Ainsi, nous pouvons simplifier la fonction comme suit :

```

convertir_celsius <- function(celsius, convert_en) {
  if (!(convert_en %in% c("fahrenheit", "kelvin"))) {
    stop("convert_en doit être fahrenheit ou kelvin")
  }

  if (convert_en == "fahrenheit") {
    out <- (celsius * 9/5) + 32
  } else if (convert_en == "kelvin") {
    out <- celsius + 273.15
  }
  return(out)
}

```

Maintenant, si nous exécutons la commande problématique originale :

```
convertir_celsius(30, "centigrade")
```

La fonction s'arrêtera immédiatement et fournira un message d'erreur clair, indiquant que "centigrade" n'est pas une échelle de température reconnue.



PRO TIP

Bien que la vérification des arguments améliore la fiabilité des fonctions, une utilisation excessive peut ralentir les performances et compliquer le code. Avec le temps, en examinant le code d'autres personnes et grâce à l'expérience, vous développerez un bon sens de l'équilibre à trouver entre rigueur, efficacité et clarté. Pour l'instant, notez qu'il est généralement bon de pencher vers plus de vérifications.

Exercice sur la vérification des arguments

Considérez la fonction `calculate_calories` que nous avons écrite plus tôt :

```
calculate_calories <- function(carb_grams = 0, protein_grams = 0, fat_grams = 0) {  
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)  
  return(result)  
}
```

Écrire une fonction appelée `calculate_calories2()` qui est identique à `calculate_calories()` sauf qu'elle vérifie si les arguments `carb_grams`, `protein_grams` et `fat_grams` sont numériques. Si l'un d'eux n'est pas numérique, la fonction doit imprimer un message d'erreur en utilisant la fonction `stop()`.

```
calculate_calories2 <- function(carb_grams = 0, protein_grams = 0, fat_grams = 0) {  
  
  # votre code ici  
  
  return(result)  
}
```

Conditionnels Vectorisés

Dans les exemples précédents, nous avons ajouté une condition sur une seule valeur, telle que l'argument `convert_en`. Maintenant, explorons comment construire des conditionnels basés sur un vecteur de valeurs. De tels cas nécessitent une gestion spéciale car l'instruction `if` n'est pas vectorisée.

Par exemple, si nous voulions écrire une fonction pour classifier les lectures de température en hypothermie, normal ou fièvre, vous pourriez penser à utiliser une construction `if-else` comme ceci :

```
classify_temp <- function(temp) {  
  if (temp < 36.5) {  
    print("hypothermie")  
  } else if (temp >= 36.5 & temp <= 37.5) {  
    print("normal")  
  } else if (temp > 37.5) {  
    print("fièvre")  
  }  
}
```

Cela fonctionne sur une seule valeur :

```
classify_temp(36)
```

```
## [1] "hypothermie"
```

Mais cela ne fonctionnera pas comme prévu sur un vecteur, car l'instruction `if` n'est pas vectorisée et évalue seulement le premier élément du vecteur. Par exemple :

```
temp_vec <- c(36, 37, 38)
```

```
classify_temp(temp_vec) # Ceci ne fonctionnera pas correctement
```

```
Erreur dans if (temp < 35) { : la condition a une longueur > 1
```

Pour traiter l'ensemble du vecteur, vous devriez utiliser des fonctions vectorisées telles que `ifelse` ou `case_when` du package `dplyr`. Voici comment vous pouvez employer `ifelse` :

```
classify_temp <- function(temp) {
  out <- ifelse(temp < 36.5, "hypothermie",
                ifelse(temp >= 36.5 & temp <= 37.5, "normal",
                       ifelse(temp > 37.5, "fièvre", NA)))
  return(out)
}

classify_temp(temp_vec) # Ceci fonctionne comme prévu
```

```
## [1] "hypothermie" "normal"      "fièvre"
```

Pour une alternative plus propre et plus lisible, `dplyr:::case_when` peut également être utilisé :

```
classify_temp <- function(temp) {
  case_when(
    temp < 36.5 ~ "hypothermie",
    temp >= 36.5 & temp <= 37.5 ~ "normal",
    temp > 37.5 ~ "fièvre",
    TRUE ~ NA_character_
  )
}

classify_temp(temp_vec) # Ceci fonctionne aussi comme prévu
```

```
## [1] "hypothermie" "normal"      "fièvre"
```

Cette fonction fonctionne sans problème avec les jeux de données aussi :

```
NHSDatasets::synthetic_news_data %>%
  select(temp) %>%
  mutate(temp_class = classify_temp(temp))
```

```
## # A tibble: 1,000 × 2
##       temp temp_class
##   <dbl> <chr>
## 1 36.8  normal
## 2 35.0 hypothermie
## 3 36.2 hypothermie
## 4 36.9 normal
## 5 36.4 hypothermie
## 6 35.3 hypothermie
## 7 35.6 hypothermie
## 8 37.2 normal
## 9 35.5 hypothermie
## 10 35.3 hypothermie
## # i 990 more rows
```

Pratique pour classifier la dose d'Isoniazide

Appliquons ces connaissances à un cas pratique. Considérez la tentative suivante pour écrire une fonction qui calcule les dosages du médicament isoniazide pour des adultes pesant plus de 30kg :

```
calculer_dosage_isoniazide <- function(poids) {
  if (poids < 30) {
    stop("Le poids doit être au moins de 30 kg.")
  } else if (poids <= 35) {
    return(150)
  } else if (poids <= 45) {
    return(200)
  } else if (poids <= 55) {
    return(300)
  } else if (poids <= 70) {
    return(300)
  } else {
    return(300)
  }
}
```

Cette fonction échoue avec un vecteur de poids. Votre tâche est d'écrire une nouvelle fonction `calculate_isoniazid dosage2()` qui peut gérer les entrées vectorielles. Pour vous assurer que tous les poids sont au-dessus de 30 kg, vous utiliserez la fonction `any()` dans votre vérification des erreurs.

Voici une ébauche pour vous aider à démarrer :

```
calculate_isoniazid_dosage2 <- function(weight) {  
  if (any(weight < 30)) stop("Tous les poids doivent être au moins de 30 kg.")  
  
  # Votre code ici  
  
  return(out)  
}  
  
calculate_isoniazid_dosage2(c(30, 40, 50, 100))
```

```
## Error in calculate_isoniazid_dosage2(c(30, 40, 50, 100)): object 'out' not found
```

Où stocker vos fonctions

Lorsque vous écrivez des scripts en R, décider où stocker vos fonctions est une considération importante pour maintenir un code propre et un flux de travail. Voici quelques stratégies clés :

1) Haut du Script

Placer les fonctions en haut de votre script est une pratique simple et couramment utilisée.

2) Script Séparé qui est utilisé comme source

À mesure que votre projet grandit, vous pouvez avoir plusieurs fonctions. Dans de tels cas, les stocker dans un script R séparé peut permettre de garder votre script d'analyse principal ordonné. Vous pouvez ensuite 'sourcer' ce script pour charger les fonctions.

```
# Sourcer un script séparé  
source("chemin_vers_votre_script_de_fonctions.R")
```

3) GitHub Gist

Pour des fonctions que vous réutilisez fréquemment ou souhaitez partager avec la communauté, les stocker dans un GitHub Gist est une bonne option. Créez un compte sur github, puis créez un gist public à <https://gist.github.com/>. Ensuite, vous pouvez copier-coller votre fonction dans le gist. Finalement, vous pouvez obtenir l'URL du gist et la sourcer dans votre script R en utilisant la fonction `source_gist()` du package `devtools` :

```
# Sourcer depuis un GitHub Gist  
pacman::p_load(devtools)  
devtools::source_gist("https://gist.github.com/kendavidn/a5e1ce486910e6b2dc77a5f")
```

Lorsque vous exécutez le code ci-dessus, cela définira une nouvelle fonction appelée `hello_from_gist()` que nous avons créée pour cette leçon.

```
hello_from_gist("Étudiant")
```

```
## [1] "Hello Étudiant! This is a function from a gist!"
```

Vous pouvez voir le code en allant directement à l'URL : <https://gist.github.com/kendavidn/a5e1ce486910e6b2dc77a5b6bddf87d0>.

Le code dans le gist peut être mis à jour à tout moment, et les changements seront reflétés dans votre script lorsque vous le sourcerez à nouveau.

4) Package

Comme précédemment mentionné, les fonctions peuvent également être stockées dans des packages. C'est une option plus avancée qui nécessite la connaissance du développement de packages R. Pour plus d'informations, voir le manuel [Writing R Extensions](#).

Conclusion !

Félicitations pour avoir suivi la leçon !

Vous avez maintenant les éléments de base pour créer des fonctions personnalisées qui automatisent les tâches répétitives dans vos flux de travail R. Bien sûr, il y a encore beaucoup à apprendre sur les fonctions, mais vous avez maintenant les fondations sur lesquelles construire.

Corrigés

Fonction Âge en Mois

```
annees_en_mois <- function(annees) {  
  mois <- annees * 12  
  return(mois)  
}  
  
# Test  
annees_en_mois(12)
```

```
## [1] 144
```

Fonction Celsius en Fahrenheit

```
celsius_en_fahrenheit <- function(celsius) {  
  fahrenheit <- celsius * 1.8 + 32  
  return(fahrenheit)  
}  
  
# Test  
beaver1 %>%  
  select(temp) %>%  
  mutate(Fahrenheit = celsius_en_fahrenheit(temp))
```

```
##      temp Fahrenheit  
## 1    36.33    97.394  
## 2    36.34    97.412  
## 3    36.35    97.430  
## 4    36.42    97.556  
## 5    36.55    97.790  
## 6    36.69    98.042  
## 7    36.71    98.078  
## 8    36.75    98.150  
## 9    36.81    98.258  
## 10   36.88    98.384  
## 11   36.89    98.402  
## 12   36.91    98.438  
## 13   36.85    98.330  
## 14   36.89    98.402  
## 15   36.89    98.402  
## 16   36.67    98.006  
## 17   36.50    97.700  
## 18   36.74    98.132  
## 19   36.77    98.186  
## 20   36.76    98.168  
## 21   36.78    98.204  
## 22   36.82    98.276  
## 23   36.89    98.402  
## 24   36.99    98.582  
## 25   36.92    98.456  
## 26   36.99    98.582  
## 27   36.89    98.402  
## 28   36.94    98.492  
## 29   36.92    98.456  
## 30   36.97    98.546  
## 31   36.91    98.438  
## 32   36.79    98.222  
## 33   36.77    98.186  
## 34   36.69    98.042  
## 35   36.62    97.916  
## 36   36.54    97.772
```

## 37	36.55	97.790
## 38	36.67	98.006
## 39	36.69	98.042
## 40	36.62	97.916
## 41	36.64	97.952
## 42	36.59	97.862
## 43	36.65	97.970
## 44	36.75	98.150
## 45	36.80	98.240
## 46	36.81	98.258
## 47	36.87	98.366
## 48	36.87	98.366
## 49	36.89	98.402
## 50	36.94	98.492
## 51	36.98	98.564
## 52	36.95	98.510
## 53	37.00	98.600
## 54	37.07	98.726
## 55	37.05	98.690
## 56	37.00	98.600
## 57	36.95	98.510
## 58	37.00	98.600
## 59	36.94	98.492
## 60	36.88	98.384
## 61	36.93	98.474
## 62	36.98	98.564
## 63	36.97	98.546
## 64	36.85	98.330
## 65	36.92	98.456
## 66	36.99	98.582
## 67	37.01	98.618
## 68	37.10	98.780
## 69	37.09	98.762
## 70	37.02	98.636
## 71	36.96	98.528
## 72	36.84	98.312
## 73	36.87	98.366
## 74	36.85	98.330
## 75	36.85	98.330
## 76	36.87	98.366
## 77	36.89	98.402
## 78	36.86	98.348
## 79	36.91	98.438
## 80	37.53	99.554
## 81	37.23	99.014
## 82	37.20	98.960
## 83	37.25	99.050
## 84	37.20	98.960
## 85	37.21	98.978
## 86	37.24	99.032
## 87	37.10	98.780
## 88	37.20	98.960
## 89	37.18	98.924
## 90	36.93	98.474
## 91	36.83	98.294
## 92	36.93	98.474

```

## 93 36.83    98.294
## 94 36.80    98.240
## 95 36.75    98.150
## 96 36.71    98.078
## 97 36.73    98.114
## 98 36.75    98.150
## 99 36.72    98.096
## 100 36.76   98.168
## 101 36.70   98.060
## 102 36.82   98.276
## 103 36.88   98.384
## 104 36.94   98.492
## 105 36.79   98.222
## 106 36.78   98.204
## 107 36.80   98.240
## 108 36.82   98.276
## 109 36.84   98.312
## 110 36.86   98.348
## 111 36.88   98.384
## 112 36.93   98.474
## 113 36.97   98.546
## 114 37.15   98.870

```

Fonction IMC

```

calculer_imc <- function(poids, taille) {
  imc <- poids / (taille^2)
  return(imc)
}

# Test
library(medicaldata)
medicaldata::smartpill %>%
  as_tibble() %>%
  select(Weight, Height) %>%
  mutate(IMC = calculer_imc(Weight, Height))

```

```

## # A tibble: 95 × 3
##       Weight   Height     IMC
##       <dbl>    <dbl>    <dbl>
## 1 102.     183.  0.00305
## 2 102.     180.  0.00314
## 3 68.0     180.  0.00209
## 4 69.9     175.  0.00227
## 5 44.9     152.  0.00193
## 6 94.8     185.  0.00276
## 7 86.2     188.  0.00244
## 8 76.2     165.  0.00280
## 9 74.4     173.  0.00249
## 10 64.9    170.  0.00224
## # i 85 more rows

```

Pratique avec l'Argument ...

```
calculate_calories <- function(carb_grams, protein_grams, fat_grams, ...) {  
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)  
  result_formatted <- format(result, ...)  
  return(result)  
}
```

Débogage d'une Fonction avec Logique Conditionnelle

```
verifier_negatifs <- function(nombres) {  
  if (any(nombres < 0)) {  
    print("x contient des nombres négatifs")  
  } else {  
    print("x ne contient pas de nombres négatifs")  
  }  
}  
  
# Test  
verifier_negatifs(c(8, 3, -2, 5))
```

```
## [1] "x contient des nombres négatifs"
```

```
verifier_negatifs(c(10, 20, 30, 40))
```

```
## [1] "x ne contient pas de nombres négatifs"
```

Pratique de Vérification des Arguments

```
calculate_calories2 <- function(carb_grams = 0, protein_grams = 0, fat_grams = 0) {  
  
  if (!is.numeric(carb_grams)) {  
    stop("carb_grams must be numeric")  
  }  
  
  if (!is.numeric(protein_grams)) {  
    stop("protein_grams must be numeric")  
  }  
  
  if (!is.numeric(fat_grams)) {  
    stop("fat_grams must be numeric")  
  }  
  
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)  
  return(result)  
}
```

Pratique de classification du dosage de l'Isoniazide

```
calculer_dosage_isoniazide2 <- function(poids) {  
  if (any(poids < 30)) stop("Tous les poids doivent être au moins de 30 kg.")  
  
  dosage <- case_when(  
    poids <= 35 ~ 150,  
    poids <= 45 ~ 200,  
    poids <= 55 ~ 300,  
    poids <= 70 ~ 300,  
    TRUE ~ 300  
  )  
  return(dosage)  
}  
  
calculer_dosage_isoniazide2(c(30, 40, 50, 100))
```

```
## [1] 150 200 300 300
```

Contributeurs

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



DANIEL CAMARA

Data Scientist at the GRAPH Network and fellowship as Public Health

researcher at Fiocruz, Brazil
Passionate about lots of things, especially when it involves people leading lives with more equality and freedom



EDUARDO ARAUJO

Student at Universidade Tecnologica Federal do Parana
Passionate about reproducible science and education



LAURE VANCAUWENBERGHE

Data analyst, the GRAPH Network
A firm believer in science for good, striving to ally programming, health and education



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement



GUY WAFEU

R Instructor and Public Health Physician
Committed to improving the quality of data analysis

Références

Certains éléments de cette leçon ont été adaptés des sources suivantes :

- Barnier, Julien. "Introduction à R et au tidyverse." Consulté le 23 mai 2022. <https://juba.github.io/tidyverse>
- Wickham, Hadley; Grolemund, Garrett. "R for Data Science." Consulté le 25 mai 2022. <https://r4ds.had.co.nz/>

This work is licensed under the [Creative Commons Attribution Share Alike](#) license.



Boucles dans R

Introduction
Objectifs d'apprentissage
Packages
Introduction aux boucles <code>for</code>
Les boucles <code>for</code> sont-elles utiles en R ?
Boucler avec un Index
Faire des boucles sur plusieurs vecteurs
Stocker les résultats d'une boucle
Instructions conditionnelles dans les boucles
Techniques rapides pour déboguer des boucles <code>for</code>
Isoler et exécuter une seule itération
Ajouter des Instructions d'Impression à la Boucle
Application Réelle des Boucles 1 : Analyser Plusieurs Jeux de Données
Application Réelle des Boucles 2 : Génération de Plusieurs Graphiques
Conclusion !
Corrigé

Introduction

Au cœur de la programmation se trouve le concept de répéter une tâche plusieurs fois. Une boucle `for` est une des manières fondamentales de le faire. Les boucles permettent une répétition efficace, économisant temps et effort.

Maîtriser ce concept est essentiel pour écrire du code R intelligent et efficace.

Plongeons et améliorez vos compétences en codage !

Objectifs d'apprentissage

À la fin de cette leçon, vous serez capable de :

- Expliquer la syntaxe et la structure d'une boucle `for` basique dans R
- Utiliser des variables d'index pour itérer à travers de multiples vecteurs simultanément dans une boucle
- Intégrer des instructions conditionnelles `if/else` dans une boucle
- Stocker les résultats des boucles dans des vecteurs et des listes
- Appliquer des boucles à des tâches comme l'analyse de multiples jeux de données et la génération de multiples graphiques
- Déboguer des boucles en isolant et en testant des itérations uniques

Packages

Cette leçon nécessitera l'installation et le chargement des packages suivants :

```
# Charger les packages
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, here, openxlsx, tools, outbreaks, medicaldata)
```

Introduction aux boucles for

Commençons par un exemple simple. Supposons que nous avons un vecteur d'âges d'enfants en années, et nous voulons convertir ces âges en mois :

```
ages <- c(7, 8, 9) # Vecteur d'âges en années
```

Nous pouvons faire cela facilement avec l'opération * dans R :

```
ages * 12
```

```
## [1] 84 96 108
```

Mais voyons en détails la manière dont nous pourrions accomplir cela en utilisant une boucle `for` à la place, car c'est (conceptuellement) ce que R fait en arrière plan.

```
for (age in ages) print(age * 12)
```

```
## [1] 84
## [1] 96
## [1] 108
```

Dans cette boucle, `age` est une variable temporaire qui prend la valeur de chaque élément dans `ages` à chaque itération. D'abord, `age` est 7, ensuite 8, puis 9.

Vous pouvez choisir n'importe quel nom pour cette variable :

```
for (nom_aleatoire in ages) print(nom_aleatoire * 12)
```

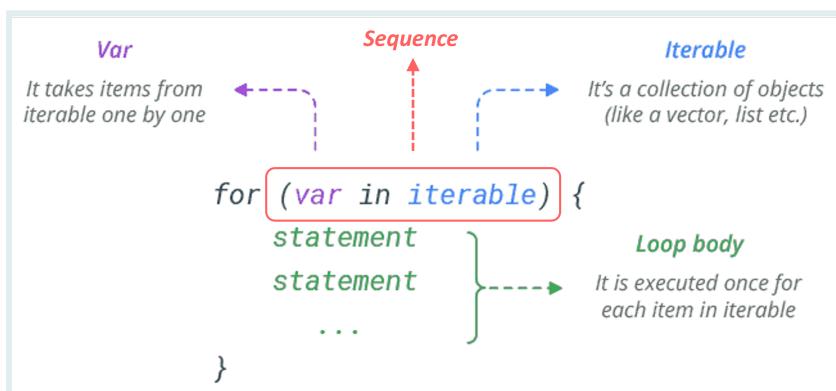
```
## [1] 84
## [1] 96
## [1] 108
```

Si le contenu de la boucle est plus d'une ligne, vous devez utiliser des accolades {} pour indiquer le corps de la boucle.

```
for (age in ages) {  
  age_en_mois = age * 12  
  print(age_en_mois)  
}
```

```
## [1] 84  
## [1] 96  
## [1] 108
```

La structure générale de n'importe quelle boucle for est illustrée dans le diagramme ci-dessous :



Boucle de base pour convertir des heures en minutes

Essayez de convertir des heures en minutes en utilisant une boucle for. Commencez avec ce vecteur d'heures :

```
hours <- c(3, 4, 5) # Vecteur d'heures  
# Votre code ici  
  
for ____ # convertissez les heures en minutes et affichez
```

Les boucles peuvent être imbriquées les unes dans les autres. Par exemple :

```
for (i in 1:2) {  
  for (j in 1:2) {  
    print(i * j)  
  }  
}
```

```
## [1] 1  
## [1] 2
```

```
## [1] 2  
## [1] 4
```

Cela crée une combinaison de valeurs i et j comme le montre ce tableau :

i	j	i * j
1	1	1
1	2	2
2	1	2
2	2	4

Les boucles imbriquées sont cependant moins courantes et ont souvent des alternatives plus efficaces.

Les boucles for sont-elles utiles en R ?

Bien que les boucles `for` soient fondamentales dans de nombreux langages de programmation, leur utilisation en R est quelque peu moins fréquente. Cela est dû au fait que R gère intrinsèquement les opérations *vectorisées*, en appliquant automatiquement une fonction à chaque élément d'un vecteur.

Par exemple, notre conversion initiale d'âge pourrait être réalisée sans boucle :

```
ages * 12
```

```
## [1] 84 96 108
```

De plus, R traite généralement avec des jeux de données plutôt qu'avec des vecteurs bruts. Pour les jeux de données, nous utilisons souvent des fonctions du package `tidyverse` pour appliquer des opérations sur les colonnes :

```
ages_df <- tibble(age = ages)  
ages_df %>%  
  mutate(age_months = age * 12)
```

```
## # A tibble: 3 × 2  
##       age   age_months  
##     <dbl>      <dbl>  
## 1     7        84  
## 2     8        96  
## 3     9       108
```

Cependant, il existe des situations où les boucles sont utiles, en particulier lorsqu'on travaille avec plusieurs jeux de données ou des objets non-dataframe (parfois appelés objets *non rectangulaires*).

Nous explorerons ces cas plus loin dans la leçon, mais d'abord nous allons passer plus de temps à nous familiariser avec les boucles en utilisant des exemples de simulation.

Boucles vs cartographie des fonctions

Il est important de noter que les boucles peuvent souvent être remplacées par des fonctions personnalisées qui sont ensuite appliquées à travers un vecteur ou un jeu de données.

Nous enseignons néanmoins les boucles parce qu'elles sont assez faciles à apprendre, à comprendre et à déboguer, même pour les débutants.

Boucler avec un Index

Il est souvent utile de parcourir un vecteur en utilisant un index, qui est un compteur qui suit l'itération en cours.

Regardons à nouveau notre vecteur ages que nous voulons convertir en mois :

```
ages <- c(7, 8, 9) # Vecteur d'âges en années
```

Pour utiliser des index dans une boucle, nous créons d'abord une séquence qui représente chaque position dans le vecteur :

```
1:length(ages) # Crée une séquence d'index de la même longueur que ages
```

```
## [1] 1 2 3
```

```
index <- 1:length(ages)
```

Maintenant, index a les valeurs 1, 2, 3, correspondant aux positions dans ages. Nous utilisons ceci dans une boucle for comme suit :

```
for (i in index) {  
  print(ages[i] * 12)  
}
```

```
## [1] 84  
## [1] 96
```

```
## [1] 108
```

Dans ce code, ages[i] fait référence au ième élément de notre liste ages.

Le nom de la variable i est arbitraire. Nous aurions pu utiliser j ou index ou position ou tout autre nom.

```
for (position in index) {  
  print(ages[position] * 12)  
}
```

```
## [1] 84  
## [1] 96  
## [1] 108
```

Souvent, nous n'avons pas besoin de créer une variable séparée pour les index. Nous pouvons simplement

Des boucles basées sur les index sont utiles pour travailler avec plusieurs vecteurs en même temps. Nous verrons cela dans la section suivante.

Boucle indexée d'heures en minutes

Réécrivez votre boucle de la dernière question en utilisant des index :

```
heures <- c(3, 4, 5) # Vecteur d'heures  
  
# Votre code ici  
  
for __ {  
  __
```

La fonction seq_along() est un raccourci pour créer une séquence d'index. Elle est équivalente à 1:length() :

```
# Ces deux sont équivalents :  
seq_along(ages)
```

```
## [1] 1 2 3
```

```
1:length(ages)
```

```
## [1] 1 2 3
```

Faire des boucles sur plusieurs vecteurs

Faire des boucles avec des index nous permet de travailler avec plusieurs vecteurs simultanément. Supposons que nous ayons des vecteurs pour les âges et les tailles :

```
ages <- c(7, 8, 9) # âges en années  
tailles <- c(120, 130, 140) # tailles en cm
```

Nous pouvons faire des boucles sur les deux en utilisant la méthode d'index :

```
for(i in 1:length(ages)) {  
  age <- ages[i]  
  taille <- tailles[i]  
  
  print(paste("Âge :", age, "Taille :", taille))  
}
```

```
## [1] "Âge : 7 Taille : 120"  
## [1] "Âge : 8 Taille : 130"  
## [1] "Âge : 9 Taille : 140"
```

À chaque itération : - i est l'index - Nous extrayons le i-ème élément de chaque vecteur et l'imprimons.

Alternativement, nous pouvons sauter l'assignation de variable et utiliser les index directement dans la fonction `print()` :

```
for(i in 1:length(ages)) {  
  print(paste("Âge :", ages[i], "Taille :", tailles[i]))  
}
```

```
## [1] "Âge : 7 Taille : 120"  
## [1] "Âge : 8 Taille : 130"  
## [1] "Âge : 9 Taille : 140"
```

Boucle de calcul de l'IMC

En utilisant une boucle `for`, calculez l'Indice de Masse Corporelle (IMC) des trois individus ci-dessous. La formule de l'IMC est $\text{IMC} = \text{poids} / (\text{taille} ^ 2)$.

```

poids <- c(30, 32, 35) # Poids en kg
tailles <- c(1.2, 1.3, 1.4) # Tailles en mètres

for(i in _____) {
    _____
    print(paste("Poids :", ___, 
                "Taille :", ___, 
                "IMC :", ___, 
                ""))
}

```

Stocker les résultats d'une boucle

Dans la plupart des cas, vous voudrez stocker les résultats d'une boucle plutôt que de les imprimer comme nous l'avons fait ci-dessus. Voyons comment faire cela.

Considérons notre exemple de conversion d'âge en mois :

```

ages <- c(7, 8, 9)

for (age in ages) {
  print(paste(age * 12, "mois"))
}

```

```

## [1] "84 mois"
## [1] "96 mois"
## [1] "108 mois"

```

Pour stocker ces âges convertis, nous créons d'abord un vecteur vide :

```

ages_mois <- vector(mode = "numeric", length = length(ages))
# Cela peut aussi être écrit comme :
ages_mois <- vector("numeric", length(ages))

ages_mois # Affiche le vecteur vide

```

```

## [1] 0 0 0

```

Cela crée un vecteur numérique de la même longueur que ages, initialement rempli de zéros. Pour stocker une valeur dans le vecteur, nous faisons ce qui suit :

```
ages_mois[1] <- 99 # Stocker 99 dans le premier élément de ages_mois  
ages_mois[2] <- 100 # Stocker 100 dans le deuxième élément de ages_mois  
ages_mois
```

```
## [1] 99 100 0
```

Maintenant, exécutons la boucle, en stockant les résultats dans ages_mois :

```
ages_mois <- vector("numeric", length(ages))  
  
for (i in 1:length(ages)) {  
  ages_mois[i] <- ages[i] * 12  
}  
ages_mois
```

```
## [1] 84 96 108
```

Dans cette boucle :

- Lors de la première itération, *i* est 1. Nous multiplions le premier élément de ages par 12 et le stockons dans le premier élément de ages_mois.
- Ensuite *i* est 2, puis 3. À chaque itération, nous multiplions l'élément correspondant de ages par 12 et le stockons dans l'élément correspondant de ages_mois.

Conversion de la taille de cm en m

Utilisez une boucle for pour convertir les mesures de taille de cm en m. Stockez les résultats dans un vecteur appelé height_meters.

```
height_cm <- c(180, 170, 190, 160, 150) # Tailles en cm  
  
height_m <- vector(_____) # vecteur numérique de même longueur que  
height_cm  
  
for ____ {  
  height_m[i] <- _____  
}
```

```
## Error: <text>:3:21: unexpected input  
## 2:  
## 3: height_m <- vector(_____  
##
```

Afin de sauvegarder les résultats de votre itération, vous devez créer votre objet vide **à l'extérieur** de la boucle. Sinon, vous ne sauvegarderez que le résultat de la dernière itération.

C'est une erreur commune. Considérez l'exemple ci-dessous :

```
ages <- c(7, 8, 9)

for (i in 1:length(ages)) {
  ages_mois <- vector("numeric", length(ages))
  ages_mois[i] <- ages[i] * 12
}
ages_mois
```

```
## [1] 0 0 108
```

Voyez-vous le problème ?

::: note latérale Si vous êtes pressé, vous pouvez sauter l'utilisation de la fonction `vector()` et initialiser votre vecteur avec `c()` à la place, puis le remplir progressivement avec des valeurs par index :

```
ages_mois <- c()

for (i in 1:length(ages)) {
  ages_mois[i] <- ages[i] * 12
}
ages_mois
```

```
## [1] 84 96 108
```

Et vous pouvez également sauter l'index et utiliser `c()` pour ajouter des valeurs à la fin du vecteur :

```
ages_mois <- c()

for (age in ages) {
  ages_mois <- c(ages_mois, age * 12)
}
ages_mois
```

```
## [1] 84 96 108
```

Cependant, dans les deux cas, R ne connaît pas la longueur finale du vecteur lorsqu'il passe par les itérations, il doit donc réallouer la mémoire à chaque itération. Cela peut entraîner des performances lentes si vous travaillez avec de grands vecteurs. :::

Instructions conditionnelles dans les boucles

Tout comme les instructions `if` peuvent être utilisées dans les fonctions, elles peuvent être intégrées dans les boucles.

Considérez cet exemple :

```
age_vec <- c(2, 12, 17, 24, 60) # Vecteur d'âges

for (age in age_vec) {
  if (age < 18) print(paste("Enfant, Âge", age))
}

## [1] "Enfant, Âge 2"
## [1] "Enfant, Âge 12"
## [1] "Enfant, Âge 17"
```

Il est souvent plus clair d'utiliser des accolades pour indiquer le corps de l'instruction `if`. Cela nous permet également d'ajouter plus de lignes de code au corps de l'instruction `if` :

```
for (age in age_vec) {
  if (age < 18) {
    print("Traitement :")
    print(paste("Enfant, Âge", age))
  }
}
```

```
## [1] "Traitement :"
## [1] "Enfant, Âge 2"
## [1] "Traitement :"
## [1] "Enfant, Âge 12"
## [1] "Traitement :"
## [1] "Enfant, Âge 17"
```

Ajoutons une autre condition pour classer comme 'Enfant' ou 'Adolescent' :

```
for (age in age_vec) {
  if (age < 13) {
    print(paste("Enfant, Âge", age))
  } else if (age >= 13 && age < 18) {
    print(paste("Adolescent, Âge", age))
  }
}
```

```
## [1] "Enfant, Âge 2"
## [1] "Enfant, Âge 12"
```

```
## [1] "Adolescent, Âge 17"
```

Nous pouvons inclure une seule instruction `else` à la fin pour prendre en compte toutes les autres tranches d'âge :

```
for (age in age_vec) {  
  if (age < 13) {  
    print(paste("Enfant, Âge", age))  
  } else if (age >= 13 && age < 18) {  
    print(paste("Adolescent, Âge", age))  
  } else {  
    print(paste("Adulte, Âge", age))  
  }  
}
```

```
## [1] "Enfant, Âge 2"  
## [1] "Enfant, Âge 12"  
## [1] "Adolescent, Âge 17"  
## [1] "Adulte, Âge 24"  
## [1] "Adulte, Âge 60"
```

Pour stocker ces classifications, nous pouvons créer un vecteur vide et utiliser une boucle basée sur l'index pour stocker les résultats :

```
age_class <- vector("character", length(age_vec)) # Créez un vecteur vide  
for (i in 1:length(age_vec)) {  
  if (age_vec[i] < 13) {  
    age_class[i] <- "Enfant" # Enfant  
  } else if (age_vec[i] >= 13 && age_vec[i] < 18) {  
    age_class[i] <- "Adolescent" # Adolescent  
  } else {  
    age_class[i] <- "Adulte" # Adulte  
  }  
}  
age_class
```

```
## [1] "Enfant"      "Enfant"      "Adolescent"  "Adulte"      "Adulte"
```

Classification de la température

Vous disposez d'un vecteur de températures corporelles en Celsius. Classez chaque température comme 'Hypothermie', 'Normale' ou 'Fièvre' en utilisant une boucle `for` combinée avec des instructions `if` et `else`.

Utilisez ces règles :

- En dessous de 36,5 °C : 'Hypothermie' (Hypothermie)
- Entre 36,5 °C et 37,5 °C : 'Normale' (Normale)

- Au-dessus de 37,5 °C : 'Fièvre' (Fièvre)

```
body_temps <- c(35, 36.5, 37, 38, 39.5) # Températures corporelles en Celsius
classif_vec <- vector(_____) # vec caractère, longueur de
body_temps
for (i in 1:length(____)) {
  # Ajoutez votre logique if-else ici
  if (tempe_corporelle[i] < 36.5) {
    out <- "Hypothermie" # Hypothermie
  } ## ajoutez d'autres conditions

  # Dernière instruction d'impression
  classif_vec[i] <- paste(tempe_corporelle[i], "°C est", out)
}
classif_vec
```

```
## Error: <text>:2:24: unexpected input
## 1: body_temps <- c(35, 36.5, 37, 38, 39.5) # Températures corporelles en
Celsius
## 2: classif_vec <- vector(_____
##
```

Un résultat attendu est ci-dessous

```
35°C est Hypothermie
36.5°C est Normale
37°C est Normale
38°C est Fièvre
39.5°C est Fièvre
```

Techniques rapides pour déboguer des boucles for

L'édition et le débogage efficaces sont cruciaux lorsqu'on travaille avec des boucles `for` en R. Il existe de nombreuses approches pour cela, mais pour le moment, nous allons montrer deux des plus simples :

- Isoler et exécuter une seule itération de la boucle
- Ajouter des instructions `print()` à la boucle pour imprimer les valeurs des variables à chaque itération

Isoler et exécuter une seule itération

Considérez cette boucle que nous avons vue précédemment :

```

age_vec <- c(2, 12, 17, 24, 60) # Vecteur d'âges
age_class <- vector("character", length(age_vec))

for (i in 1:length(age_vec)) {
  if (age_vec[i] < 18) {
    age_class[i] <- "Enfant" # Enfant
  } else {
    age_class[i] <- "Adulte" # Adulte
  }
}
age_class

```

```
## [1] "Enfant" "Enfant" "Enfant" "Adulte" "Adulte"
```

Voyons un exemple d'erreur que nous pourrions rencontrer en utilisant la boucle :

```

# Vecteur d'âge issu du jeu de données fluH7N9_china_2013
flu_dat <- outbreaks::fluH7N9_china_2013
head(flu_dat)

```

	case_id	date_of_onset	date_of_hospitalisation	date_of_outcome	outcome
gender	age	province			
## 1	1	2013-02-19	<NA>	2013-03-04	Death
m	87	Shanghai			
## 2	2	2013-02-27	2013-03-03	2013-03-10	Death
m	27	Shanghai			
## 3	3	2013-03-09	2013-03-19	2013-04-09	Death
f	35	Anhui			
## 4	4	2013-03-19	2013-03-27	<NA>	<NA>
f	45	Jiangsu			
## 5	5	2013-03-19	2013-03-30	2013-05-15	Recover
f	48	Jiangsu			
## 6	6	2013-03-21	2013-03-28	2013-04-26	Death
f	32	Jiangsu			

```

flu_dat_age <- flu_dat$age
age_class <- vector("character", length(flu_dat_age))
for (i in 1:length(flu_dat_age)) {
  if (flu_dat_age[i] < 18) {
    age_class[i] <- "Enfant" # Enfant
  } else {
    age_class[i] <- "Adulte" # Adulte
  }
}

```

```
## Warning in Ops.factor(flu_dat_age[i], 18): '<' not meaningful for factors
```

```
## Error in if (flu_dat_age[i] < 18) { : missing value where TRUE/FALSE needed
```

Nous obtenons cette erreur :

```
Erreurs dans if (flu_dat_age[i] < 18) { :
  valeur manquante là où TRUE / FALSE est requis
De plus : Message d'avertissement :
Dans Ops.factor(flu_dat_age[i], 18) :
  '<' n'est pas pertinent pour des variables facteurs
```

Vous savez peut-être déjà ce que signifie cette erreur, mais supposons que vous ne le sachiez pas.

Nous pouvons entrer dans la boucle et parcourir manuellement la première itération pour voir ce qui se passe :

```
for (i in 1:length(flu_dat_age)) {

  # ► Exécutez à partir de cette ligne
  i <- 1 # Définir manuellement i à 1

  # Ensuite, soulignez `flu_dat_age[i]` et appuyez sur Ctrl + Entrée pour
  exécuter juste ce code
  # Après cela, soulignez et exécutez `flu_dat_age[i] < 18`
```

```
if (flu_dat_age[i] < 18) {
  age_class[i] <- "Enfant" # Enfant
} else {
  age_class[i] <- "Adulte" # Adulte
}
```

```
}
```

```
## Warning in Ops.factor(flu_dat_age[i], 18): '<' not meaningful for factors
```

```
## Error in if (flu_dat_age[i] < 18) { : missing value where TRUE/FALSE needed
```

En suivant le processus ci-dessus, nous pouvons voir que `flu_dat_age` est un facteur, et non un vecteur numérique. Nous pouvons manuellement changer cela, en plein milieu du processus de débogage. C'est une bonne idée de convertir d'abord le facteur en vecteur de caractères, puis en vecteur numérique. Sinon, nous pourrions obtenir des résultats inattendus.

Considérez :

```
flu_dat_age[75]
```

```
## [1] ?  
## 61 Levels: ? 15 2 21 25 26 27 31 32 34 35 36 37 38 4 41 43 45 47 48 49 50  
51 52 53 54 55 56 57 58 59 6 60 ... 91
```

```
as.numeric(flu_dat_age[75])
```

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

```
# `?`, qui signifie manquant dans ce cas, est converti en 1, car il est le  
premier niveau du facteur  
  
# Nous avons donc besoin :  
as.numeric(as.character(flu_dat_age[75]))
```

```
## Warning: NAs introduced by coercion
```

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

Maintenant essayons de corriger la boucle, et exécutons juste la première itération à nouveau :

```
for (i in 1:length(flu_dat_age)) {  
  
  # ► Exécuter à partir de cette ligne  
  i <- 1 # Fixer manuellement i à 1  
  
  age_num <- as.numeric(as.character(flu_dat_age[i]))  
  
  # Ensuite, surligner `age_num < 18` et appuyer sur Ctrl + Entrée  
  if (age_num < 18) {  
    age_class[i] <- "Enfant"  
  } else {  
    age_class[i] <- "Adulte"  
  }  
}
```

Maintenant, la première itération fonctionne, mais voyons ce qui se passe lorsque nous exécutons la boucle entière :

```

age_class <- vector("character", length(flu_dat_age))

for (i in 1:length(flu_dat_age)) {
  age_num <- as.numeric(as.character(flu_dat_age[i]))

  if (age_num < 18) {
    age_class[i] <- "Enfant"
  } else {
    age_class[i] <- "Adulte"
  }
}
head(age_class)

```

Erreur dans if (age_num < 18) { :
 valeur manquante là où TRUE / FALSE est requis
 De plus : Message d'avis :
 Dans as.numeric(as.character(flu_dat_age[i])) :
 NAs introduits lors de la conversion automatique

Encore une fois, vous savez peut-être déjà ce que signifie cette erreur, mais supposons que non. Nous allons essayer notre prochaine technique de débogage.

Ajouter des Instructions d'Impression à la Boucle

Dans la dernière section, nous avons vu que la boucle fonctionne bien pour la première itération, mais semble échouer lors d'une itération ultérieure.

Pour détecter sur quelle itération elle échoue, nous pouvons ajouter des instructions `print()` à la boucle :

```

for (i in 1:length(flu_dat_age)) {

  print(i) # Imprimer le numéro de l'itération
  age_num <- as.numeric(as.character(flu_dat_age[i]))

  print(age_num) # Imprimer la valeur de age_num

  if (age_num < 18) {
    age_class[i] <- "Enfant"
  } else {
    age_class[i] <- "Adulte"
  }

  print(age_class[i]) # Imprimer la valeur de la sortie
}
head(age_class)

```

Maintenant, en inspectant la sortie, nous pouvons voir que la boucle échoue à la 74ème itération :

```
[1] 73 ➔ 73ème itération
[1] 43 ➔ Valeur de age_num
[1] "Adulte, Âge 43" ➔ Valeur de la sortie à la 73ème itération
[1] 74 ➔ 74ème itération
[1] NA ➔ Valeur de age_num
```

Cela se produit parce que la 74ème valeur de `flu_dat_age` est NA (à cause de notre conversion de facteur en numérique), donc R ne peut pas évaluer si elle est inférieure à 18.

Nous pouvons corriger cela en ajoutant une instruction `if` pour vérifier les valeurs NA :

```
for (i in 1:length(flu_dat_age)) {

  age_num <- as.numeric(as.character(flu_dat_age[i]))

  if (is.na(age_num)) {
    age_class[i] <- "NA"
  } else if (age_num < 18) {
    age_class[i] <- "Enfant"
  } else {
    age_class[i] <- "Adulte"
  }
}
```

```
## Warning: NAs introduced by coercion
## Warning: NAs introduced by coercion
```

```
# Vérifier la 74ème valeur de age_class
age_class[74]
```

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

Super ! Maintenant, nous avons corrigé l'erreur.

Comme vous pouvez le voir, même avec notre boucle "jouet", le débogage peut être un processus long. Comme le disait votre mère : "La programmation, c'est 98 % de débogage et 2 % d'écriture de code."

R offre plusieurs autres techniques pour diagnostiquer et gérer les erreurs :

- Les fonctions `try()` et `tryCatch()` permettent de capturer les erreurs tout en continuant l'exécution de la boucle.
- La fonction `browser()` met la boucle en pause à un point désigné, permettant une exécution pas à pas.

Ce sont des méthodes plus avancées, et bien quelques ne soient pas couvertes ici, vous pouvez vous référer à la documentation de R pour des conseils supplémentaires lorsque cela est nécessaire. Ou consultez le livre [Advanced R](#) de Hadley Wickham.

Application Réelle des Boucles 1 : Analyser Plusieurs Jeux de Données

Maintenant que vous avez une solide compréhension des boucles `for`, appliquons nos connaissances à une tâche de boucle plus réaliste : travailler avec plusieurs jeux de données.

Nous avons un dossier de fichiers CSV contenant des données sur les décès liés au VIH pour les municipalités de Colombie.

municipality	death_location	birth_date	death_year	death_month
Municipal head	Hospital/clinic	1956-05-26	2012	Sep
Municipal head	Hospital/clinic	1983-10-10	2012	Mar

Imaginez que nous devions compiler un seul tableau comprenant les informations suivantes pour chaque jeu de données : le nombre de lignes (nombre de décès), le nombre de colonnes et les noms de toutes les colonnes.

Nous pourrions faire cela un par un, mais cela serait fastidieux et source d'erreurs. Au lieu de cela, nous pouvons utiliser une boucle pour automatiser le processus.

Tout d'abord, listons les fichiers dans le dossier :

```
chemins_donnees_colom <- list.files(here("data/colombia_hiv_deaths"),
                                         full.names = TRUE)
head(chemins_donnees_colom) # Montrer les 6 premiers chemins de fichiers
```

```
## [1]
"/Users/kendavidn/Dropbox/tgc_github_projects/fdar_staging/FDAR_FR_loops/data/c-
```

```
olombia_hiv_deaths/Aguadas.csv"
## [2]
"/Users/kendavidn/Dropbox/tgc_github_projects/fdar_staging/FDAR_FR_loops/data/colombia_hi
## [3]
"/Users/kendavidn/Dropbox/tgc_github_projects/fdar_staging/FDAR_FR_loops/data/colombia_hi
## [4]
"/Users/kendavidn/Dropbox/tgc_github_projects/fdar_staging/FDAR_FR_loops/data/colombia_hi
## [5]
"/Users/kendavidn/Dropbox/tgc_github_projects/fdar_staging/FDAR_FR_loops/data/colombia_hi
## [6]
"/Users/kendavidn/Dropbox/tgc_github_projects/fdar_staging/FDAR_FR_loops/data/colombia_hi
```

Maintenant, importons un jeu de données comme exemple pour démontrer ce que nous voulons atteindre. Une fois cela fait, nous pourrons appliquer le même processus à tous les jeux de données.

```
donnees_colom <- read_csv(chemins_donnees_colom[1]) # Importer le premier jeu
de données
```

```
## Rows: 2 Columns: 15
## — Column specification

## Delimiter: ","
## chr (9): municipality, death_location, death_month, municipality_code,
primary_cause_death_description, prim...
## dbl (2): death_year, death_day
## lgl (3): tertiary_cause_death_description,
quaternary_cause_death_description, quaternary_cause_death_code
## date (1): birth_date
##
## 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.
```

```
donnees_colom
```

```
## # A tibble: 2 × 15
##   municipality   death_location   birth_date death_year
##   <chr>           <chr>           <date>      <dbl>
## 1 Municipal head Hospital/clinic 1956-05-26     2012
## 2 Municipal head Hospital/clinic 1983-10-10     2012
## # i 11 more variables: death_month <chr>, death_day <dbl>,
## #   municipality_code <chr>, ...
```

Ensuite, nous appliquons une série de fonctions R pour recueillir les informations que nous voulons de chaque jeu de données :

```
file_path_sans_ext(basename(chemins_donnees_colom[1])) # Nom du jeu de
données/municipalité
```

```

## [1] "Aguadas"

nrow(donnees_colom) # Nombre de lignes, ce qui équivaut au nombre de décès

## [1] 2

ncol(donnees_colom) # Nombre de colonnes

## [1] 15

paste(names(donnees_colom), collapse = ", ") # Noms de toutes les colonnes

## [1] "municipality, death_location, birth_date, death_year, death_month,
death_day, municipality_code, primary_cause_death_description,
primary_cause_death_code, secondary_cause_death_description,
secondary_cause_death_code, tertiary_cause_death_description,
tertiary_cause_death_code, quaternary_cause_death_description,
quaternary_cause_death_code"

```

`basename` : extrait le nom de fichier d'un chemin de fichier.

```

chemins_donnees_colom[1]

## [1]
"/Users/kendavidn/Dropbox/tgc_github_projects/fdar_staging/FDAR_FR_loops/data/colombia_hi

basename(chemins_donnees_colom[1])

## [1] "Aguadas.csv"

```

Et `file_path_sans_ext` du package `{tools}` supprime l'extension de fichier des noms de fichiers. Nous l'utilisons avec `basename` pour obtenir le nom de la municipalité.

```
file_path_sans_ext(basename(chemins_donnees_colom[1]))
```

```
## [1] "Aguadas"
```

Maintenant, nous devons créer un dataframe avec ces informations. Nous pouvons utiliser la fonction `tibble` pour faire cela :

```
ligne_unique <-  
  tibble(jeu_de_donnees = basename(chemins_donnees_colom[1]),  
         n_deces = nrow(donnees_colom),  
         n_colonnes = ncol(donnees_colom),  
         noms_colonnes = paste(names(donnees_colom), collapse = ", "))  
ligne_unique
```

```
## # A tibble: 1 × 4  
##   jeu_de_donnees n_deces n_colonnes noms_colonnes  
##   <chr>          <int>     <int> <chr>  
## 1 Aguadas.csv      2         15 municipality, death_loc...
```

Nous allons donc devoir répéter ce processus pour chaque jeu de données. Dans la boucle, nous stockerons chaque dataframe à une seule ligne dans une liste, puis les combinerons à la fin. Rappelons que les listes sont des objets R qui peuvent contenir d'autres objets R, y compris des dataframes.

Initialisons cette liste vide maintenant :

```
liste_dataframes <- vector("list", length(chemins_donnees_colom))  
head(liste_dataframes) # Montrer les 6 premiers éléments
```

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

Ajoutons la première dataframe à une seule ligne à la liste :

```
liste_dataframes[[1]] <- ligne_unique
```

Maintenant, si nous regardons la liste, nous voyons que le premier élément est la dataframe à une seule ligne :

```
head(liste_dataframes)
```

```
## [[1]]
## # A tibble: 1 × 4
##   jeu_de_donnees n_deces n_colonnes noms_colonnes
##   <chr>          <int>     <int> <chr>
## 1 Aguadas.csv      2         15 municipality, death_loc...
##
## [[2]]
## NULL
##
## [[3]]
## NULL
##
## [[4]]
## NULL
##
## [[5]]
## NULL
##
## [[6]]
## NULL
```

Et nous pouvons accéder à la dataframe en sous-ensemble de la liste :

```
liste_dataframes[[1]]
```

```
## # A tibble: 1 × 4
##   jeu_de_donnees n_deces n_colonnes noms_colonnes
##   <chr>          <int>     <int> <chr>
## 1 Aguadas.csv      2         15 municipality, death_loc...
```

Notez l'utilisation de doubles crochets pour accéder aux éléments de la liste.

Nous avons maintenant toutes les pièces dont nous avons besoin pour créer une boucle qui traitera chaque jeu de données et stockera les résultats dans une liste. Allons-y !

```

for (i in 1:length(chemins_donnees_colom)) {
  chemin <- chemins_donnees_colom[i]

  # Importation
  donnees_colom <- read_csv(chemin)

  # Récupération des infos
  n_deces <- nrow(donnees_colom)
  n_colonnes <- ncol(donnees_colom)
  noms_colonnes <- paste(names(donnees_colom), collapse = ", ")

  # Création du dataframe pour cet ensemble de données
  ligne_donnees_vih <- tibble(jjeu_de_donnees =
    file_path_sans_ext(basename(chemin)),
    n_deces = n_deces,
    n_colonnes = n_colonnes,
    noms_colonnes = noms_colonnes)

  # Stockage dans la liste
  liste_dataframes[[i]] <- ligne_donnees_vih
}

```

Vérifions la liste :

```
head(liste_dataframes, 2) # Montrer les 2 premiers éléments
```

```

## [[1]]
## # A tibble: 1 × 4
##   jjeu_de_donnees n_deces n_colonnes noms_colonnes
##   <chr>           <int>     <int> <chr>
## 1 Aguadas          2         15 municipality, death_lo...
##
## [[2]]
## # A tibble: 1 × 4
##   jjeu_de_donnees n_deces n_colonnes noms_colonnes
##   <chr>           <int>     <int> <chr>
## 1 Anserma          15        16 municipality, death_lo...

```

Et maintenant, nous pouvons combiner tous les dataframes de la liste en un seul dataframe final. Cela peut être fait avec la fonction `bind_rows` du package `{dplyr}` :

```
donnees_colom_finales <- bind_rows(liste_dataframes)
donnees_colom_finales
```

```

## # A tibble: 25 × 4
##   jjeu_de_donnees n_deces n_colonnes noms_colonnes
##   <chr>           <int>     <int> <chr>
## 1 Aguadas          2         15 municipality, death_l...
## 2 Anserma          15        16 municipality, death_l...
## 3 Aranzazu          2         16 municipality, death_l...
## 4 Belalcázar        4         14 municipality, death_l...

```

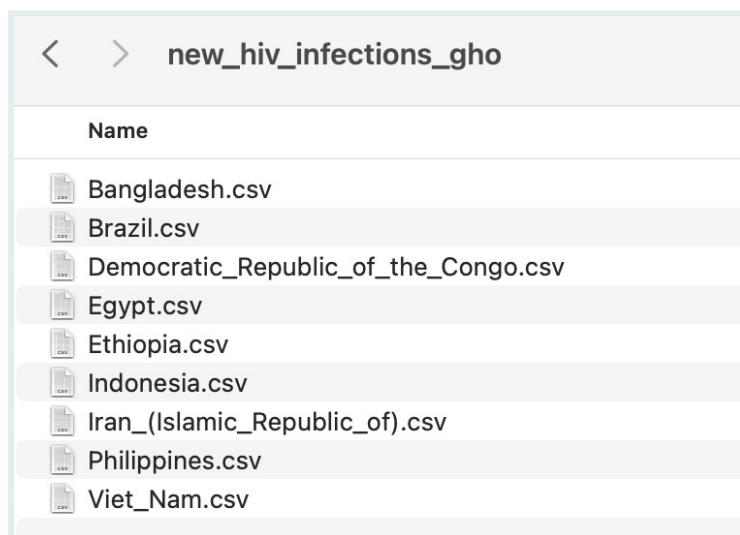
```

## 5 Chinchiná          62      17 municipality, death_l...
## 6 Filadelfia         5       15 municipality, death_l...
## 7 La Dorada          46      16 municipality, death_l...
## 8 La Merced          3       17 municipality, death_l...
## 9 Manizales          199     17 municipality, death_l...
## 10 Manzanares         3       14 municipality, death_l...
## # i 15 more rows

```

Propriétés du fichier

Vous avez un dossier contenant des fichiers CSV avec des données sur les cas de VIH, provenant de l'OMS.



En utilisant les principes appris, vous écrirez une boucle qui extrait les informations suivantes de chaque jeu de données et les stocke dans un seul dataframe :

- Le nom du jeu de données (c.-à-d. le pays)
- La taille du jeu de données en octets
- La date de dernière modification du jeu de données

Vous pouvez utiliser les fonctions `file.size()` et `file.mtime()` pour obtenir les deux dernières informations. Par exemple :

```
file.size(here("data/new_hiv_infections_gho/Bangladesh.csv"))
```

```
## [1] 6042
```

```
file.mtime(here("data/new_hiv_infections_gho/Bangladesh.csv"))
```

```
## [1] "2023-12-11 17:34:28 GMT"
```

Notez que vous n'avez pas besoin d'importer les CSV pour obtenir ces informations.

```
# Lister les fichiers
csv_files <- list.files(path = "data/new_hiv_infections_gho",
                        )
for (i in _____) {
  path <- csv_files[i]
  # Obtenir le nom du pays. Conseil : utilisez file_path_sans_ext et basename
  nom_pays <- _____
  # Obtenir la taille du fichier et la date de modification
  taille <- _____
  date <- _____
  # Dataframe pour cette itération. Conseil : utilisez tibble() pour combiner
  # les objets ci-dessus
  vih_ligne_df <- _____
  # Stocker dans la liste. Conseil : utilisez des doubles crochets et l'index
  i
  liste_dataframes_____ <- vih_ligne_df
}
# Combiner en un seul dataframe
infos_fichiers_vih_final <- bind_rows(liste_dataframes)
```

Boucle de Filtrage des Données

Vous travaillerez à nouveau avec le dossier des jeux de données sur le VIH de la question précédente. Voici un exemple d'un des jeux de données par pays de ce dossier :

```
donnees_bangla <- read_csv(here("data/new_hiv_infections_gho/Bangladesh.csv"))
donnees_bangla
```

```
## # A tibble: 89 × 5
##   Continent    Country   Year Sex
##   <chr>        <chr>     <dbl> <chr>
## 1 South-East Asia Bangladesh 2022 Female
## 2 South-East Asia Bangladesh 2022 Both sexes
## 3 South-East Asia Bangladesh 2022 Male
## 4 South-East Asia Bangladesh 2021 Female
## 5 South-East Asia Bangladesh 2021 Both sexes
## 6 South-East Asia Bangladesh 2021 Male
## 7 South-East Asia Bangladesh 2020 Female
## 8 South-East Asia Bangladesh 2020 Both sexes
## 9 South-East Asia Bangladesh 2020 Male
## 10 South-East Asia Bangladesh 2019 Female
```

```
## # i 79 more rows
## # i 1 more variable: NewHIVCases <chr>
```

Votre tâche est de compléter le modèle de boucle ci-dessous afin qu'il : - Importe chaque CSV du dossier - Filtre les données uniquement au sexe "Féminin" - Sauvegarde chaque jeu de données filtré comme un CSV dans votre dossier `sorties`

Notez que dans ce cas, vous n'avez pas besoin de stocker les sorties dans une liste, car vous importez, modifiez puis exportez directement chaque jeu de données.

```
# Lister les fichiers
csv_files <- list.files(path = "data/new_hiv_infections_gho",
                         pattern = "*.csv", full.names = TRUE)

for (fichier in _____) {

  # Importer les données. Conseil : utilisez read_csv avec la variable
  `fichier` comme chemin
  donnees_vih _____

  # Filtrer. Conseil : utilisez filter() et la variable `Sexe`
  donnees_vih_filtrees <- _____

  # Nommer le fichier de sortie
  # Cette ligne est faite pour vous, mais assurez-vous de la comprendre
  nom_fichier_sortie <- paste0(here(), "sorties/", "Feminin_",
                                basename(fichier))

  # Exporter.
  write_csv(donnees_vih_filtrees, nom_fichier_sortie)
}
```

Application Réelle des Boucles 2 : Génération de Plusieurs Graphiques

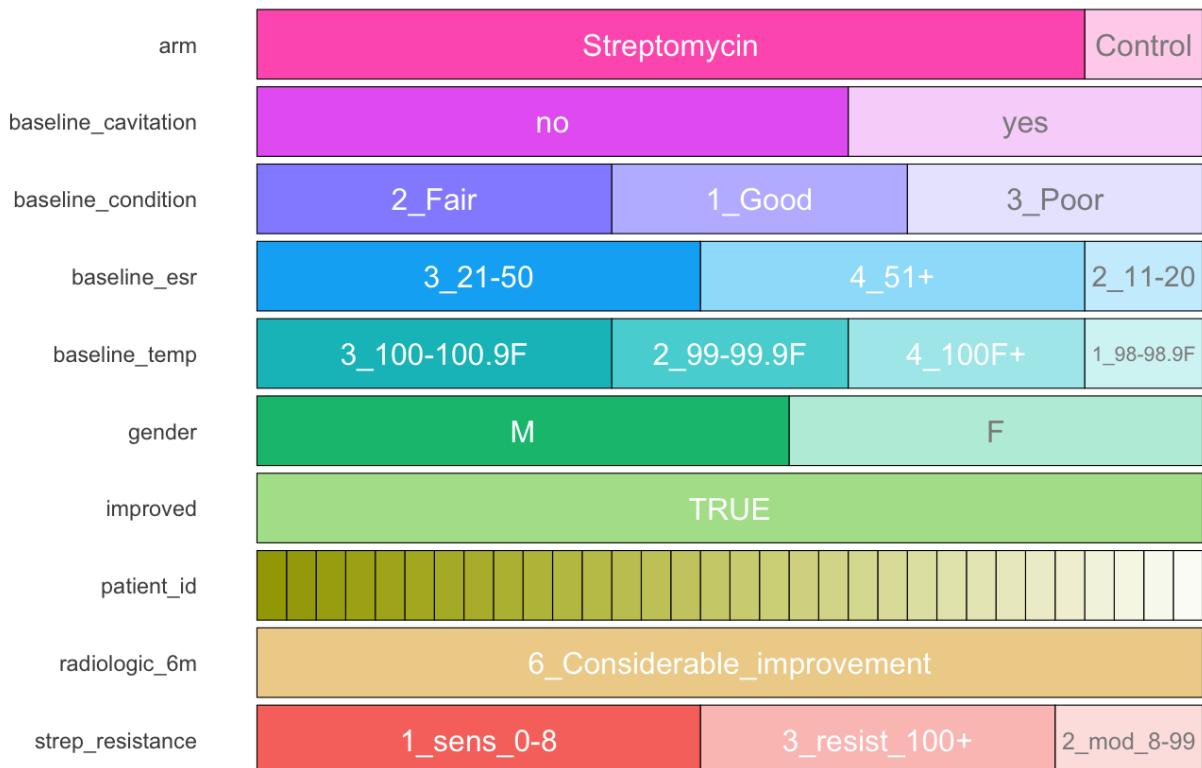
Une autre application courante des boucles est la génération de multiples graphiques pour différents groupes au sein d'un jeu de données. Nous utiliserons le jeu de données `strep_tb` du package `medicaldata` pour illustrer cela. Notre objectif est de créer des graphiques d'inspection des catégories pour chaque groupe d'amélioration radiologique à 6 mois.

Commençons par créer un graphique pour l'un des groupes. Nous utiliserons `inspectdf:::inspect_cat()` pour générer un graphique d'inspection des catégories :

```
cat_plot <-
  medicaldata::strep_tb %>%
  filter(radiologic_6m == "6_Con siderable_improvement") %>%
  inspectdf::inspect_cat() %>%
  inspectdf::show_plot()
cat_plot
```

Frequency of categorical levels in df::Piped data

Gray segments are missing values



Ce graphique nous offre un moyen rapide de visualiser la distribution des catégories dans notre jeu de données.

Maintenant, nous voulons créer des graphiques similaires pour chaque groupe d'amélioration radiologique dans le jeu de données. D'abord, identifions tous les groupes uniques en utilisant la fonction unique :

```
niveaux_radiologiques_6m <- medicaldata::strep_tb$radiologic_6m %>% unique()
niveaux_radiologiques_6m
```

```
## [1] 6_Con siderable_improvement 5_Moderate_improvement 4_No_change
## [4] 3_Moderate_deterioration 2_Con siderable_deterioration 1_Death
## 6 Levels: 6_Con siderable_improvement 5_Moderate_improvement 4_No_change
... 1_Death
```

Ensuite, initialisons un objet liste vide où nous stockerons les graphiques.

```
liste_graphiques_cat <- vector("list", length(niveaux_radiologiques_6m))  
liste_graphiques_cat
```

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

Nous allons également définir les noms des éléments de la liste pour les groupes d'amélioration radiologique. C'est une étape facultative, mais cela facilite l'accès aux graphiques spécifiques plus tard.

```
names(liste_graphiques_cat) <- niveaux_radiologiques_6m  
liste_graphiques_cat
```

```
## $`6_Considerable_improvement`  
## NULL  
##  
## $`5_Moderate_improvement`  
## NULL  
##  
## $`4_No_change`  
## NULL  
##  
## $`3_Moderate_deterioration`  
## NULL  
##  
## $`2_Considerable_deterioration`  
## NULL  
##  
## $`1_Death`  
## NULL
```

Finalement, nous utiliserons une boucle pour générer un graphique pour chaque groupe et le stocker dans la liste :

```

for (niveau in niveaux_radiologiques_6m) {

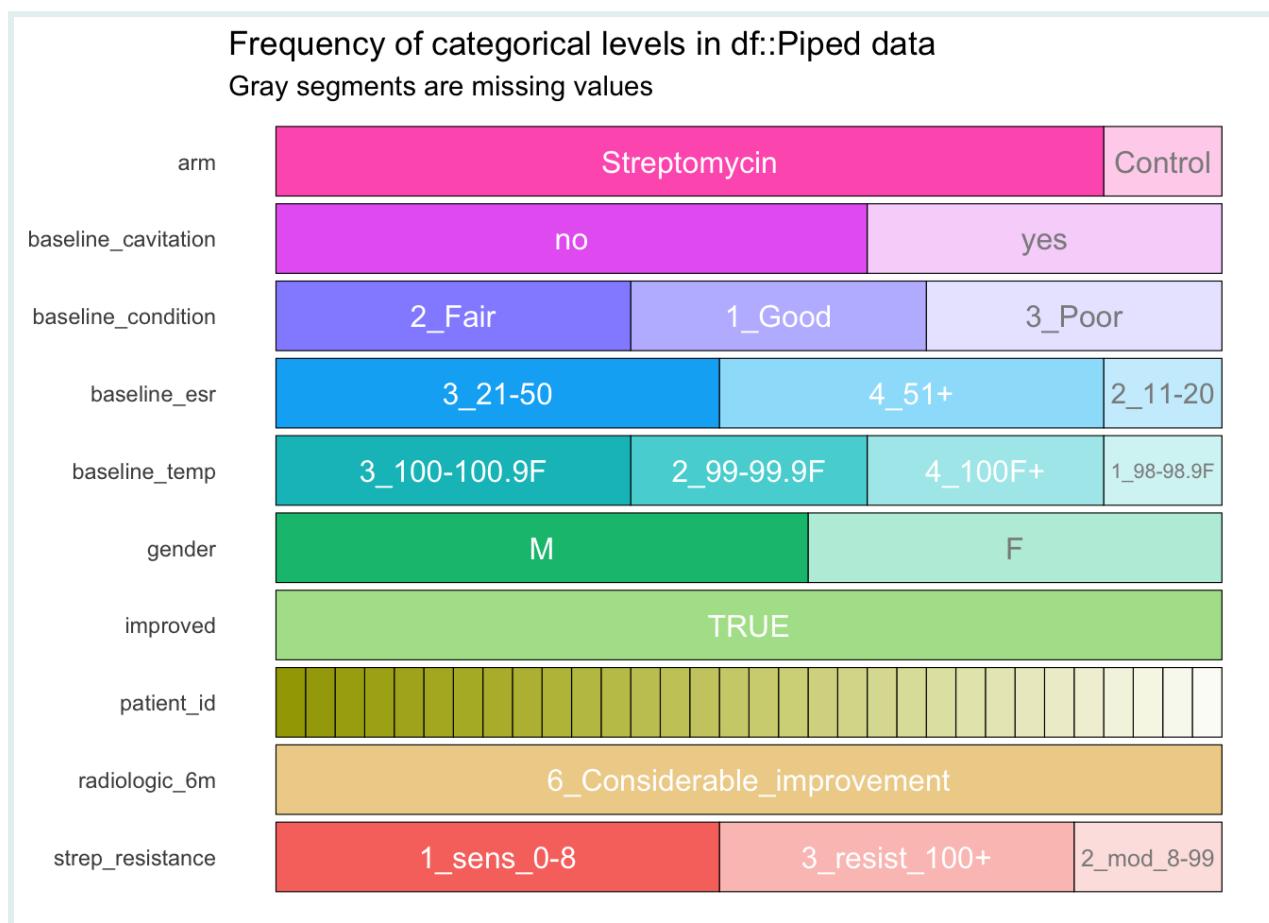
  # Générer le graphique pour chaque niveau
  cat_plot <-
    medicaldata::strep_tb %>%
    filter(radiologic_6m == niveau) %>%
    inspectdf::inspect_cat() %>%
    inspectdf::show_plot()

  # Ajouter à la liste
  liste_graphiques_cat[[niveau]] <- cat_plot
}

```

Pour accéder à un graphique spécifique, nous pouvons utiliser la syntaxe à double crochet :

```
liste_graphiques_cat[["6_Considerable_improvement"]]
```



Notez que dans ce cas, les éléments de la liste sont *nommés*, plutôt que simplement numérotés. Cela est dû au fait que nous avons utilisé la variable niveau comme index dans la boucle.

Pour afficher tous les graphiques à la fois, nous appelons simplement la liste entière.

liste_graphiques_cat

```

## $`6_Con siderable_improvement`  
  

##  
## $`5_Moderate_improvement`  
  

##  
## $`4_No_change`  
  

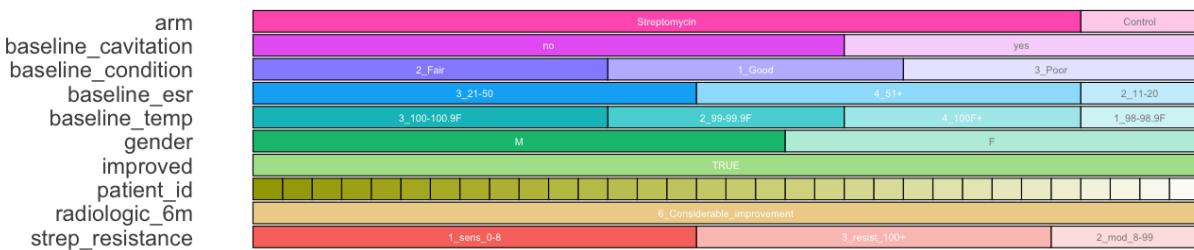
##  
## $`3_Moderate_deterioration`  
  

##  
## $`2_Con siderable_deterioration`  
  

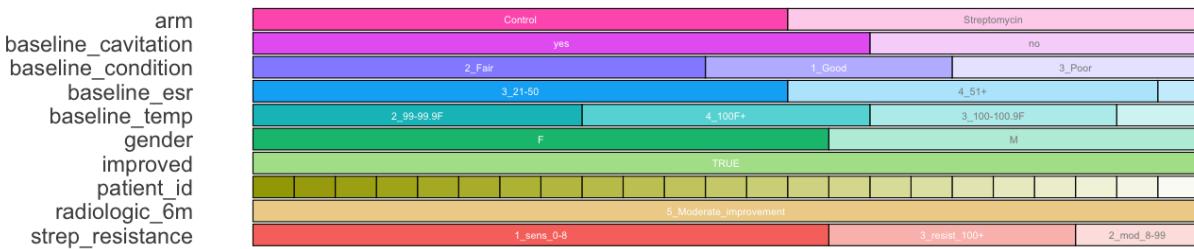
##  
## $`1_Death`  


```

Frequency of categorical levels in df::Piped data Gray segments are missing values

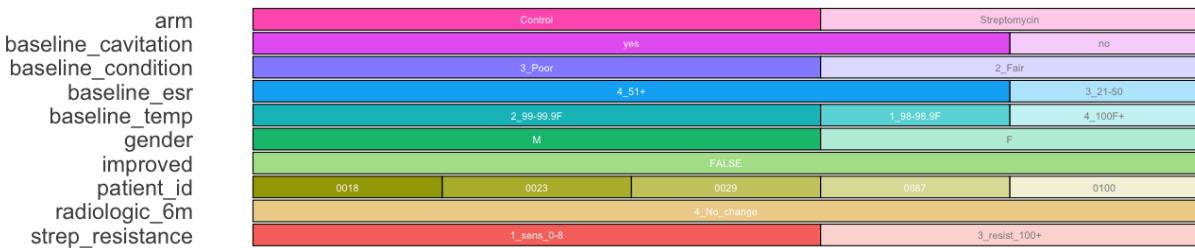


Frequency of categorical levels in df::Piped data Gray segments are missing values



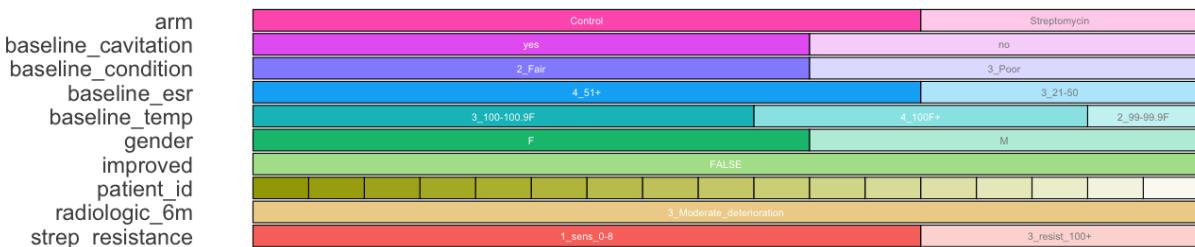
Frequency of categorical levels in df::Piped data

Gray segments are missing values



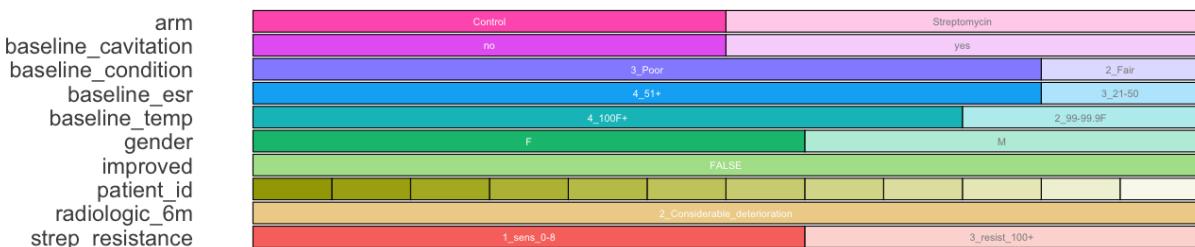
Frequency of categorical levels in df::Piped data

Gray segments are missing values



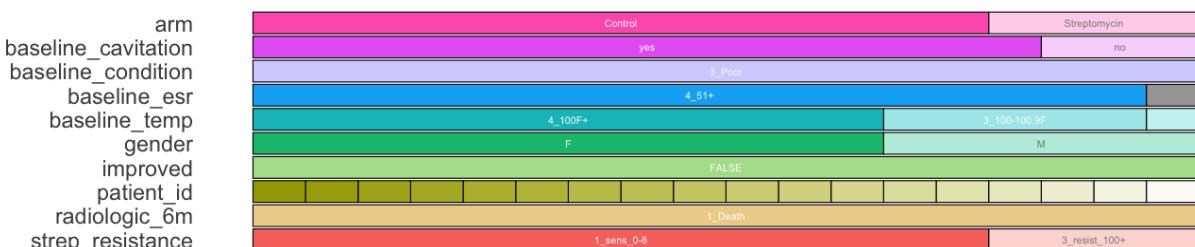
Frequency of categorical levels in df::Piped data

Gray segments are missing values



Frequency of categorical levels in df::Piped data

Gray segments are missing values



Visualisation des cas de tuberculose

Dans cet exercice, vous utiliserez des données de l'OMS du package `tidyverse` pour créer des graphiques en lignes montrant le nombre de nouveaux cas de tuberculose chez les enfants au fil des années dans les pays d'Amérique du Sud.

D'abord, nous préparerons les données :

```

cas_tb_enfants <- tidyverse::who2 %>%
  transmute(country, year,
            tb_cases_children = sp_m_014 + sp_f_014 + sn_m_014 + sn_f_014) %>%
  filter(country %in% c("Brazil", "Colombia", "Argentina",
                        "Uruguay", "Chile", "Guyana")) %>%
  filter(year >= 2006)

cas_tb_enfants

```

```

## # A tibble: 48 × 3
##   country     year tb_cases_children
##   <chr>      <dbl>           <dbl>
## 1 Argentina  2006             880
## 2 Argentina  2007            1162
## 3 Argentina  2008             961
## 4 Argentina  2009             593
## 5 Argentina  2010             491
## 6 Argentina  2011             867
## 7 Argentina  2012             745
## 8 Argentina  2013             NA
## 9 Brazil     2006            2254
## 10 Brazil    2007            2237
## # ... i 38 more rows

```

Maintenant, remplissez les blancs dans le modèle ci-dessous pour créer un graphique en lignes pour chaque pays en utilisant une boucle `for` :

```

# Obtenez la liste des pays. Indice : Utilisez unique() sur la colonne pays
pays <- _____

# Créez une liste pour stocker les graphiques. Indice : Initialisez une liste vide
graphiques_cas_tb_enfants <- vector("list", _____)
names(graphiques_cas_tb_enfants) <- pays # Définissez les noms des éléments de la liste

# Boucle à travers les pays
for (pays in _____) {

  # Filtrer les données pour chaque pays
  cas_tb_enfants_filtrés <- _____

  # Créer le graphique
  graphique_cas_tb_enfants <- _____

  # Ajouter à la liste. Indice : Utilisez des doubles crochets
  graphiques_cas_tb_enfants[[pays]] <- graphique_cas_tb_enfants
}

graphiques_cas_tb_enfants

```

```
## Error: <text>:2:10: unexpected input
## 1: # Obtenez la liste des pays. Indice : Utilisez unique() sur la colonne
##   pays
## 2: pays <- __
##
```

Conclusion !

Dans cette leçon, nous nous sommes plongés dans les boucles `for` en R, démontrant leur utilité pour des tâches simples jusqu'à des analyses de données complexes impliquant plusieurs jeux de données et la génération de graphiques. Malgré la préférence de R pour les opérations vectorisées, les boucles `for` sont indispensables dans certains scénarios. Espérons que cette leçon vous a équipé des compétences nécessaires pour mettre en œuvre avec confiance les boucles `for` dans divers contextes de traitement de données.

Corrigé

Boucle Basique pour Convertir des Heures en Minutes

```
heures <- c(3, 4, 5) # Vecteur d'heures

for (heure in heures) {
  minutes <- heure * 60
  print(minutes)
}
```

```
## [1] 180
## [1] 240
## [1] 300
```

Boucle Indexée pour Convertir des Heures en Minutes

```
heures <- c(3, 4, 5) # Vecteur d'heures

for (i in 1:length(heures)) {
  minutes <- heures[i] * 60
  print(minutes)
}
```

```
## [1] 180  
## [1] 240  
## [1] 300
```

Boucle de Calcul de l'IMC

```
poids <- c(30, 32, 35) # Poids en kg  
tailles <- c(1.2, 1.3, 1.4) # Tailles en mètres  
  
for(i in 1:length(poids)) {  
  imc <- poids[i] / (tailles[i] ^ 2)  
  
  print(paste("Poids :", poids[i],  
             "Taille :", tailles[i],  
             "IMC :", imc))  
}
```

```
## [1] "Poids : 30 Taille : 1.2 IMC : 20.833333333333"  
## [1] "Poids : 32 Taille : 1.3 IMC : 18.9349112426035"  
## [1] "Poids : 35 Taille : 1.4 IMC : 17.8571428571429"
```

Conversion de la Taille de cm en m

```
taille_cm <- c(180, 170, 190, 160, 150) # Tailles en cm  
  
taille_m <- vector("numeric", length = length(taille_cm))  
  
for (i in 1:length(taille_cm)) {  
  taille_m[i] <- taille_cm[i] / 100  
}  
taille_m
```

```
## [1] 1.8 1.7 1.9 1.6 1.5
```

Classification de la Température

```
temp_corporelle <- c(35, 36.5, 37, 38, 39.5) # Températures corporelles en  
Celsius  
vect_classif <- vector("character", length = length(temp_corps)) # vecteur de  
caractères
```

```
## Error in vector("character", length = length(temp_corps)): object  
'temp_corps' not found
```

```

for (i in 1:length(temp_corps)) {
  # Ajoutez votre logique if-else ici
  if (temp_corporelle[i] < 36.5) {
    sortie <- "Hypothermie"
  } else if (temp_corporelle[i] <= 37.5) {
    sortie <- "Normal"
  } else {
    sortie <- "Fièvre"
  }

  # Instruction d'impression finale
  vect_classif[i] <- paste(temp_corporelle[i], "°C est", sortie)
}

```

Error in eval(expr, envir, enclos): object 'temp_corps' not found

vect_classif

Error in eval(expr, envir, enclos): object 'vect_classif' not found

Propriétés des Fichiers

```

# En supposant que le chemin et la structure des fichiers sont corrects
fichiers_csv <- list.files(path = "data/new_hiv_infections_gho",
                           pattern = "\\.csv$", full.names = TRUE)

liste_data_frames <- vector("list", length = length(fichiers_csv))

for (i in 1:length(fichiers_csv)) {

  chemin <- fichiers_csv[i]
  nom_pays <- tools::file_path_sans_ext(basename(chemin))

  taille <- file.size(chemin)
  date <- file.mtime(chemin)

  ligne_data_hiv <- tibble(pays = nom_pays, taille = taille, date = date)

  liste_data_frames[[i]] <- ligne_data_hiv
}

info_fichiers_hiv_final <- bind_rows(liste_data_frames)
info_fichiers_hiv_final

```

```

## # A tibble: 9 × 3
##   pays           taille date
##   <chr>          <dbl> <dttm>
## 1 Bangladesh     6042  2023-12-11 17:34:28

```

```

## 2 Brazil 5946 2023-12-11 17:34:28
## 3 Democratic_Republic_of_the_Con... 8028 2023-12-11 17:34:28
## 4 Egypt 6181 2023-12-11 17:34:28
## 5 Ethiopia 5754 2023-12-11 17:34:28
## 6 Indonesia 6621 2023-12-11 17:34:28
## 7 Iran_(Islamic_Republic_of) 8037 2023-12-11 17:34:28
## 8 Philippines 6321 2023-12-11 17:34:28
## 9 Viet_Nam 6230 2023-12-11 17:34:28

```

Filtrage des Données en Boucle

```

fichiers_csv <- list.files(path = "data/new_hiv_infections_gho",
                           pattern = "*.csv", full.names = TRUE)

for (fichier in fichiers_csv) {
  donnees_vih <- read_csv(fichier)

  donnees_vih_filtrees <- donnees_vih %>% filter(Sex == "Female")

  nom_fichier_sortie <- paste0(here(), "/outputs/", "Female_",
                                basename(fichier))

  write_csv(donnees_vih_filtrees, nom_fichier_sortie)
}

```

```

## Rows: 89 Columns: 5
## — Column specification

## Delimiter: ","
## chr (4): Continent, Country, Sex, NewHIVCases
## dbl (1): Year
##
## 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.

## Error: Cannot open file for writing:
## *
'/Users/kendavidn/Dropbox/tgc_github_projects/fdar_staging/FDAR_FR_loops/outputs/Female_E

```

Visualisation des Cas de Tuberculose

```
# En supposant que tb_child_cases est un dataframe avec les colonnes nécessaires
pays <- unique(cas_tb_enfants$country)

# Crée une liste pour stocker les graphiques
graphiques_cas_tb_enfants <- vector("list", length(pays))
names(graphiques_cas_tb_enfants) <- pays

# Boucle à travers les pays
for (nom_pays in pays) {

  # Filtre les données pour chaque pays
  cas_tb_enfants_filtres <- filter(cas_tb_enfants, country == nom_pays)

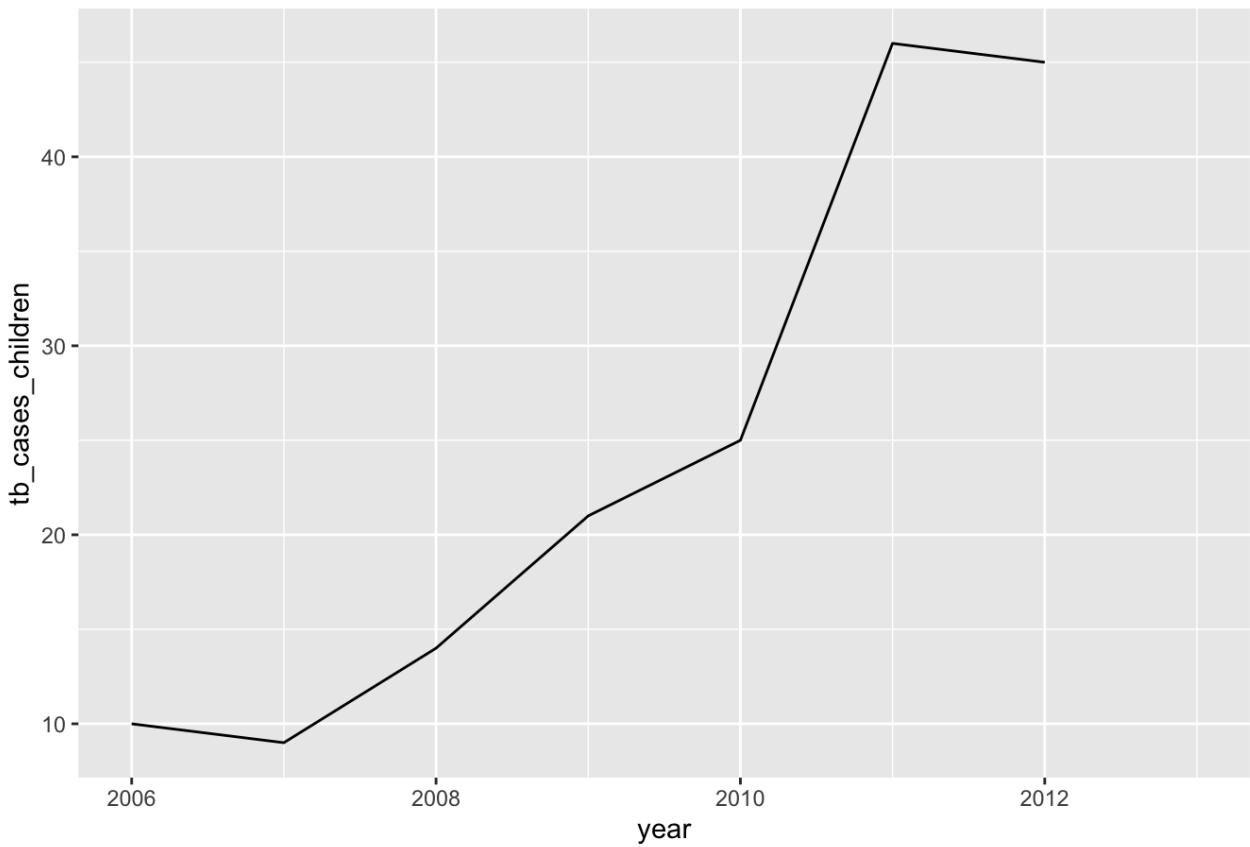
  # Crée le graphique
  graphique_cas_tb_enfant <- ggplot(cas_tb_enfants_filtres, aes(x = year, y =
tb_cases_children)) +
    geom_line() +
    ggtitle(paste("Cas de TB chez les Enfants -", nom_pays))

  # Ajoute au liste
  graphiques_cas_tb_enfants[[nom_pays]] <- graphique_cas_tb_enfant
}

graphiques_cas_tb_enfants[["Uruguay"]]
```

Warning: Removed 1 row containing missing values (`geom_line()`).

Cas de TB chez les Enfants - Uruguay



Contributeurs

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



SABINA RODRIGUEZ VELÁSQUEZ

Project Manager and Scientific Collaborator, The GRAPH Network
Infectiously enthusiastic about microbes and Global Health



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement



GUY WAFFEU

R Instructor and Public Health Physician
Committed to improving the quality of data analysis

Références

Du matériel dans cette leçon a été adapté des sources suivantes :

- Barnier, Julien. "Introduction à R et au tidyverse." <https://juba.github.io/tidyverse>
- Wickham, Hadley; Grolemund, Garrett. "R for Data Science." <https://r4ds.had.co.nz/>
- Wickham, Hadley; Grolemund, Garrett. "R for Data Science (2e)." <https://r4ds.hadley.nz/>

Nettoyage de données: analyse exploratoire

Introduction
Objectifs d'apprentissage
Packages
Jeu de données
Visualisation des données manquantes avec <code>visdat::vis_dat()</code>
Génération de statistiques sommaires avec <code>skimr::skim()</code>
Visualisation des statistiques sommaires avec le package <code>inspectdf</code>
Explorer les variables catégorielles avec <code>gtsummary::tbl_summary()</code>
Création de rapports de données avec <code>DataExplorer::create_report()</code>
En Résumé
Answer Key

Introduction

Le nettoyage des données est le processus qui consiste à transformer des données brutes et “désordonnées” en des données fiables pouvant être analysées correctement. Cela implique d’identifier les points de données **inexact**s, **incomplet**s ou **improbable**s et de résoudre les incohérences ou les erreurs dans les données, ainsi que de renommer les noms de variables pour les rendre plus clairs et simples à manipuler.

Les tâches de nettoyage de données peuvent souvent être fastidieuses. Une blague courante chez les analystes de données dit : “80% de la science des données consiste à nettoyer les données et les 20% restants à se plaindre du nettoyage des données”. Mais le nettoyage des données est une étape essentielle du processus d’analyse des données. Un peu de nettoyage avant de commencer votre analyse contribuera à améliorer la qualité de vos analyses et la facilité avec laquelle ces analyses peuvent être effectuées. Et une gamme de packages et de fonctions dans R peuvent énormément simplifier le processus de nettoyage des données.

Dans cette leçon, nous allons commencer à examiner un processus de nettoyage de données typique dans R. Les étapes de nettoyage couvertes ici ne correspondent probablement pas exactement à ce dont vous aurez besoin pour vos propres données, mais elles constituent certainement un bon point de départ.

Commençons !

Objectifs d'apprentissage

- Vous pouvez énumérer les opérations typiques impliquées dans le processus de nettoyage des données

- Vous pouvez diagnostiquer des problèmes dans vos jeux de données à l'aide des fonctions suivantes :

- `visdat::vis_dat()`
- `skimr::skim()`
- `inspectdf::inspect_cat()`
- `inspectdf::inspect_num()`
- `gtsummary::tbl_summary()`
- `DataExplorer::create_report()`

Packages

Les packages chargés ci-dessous seront nécessaires pour cette leçon :

```
if(!require("pacman")) install.packages("pacman")
pacman::p_load(visdat,
                skimr,
                inspectdf,
                gtsummary,
                DataExplorer,
                tidyverse)
```

Jeu de données

Le jeu de données principal que nous utiliserons dans cette leçon provient d'une étude menée dans trois centres de santé de Zambezia, au Mozambique. L'étude a examiné les facteurs individuels associés au temps jusqu'à la non-adhérance aux services de soins et de traitement du VIH. Pour les besoins de cette leçon, nous ne regarderons qu'un sous-ensemble modifié de l'ensemble de données complet.

L'ensemble de données complet peut être trouvé [ici](#), et l'article peut être consulté [ici](#).

Jetons un coup d'œil à ce jeu de données :

```
non_adherence <- read_csv(here("data/non_adherence_VF.csv"))
```

```
non_adherence
```

```

## # A tibble: 5 × 15
##   id_patient District `Unite de sante` Sexe Age_35
##       <dbl>      <dbl>          <dbl> <chr> <chr>
## 1     10037      1             1 Homme plus de 35 ans
## 2     10537      1             1 F     plus de 35 ans
## 3     5489       2             3 F     Moins de 35 ans
## 4     5523       2             3 Homme Moins de 35 ans
## 5     4942       2             3 F     plus de 35 ans
## # i 10 more variables: `Age a l'initiation du ARV` <dbl>,
## #   Education <chr>, Occupation <chr>, ...

```

La première étape du nettoyage des données consistera à explorer cet ensemble de données afin d'identifier les problèmes potentiels qui nécessitent un nettoyage. Cette étape préliminaire est parfois appelée "analyse exploratoire des données" ou AED.

Regardons quelques fonctions simples dans R qui vous aideront à identifier les erreurs et incohérences de données possibles.

Visualisation des données manquantes avec `visdat::vis_dat()`

La fonction `vis_dat()` du package `visdat` est un excellent moyen de visualiser rapidement les types de données et les valeurs manquantes dans un jeu de données. Elle crée un graphique qui montre une vue "zoomée" de votre datafram : chaque ligne du datafram est représentée par une seule ligne sur le graphique.

Essayons d'abord avec un petit jeu de données fictif pour comprendre son fonctionnement. Copiez le code suivant pour créer un datafram de 8 patients et leurs informations de diagnostic et de guérison du COVID-19. Comme vous pouvez le voir ci-dessous, certaines informations sont manquantes pour certains patients, représentées par NA.

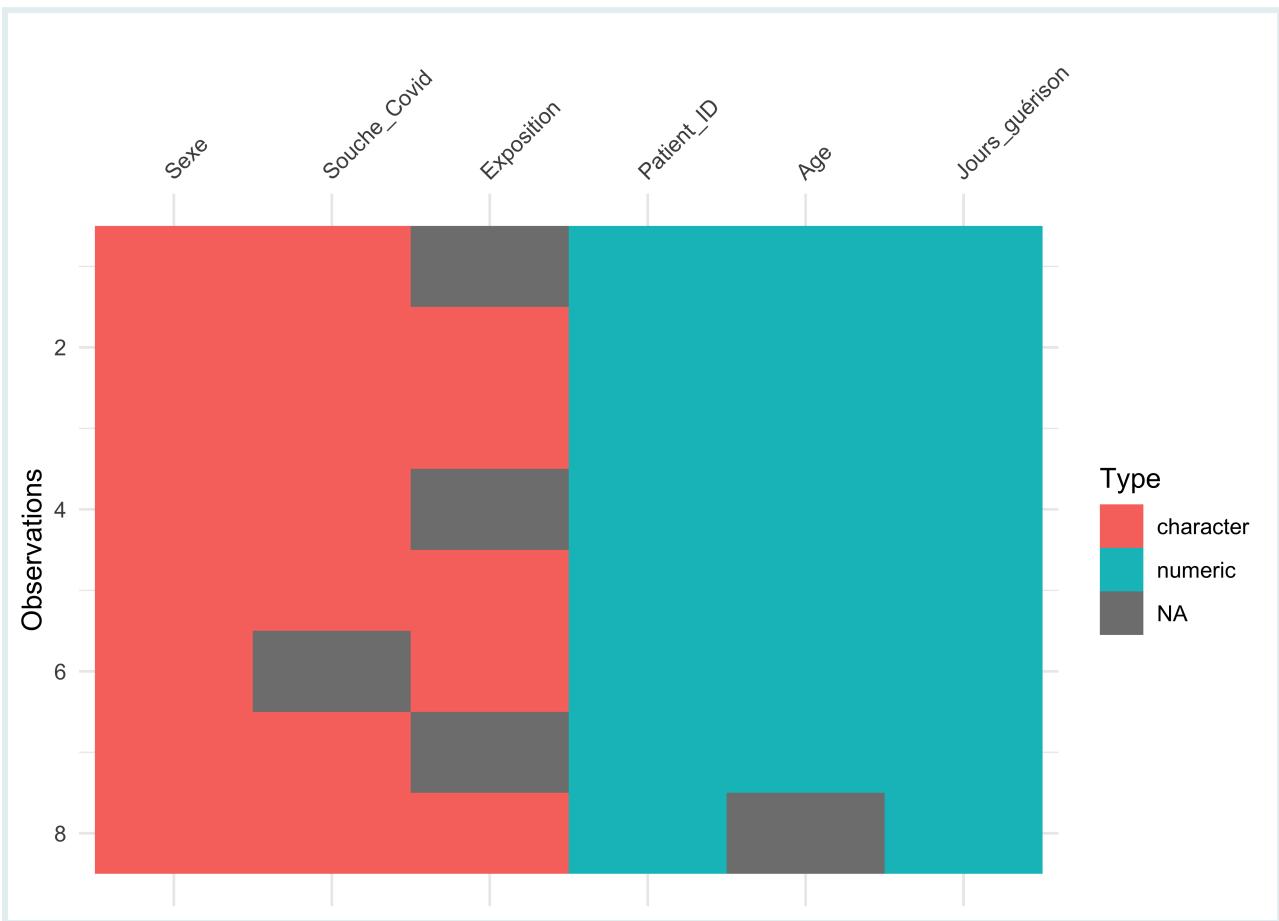
```

covid_pat <- tribble(
  ~Patient_ID, ~Age, ~Sexe, ~Souche_Covid, ~Exposition, ~Jours_guérison,
  1,        25,  "Homme", "Alpha",      NA,           10,
  2,        32,  "Femme", "Delta",    "Hôpital",      15,
  3,        45,  "Homme", "Beta",     "Voyage",       7,
  4,        19,  "Femme", "Omicron",    NA,           21,
  5,        38,  "Homme", "Alpha",    "Inconnu",      14,
  6,        55,  "Femme",  NA,        "Communauté",  19,
  7,        28,  "Femme", "Omicron",    NA,           8,
  8,        NA,  "Femme", "Omicron", "Voyage",      26
)
covid_pat

```

Maintenant, utilisons la fonction `vis_dat()` sur notre datafram pour obtenir une représentation visuelle des types de données et des valeurs manquantes.

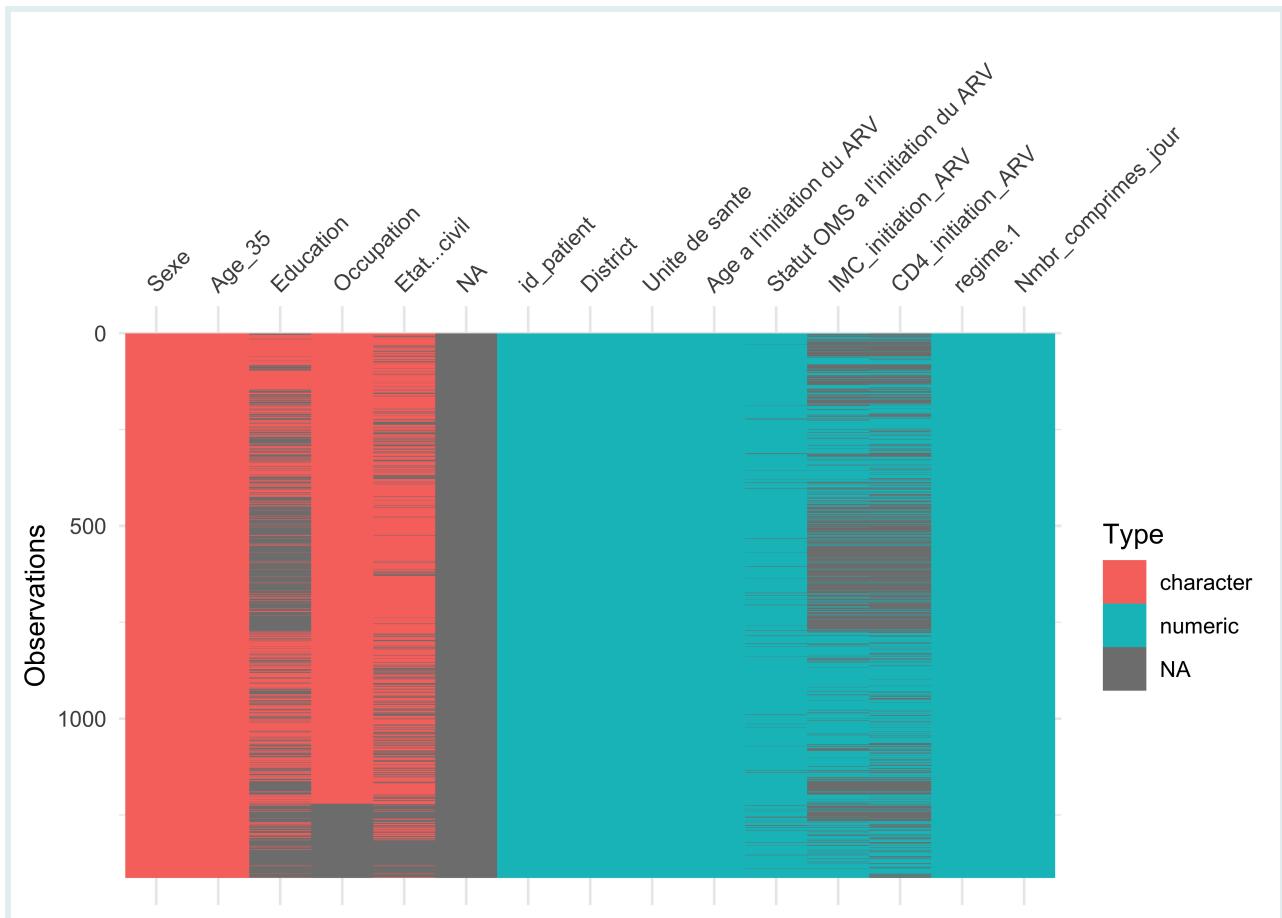
```
vis_dat(covid_pat)
```



C'est parfait ! Chaque ligne de notre dataframme est représentée par une seule ligne dans le graphique et différentes couleurs sont utilisées pour illustrer les variables caractères (rose) et numériques (bleu), ainsi que les valeurs manquantes (gris). À partir de ce graphique, nous pouvons voir que plusieurs patients de notre jeu de données ont des données manquantes pour la variable Exposition.

Regardons maintenant notre jeu de données du monde réel, qui est beaucoup plus grand et désordonné. Les grands jeux de données réels peuvent présenter des structures complexes qui sont difficiles à repérer sans visualisation, donc des fonctions comme `vis_dat()` peuvent être particulièrement utile. Essayons-le maintenant !

```
vis_dat(non_adherence)
```



Génial ! À partir de cette vue, nous pouvons déjà voir certains problèmes :

- Il semble y avoir une colonne complètement vide (la colonne NA qui est entièrement grise)
- Plusieurs variables ont beaucoup de valeurs manquantes (comme Education, IMC_initiation_ARV et CD4_initiation_ARV)
- Les noms de certaines variables sont peu clairs/non nettoyés (par exemple, Age a l'initiation du ARV et Statut OMS a l'initiation du ARV ont des espaces dans leurs noms et Etat...civil et regimen.1 ont des caractères spéciaux .)

Dans la prochaine leçon, nous essayerons de remédier à ces problèmes pendant le processus de nettoyage des données. Mais pour l'instant, l'objectif est que nous comprenions les fonctions utilisées pour les identifier. Maintenant que nous savons visualiser les données manquantes avec `vis_dat()`, jetons un coup d'œil à un autre package et fonction qui peut nous aider à générer des statistiques sommaires de nos variables !

Q : Repérer les problèmes de données avec `vis_dat()`

L'ensemble de données suivant a été adapté d'une étude qui a examiné les opportunités manquées de dépistage du VIH chez les patients se présentant pour la première fois pour

des soins contre le VIH dans un hôpital universitaire suisse. L'ensemble de données complet peut être trouvé [ici](#).

```
opp_manquees <- read_csv(here("data/opportunites_manquees.csv"))
```

```
## Rows: 201 Columns: 16
## — Column specification

## Delimiter: ","
## chr (11): sexe, age, origine, acquis, chronic, aigu, present_tardive,
## raison_diagnostic, testlocal, cat_consul...
## dbl (4): cd4, categorie_cd4, nmbr_consult, opp_manquees
## lgl (1): NaN.
##
## 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.
```

Utilisez la fonction `vis_dat()` pour obtenir une représentation visuelle des données. Quels problèmes potentiels pouvez-vous repérer ?

Génération de statistiques sommaires avec `skimr::skim()`

La fonction `skim()` du package `skimr` fournit un résumé de chaque colonne (par classe/type) dans la console. Essayons-la d'abord sur nos données fictives `covid_pat` pour comprendre son fonctionnement. Tout d'abord, pour rappel, voici notre dataframe `covid_pat` :

```
covid_pat
```

Parfait, essayons maintenant la fonction `skim()` !

```
skimr::skim(covid_pat)
```

Table 1: Data summary

Name	covid_pat
Number of rows	8
Number of columns	6
<hr/>	
Column type frequency:	
character	3
numeric	3
<hr/>	
Group variables	None

Variable type: character

skim_variable	n_missing	complete_rate	min	max	empty	n_unique	whitespace
Sexe	0	1.00	5	5	0	2	0
Souche_Covid	1	0.88	4	7	0	4	0
Exposition	3	0.62	6	10	0	4	0

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
Patient_ID	0	1.00	4.50	2.45	1	2.75	4.5	6.25	8	
Age	1	0.88	34.57	12.39	19	26.50	32.0	41.50	55	
Jours_guérison	0	1.00	15.00	6.68	7	9.50	14.5	19.50	26	

Incroyable ! Comme nous pouvons le voir, cette fonction fournit :

- Un aperçu des lignes et des colonnes du dataframe
- Le type de données pour chaque variable
- n_missing, le nombre de valeurs manquantes pour chaque variable
- complete_rate, le taux d'exhaustivité pour chaque variable
- Un ensemble de statistiques sommaires : la moyenne, l'écart-type, et les quartiles pour les variables numériques ; et la fréquence et les proportions pour les variables catégorielles
- Des histogrammes spark pour les variables numériques

Maintenant, nous pouvons voir pourquoi cela est si utile avec de grands dataframes !

Revenons à notre jeu de données non_adherence et exécutons la fonction skim() dessus.

```
non_adherence
```

```
skimr::skim(non_adherence)
```

Table 2: Data summary

Name	non_adherence
Number of rows	1413
Number of columns	15
<hr/>	
Column type frequency:	
character	5
logical	1
numeric	9

Group variables	None
-----------------	------

Variable type: character

skim_variable	n_missing	complete_rate	min	max	empty	n_unique	whitespace
Sexe	0	1.00	1	5	0	2	0
Age_35	0	1.00	14	15	0	2	0
Education	776	0.45	4	10	0	5	0
Occupation	193	0.86	4	25	0	49	0
Etat...civil	412	0.71	7	12	0	4	0

Variable type: logical

skim_variable	n_missing	complete_rate	mean	count
NA	1413		0	NaN :

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100
skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100
id_patient	0	1.00	7364.75	3312.95	147.0	4705.00	7355.0	10098.0	24300.0
District	0	1.00	1.35	0.48	1.0	1.00	1.0	2.0	2.0
Unite de sante	0	1.00	1.87	0.90	1.0	1.00	2.0	3.0	3.0
Age a l'initiation du ARV	0	1.00	32.07	9.51	15.1	25.20	30.4	36.8	42.0
Statut OMS a l'initiation du ARV	45	0.97	1.80	0.98	1.0	1.00	1.0	3.0	5.0
IMC_initiation_ARV	588	0.58	20.37	3.54	9.0	18.12	20.1	22.2	55.0
CD4_initiation_ARV	674	0.52	398.19	243.64	3.0	233.00	343.0	538.0	1400.0
regime.1	0	1.00	4.64	1.54	1.0	3.00	6.0	6.0	6.0
Nmbr_comprimes_jour	0	1.00	1.51	0.59	1.0	1.00	1.0	2.0	2.0

À partir de cette sortie de données, nous pouvons identifier certains problèmes potentiels :

- Nous pouvons confirmer que la colonne NA est bien complètement vide : elle a un complete_rate de 0
- La distribution de Age a l'initiation du ARV est asymétrique

Q : Générer des statistiques sommaires avec `skim()`

Utilisez `skim()` pour obtenir un aperçu détaillé de l'ensemble de données `opp_manquees`.

Super ! Maintenant nous savons comment générer un bref rapport de statistiques sommaires pour nos variables. Dans la prochaine section, nous découvrirons quelques fonctions utiles du package `inspectdf` qui nous permettent de visualiser différentes statistiques sommaires.

Visualisation des statistiques sommaires avec le package `inspectdf`

Bien que la fonction `skimr::skim()` vous donne des résumés de variables dans la console, parfois il est préférable d'avoir un résumé de variables sous une forme graphique plus riche. Pour cela, les fonctions `inspectdf::inspect_cat()` et `inspectdf::inspect_num()` peuvent être utilisées.

Si vous exécutez `inspect_cat()` sur un jeu de données, vous obtenez un résumé des variables catégorielles dans un tableau (les informations importantes sont cachées dans

la colonne `levels`). Essayons-le d'abord sur le jeu de données `covid_pat`. Pour rappel, voici notre jeu de données :

```
covid_pat
```

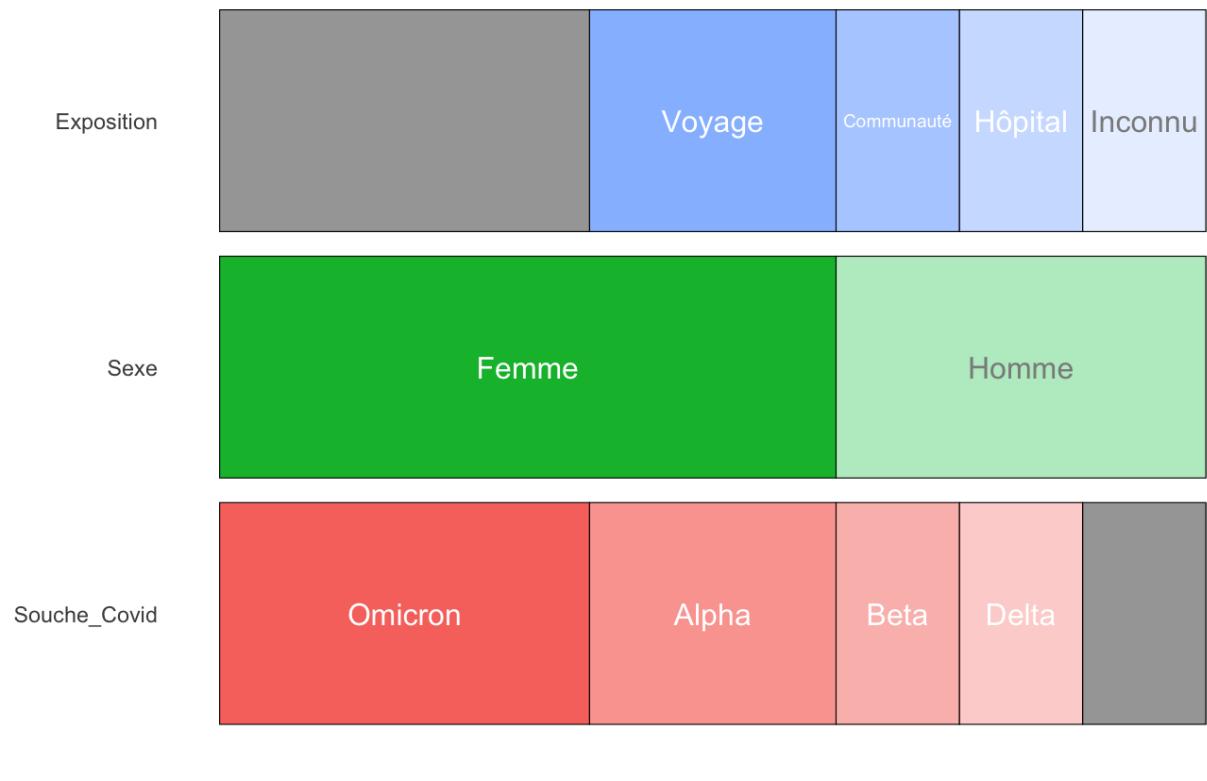
```
inspect_cat(covid_pat)
```

La magie se produit lorsque vous utilisez la fonction `show_plot()` sur le résultat de `inspect_cat()` :

```
inspect_cat(covid_pat) %>%  
  show_plot()
```

Frequency of categorical levels in df::covid_pat

Gray segments are missing values



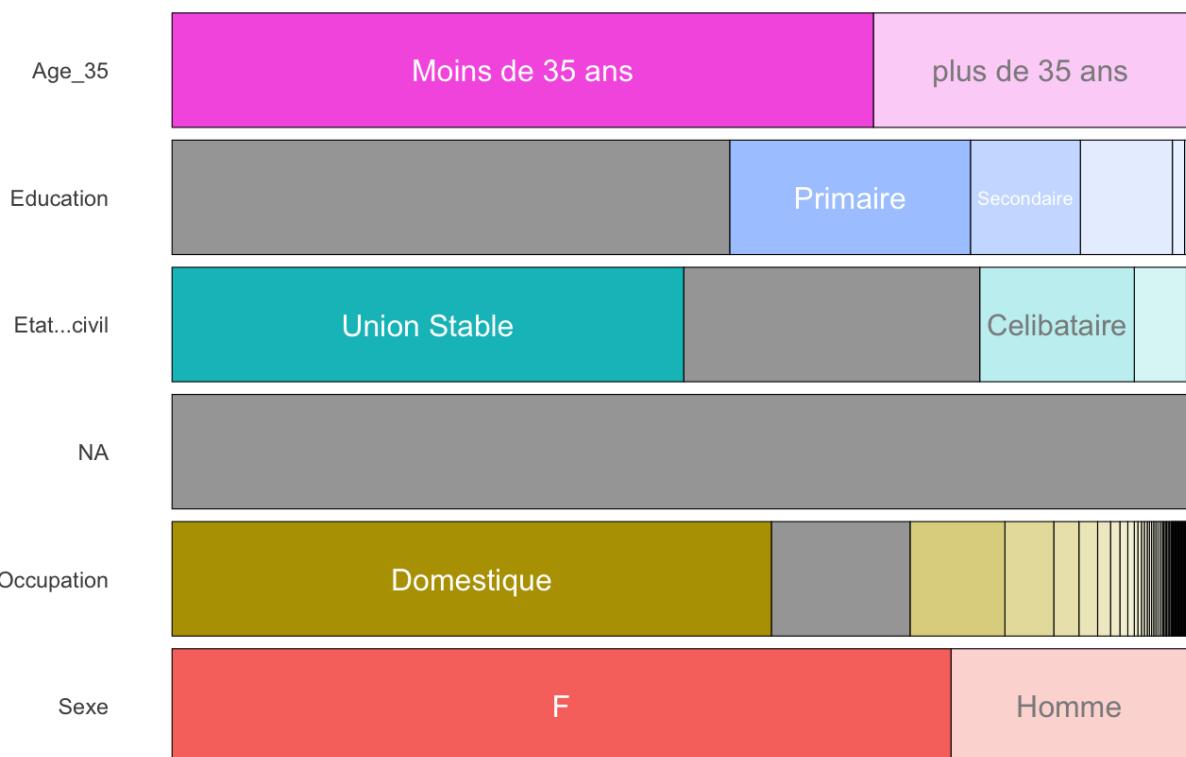
C'est parfait ! Vous obtenez une belle figure récapitulative montrant la distribution des variables catégorielles ! Les niveaux de variable sont également étiquetés, s'il y a suffisamment d'espace pour afficher une étiquette.

Maintenant, essayons-le sur notre jeu de données `non_adherence` :

```
inspect_cat(non_adherence) %>%  
  show_plot()
```

Frequency of categorical levels in df::non_adherence

Gray segments are missing values



À partir de là, vous pouvez observer certains problèmes avec quelques variables catégorielles :

- Pour la variable Age_35, on a deux niveaux. Le niveau Moins de 35 ans commence par une majuscule, tandis que plus de 35 ans commence par une minuscule. Ca pourrait être un bon idée de standardiser cela.
- La variable sexe a les niveaux F et Homme. Cela pourrait également valoir la peine d'être standardisé.
- Comme nous l'avons vu précédemment, NA est complètement vide.

Q : Repérer les problèmes de données avec inspect_cat()

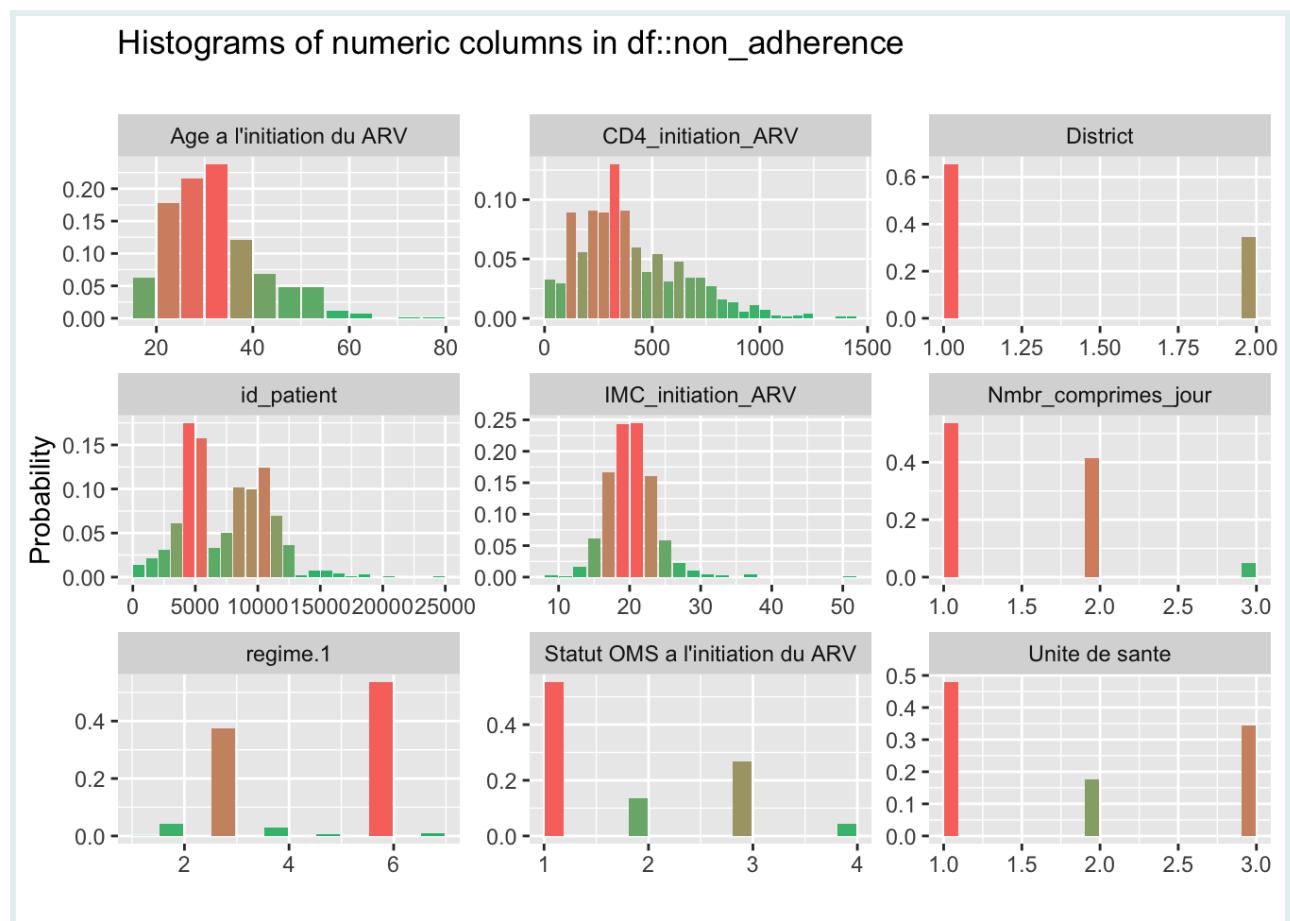
Complétez le code suivant pour obtenir un résumé visuel des variables catégorielles dans le jeu de données opp_manquees.

```
inspect__() %>%
```

Combien de problèmes de données potentiels pouvez-vous repérer ?

De même, vous pouvez obtenir un graphique de résumé pour les variables numériques du jeu de données avec `inspect_num()`. Essayons cela sur notre jeu de données `non_adherence` :

```
inspect_num(non_adherence) %>%
  show_plot()
```



À partir de ce graphique généré, vous remarqueriez que de nombreuses variables qui devraient être des variables factor sont codées comme numériques. En fait, les seules vraies variables numériques sont `Age a l'initiation du ARV`, `IMC_initiation_ARV`, `CD4_initiation_ARV` et `Nmbr_comprimes_jour`. Nous corrigerons ces problèmes dans la prochaine leçon lorsque nous passerons au nettoyage des données. En attendant, regardons de plus près une autre fonction qui s'avère particulièrement utile pour les variables catégorielles !

Q : Types de variables avec `inspect_num()`

Utilisez `inspect_num` pour créer des histogrammes de vos variables numériques dans le jeu de données `opp_manquees`. Les types de variables numériques sont-ils corrects ?

Explorer les variables catégorielles avec gtsummary::tbl_summary()

Bien que la fonction `inspect_cat()` soit utile pour un aperçu graphique des variables catégorielles, elle ne fournit pas d'informations sur les fréquences et les pourcentages pour les différents niveaux. Pour cela, `tbl_summary()` du package `gtsummary` est particulièrement utile ! La sortie de données étant particulièrement longue, nous allons regarder la forme tibble et montrer une photo de la partie importante pour notre jeu de données. Vous pouvez explorer les données complètes en codant chez vous.

Essayons-le sur notre jeu de données `non_adherence` :

```
gtsummary::tbl_summary(non_adherence) %>%  
  as_tibble()
```

```
## # A tibble: 103 × 2  
##   `**Characteristic**` `**N = 1,413**`  
##   <chr>                 <chr>  
## 1 id_patient            7,355 (4,705, 10,098)  
## 2 District              <NA>  
## 3 1                      925 (65%)  
## 4 2                      488 (35%)  
## 5 Unite de sante         <NA>  
## 6 1                      678 (48%)  
## 7 2                      247 (17%)  
## 8 3                      488 (35%)  
## 9 Sexe                  <NA>  
## 10 F                     1,084 (77%)  
## # i 93 more rows
```

Super ! Comme nous pouvons le voir, cela nous fournit un résumé des fréquences et des pourcentages pour les variables catégorielles et la médiane et l'IQR pour les variables numériques.

Ci-dessous vous pouvez voir une photo d'une partie de la sortie où nous pouvons remarquer des problèmes supplémentaires dans nos données qui n'étaient pas clairs avec la fonction `inspect_cat()`. Certaines valeurs de notre variable `Occupation` sont en majuscules, tandis que d'autres sont entièrement en minuscules.

Police	3 (0.2%)
Pompier	1 (<0.1%)
professeur	11 (0.9%)
Professeur	35 (2.9%)
Proprietaire de magasin	1 (<0.1%)
Receptionniste	1 (<0.1%)
Retraite	2 (0.2%)

Cela signifie que R ne les reconnaît pas comme étant la même valeur, ce qui poserait problème lors de l'analyse. Nous corrigerons ces erreurs dans la prochaine leçon, pour l'instant passons à notre dernière fonction pour l'analyse exploratoire des données !



Q : Repérer les problèmes de données avec `tbl_summary()`

Utilisez `tbl_summary()` pour produire un résumé de votre jeu de données `opp_manquees`. Pouvez-vous identifier des problèmes de données supplémentaires ?

Création de rapports de données avec `DataExplorer::create_report()`

Enfin, la fonction `create_report()` du package `DataExplorer` crée un profil complet d'un dataframe : un fichier HTML avec des statistiques de base et des visualisations de distribution.

Utilisons cette fonction sur notre jeu de données `non_adherence`. Notez que cela peut prendre un certain temps. Si cela prend trop de temps, vous pouvez l'exécuter sur un sous-ensemble du jeu de données plutôt que sur l'ensemble du jeu de données.

```
create_report(non_adherence)
```

Comme vous pouvez le voir, le rapport est assez complet. Nous ne passerons pas en revue toutes les sorties de ce rapport de données car de nombreuses graphiques sont les mêmes que celles que nous avons vues avec les fonctions précédentes ! Cependant, certain nouveautés incluent :

- Un graphique QQ pour évaluer la normalité des variables numériques
- Une analyse de corrélation (lorsqu'il y a suffisamment de lignes complètes)
- Une analyse en composantes principales (lorsqu'il y a suffisamment de lignes complètes)

N'hésitez pas à explorer la [documentation du package](#) par vous-même.

Q : Rapport de données avec create_report()

Créez un rapport de données pour vos données `opp_manquees` en utilisant la fonction `create_report()` !

En Résumé

En nous familiarisant avec les données, nous avons pu identifier certains problèmes potentiels qu'il faudra résoudre avant d'utiliser les données dans une analyse.

Et comme vous l'avez vu d'autres développeurs R ont fait le travail difficile pour créer d'incroyables packages pour analyser rapidement les jeux de données et identifier les problèmes.

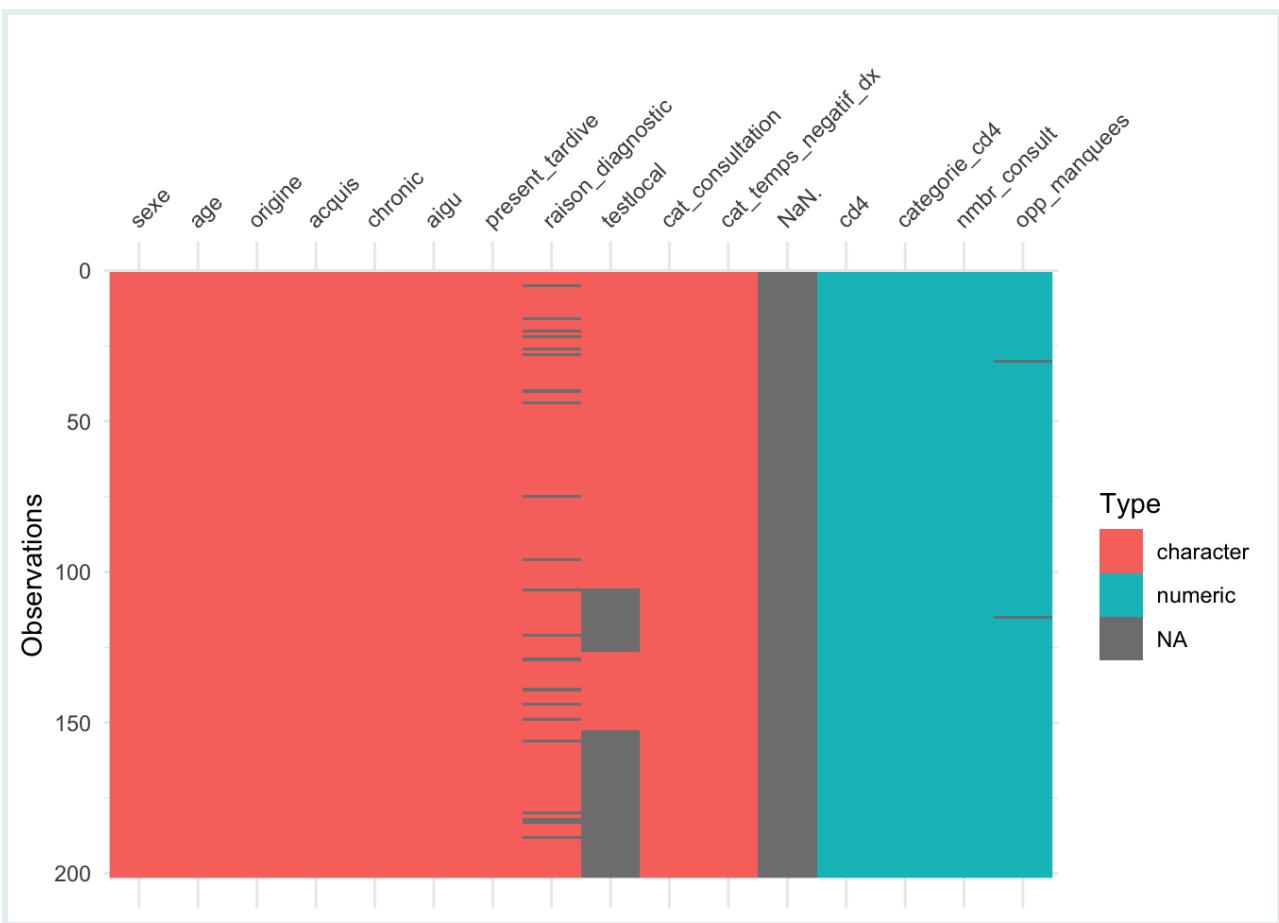
À partir de la prochaine leçon, nous allons aborder ces problèmes identifiés un par un, en commençant par le problème de noms de variables incohérents et désordonnés.

On se retrouve à la prochaine leçon !

Answer Key

Q : Repérer les problèmes de données avec vis_dat()

```
vis_dat(opp_manquees)
```



- La colonne NaN est complètement vide

Q : Générer des statistiques sommaires avec `skim()`

`skim(opp_manquees)`

Table 3: Data summary

Name	opp_manquees
Number of rows	201
Number of columns	16
<hr/>	
Column type frequency:	
character	11
logical	1
numeric	4
<hr/>	
Group variables	None

Variable type: character

skim_variable	n_missing	complete_rate	min	max	empty	n_unique	whitespace
sexe	0	1.00	1	5	0	3	0
age	0	1.00	3	5	0	3	0
origine	0	1.00	6	21	0	3	0
acquis	0	1.00	3	19	0	4	0
chronic	0	1.00	3	3	0	2	0
aigu	0	1.00	3	8	0	2	0
present_tardive	0	1.00	3	3	0	2	0
raison_diagnostic	21	0.90	9	50	0	8	0
testlocal	70	0.65	20	26	0	3	0
cat_consultation	0	1.00	28	30	0	3	0
cat_temps_negatif_dx	0	1.00	29	39	0	3	0

Variable type: logical

skim_variable	n_missing	complete_rate	mean	count
NaN.	201		0	NaN :

Variable type: numeric

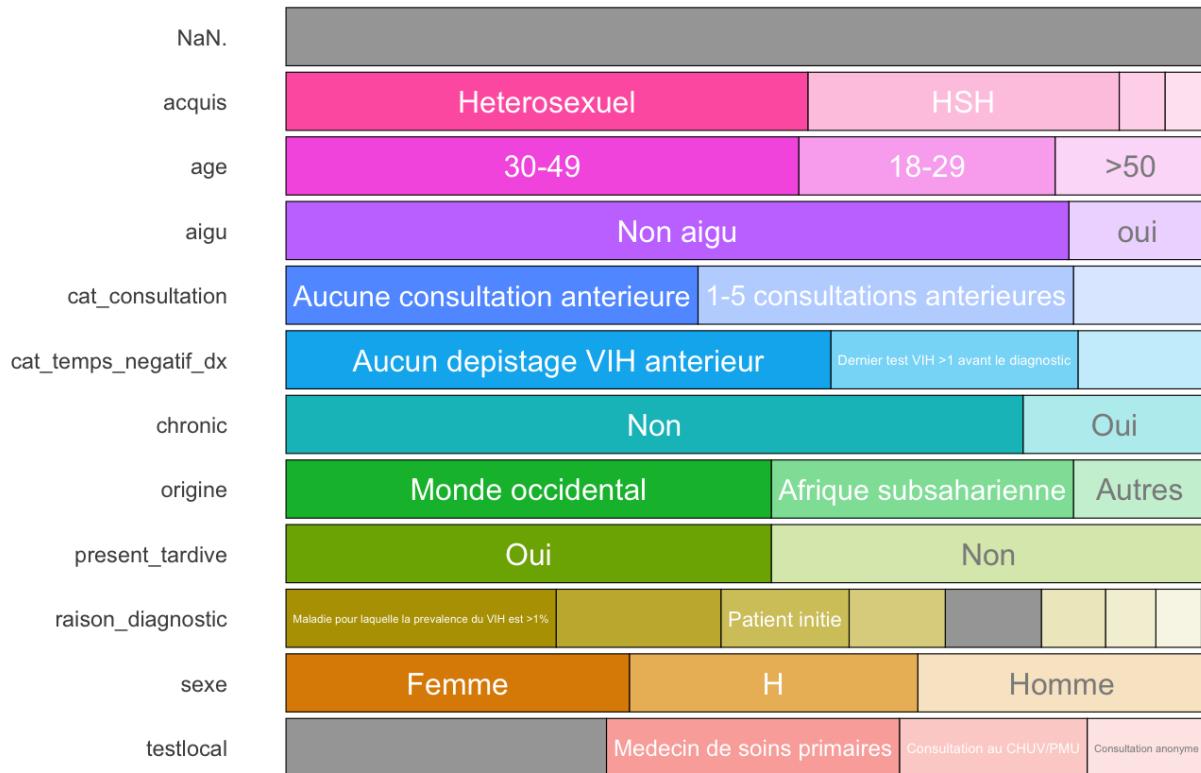
skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
cd4	0	1.00	342.13	261.68	2	147	293	452	1261	
categorie_cd4	0	1.00	2.27	1.12	1	1	2	3	4	
nmbr_consult	0	1.00	1.98	2.90	0	0	1	3	21	
opp_manquees	2	0.99	1.83	3.01	0	0	0	3	17	

Q : Repérer les problèmes de données avec `inspect_cat()`

```
inspect_cat(opp_manquees) %>%
  show_plot()
```

Frequency of categorical levels in df::opp_manquees

Gray segments are missing values

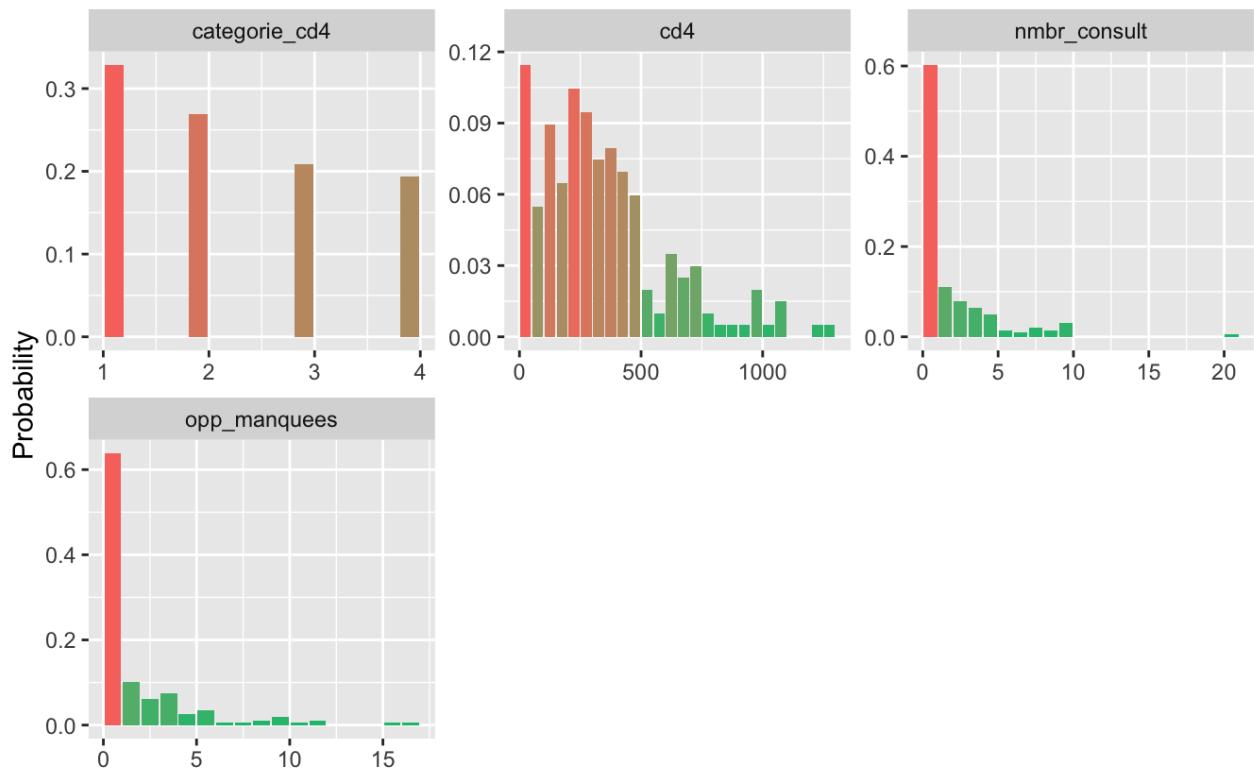


- La variable acute a 2 niveaux : Not acute et yes. Cela devrait être normalisé.
- La variable sex a 3 niveaux : Female, Male et M. Le M devrait être changé en Male.

Q : Types de variables avec inspect_num()

```
inspect_num(opp_manquees) %>%
  show_plot()
```

Histograms of numeric columns in df::opp_manquees



- La variable cd4category devrait être une variable de type facteur.

Q : Repérer les problèmes de données avec `tbl_summary()`

```
tbl_summary(opp_manquees)
```

Characteristic	N = 201¹
sexe	
Femme	75 (37%)
H	63 (31%)
Homme	63 (31%)
age	
>50	33 (16%)
18-29	56 (28%)
30-49	112 (56%)
origine	
Afrique subsaharienne	66 (33%)
Autres	29 (14%)
Monde occidental	106 (53%)
acquis	
Autre	10 (5.0%)
Heterosexuel	114 (57%)
HSH	68 (34%)
Usage de drogues IV	9 (4.5%)
chronic	
Non	161 (80%)
Oui	40 (20%)
cd4	293 (147, 452)
categorie_cd4	
1	66 (33%)
2	54 (27%)

	Characteristic	N = 201¹
4		39 (19%)
aigu		
Non aigu		171 (85%)
oui		30 (15%)
present_tardive		
Non		95 (47%)
Oui		106 (53%)
raison_diagnostic		
Grossesse		14 (7.8%)
Introduction d'un traitement immunosupresseur		1 (0.6%)
Maladie definissant le sida		21 (12%)
Maladie pour laquelle la prevalence du VIH est >1%		59 (33%)
Patient initie		28 (16%)
Risque epidem.		10 (5.6%)
Risque epidemiologique		11 (6.1%)
Suspicion d'infection aigue par le VIH		36 (20%)
Unknown		21
testlocal		
Consultation anonyme		26 (20%)
Consultation au CHUV/PMU		41 (31%)
Medecin de soins primaires		64 (49%)
Unknown		70
nmbr_consult		1.00 (0.00, 3.00)
cat_consultation		
>5 consultations anterieures		29 (14%)
1-5 consultations anterieures		82 (41%)
Aucune consultation anterieure		90 (45%)
cat_temps_negatif_dx		
Aucun depistage VIH anterieur		119 (59%)
Depistage du VIH l'annee precedente		28 (14%)
Dernier test VIH >1 avant le diagnostic		54 (27%)
opp_manquees		0.00 (0.00, 3.00)
Unknown		2
NaN.		0 (NA%)
Unknown		201

¹ n (%); Median (IQR)

Q : Rapport de données avec `create_report()`

```
DataExplorer::create_report(opp_manques)
```

References

Une partie du matériel de cette leçon a été adaptée des sources suivantes :

- Batra, Neale, et al. The Epidemiologist R Handbook. 2021. *Cleaning data and core functions*. <https://epirhandbook.com/en/cleaning-data-and-core-functions.html#cleaning-data-and-core-functions>
- Waring E, Quinn M, McNamara A, Arino de la Rubia E, Zhu H, Ellis S (2022). skimr: Compact and Flexible Summaries of Data. <https://docs.ropensci.org/skimr/> (website), <https://github.com/ropensci/skimr/>.

Contributeurs

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



AMANDA MCKINLEY

R Developer and Instructor, the GRAPH Network



KENE DAVID NWOSU

Data analyst, the GRAPH Network

Passionate about world improvement



LAURE VANCAUWENBERGHE

Data analyst, the GRAPH Network

A firm believer in science for good, striving to ally programming, health and education

Nettoyage de données II: corriger les incohérences

Introduction
Objectifs d'apprentissage
Packages
Jeu de données
Nettoyage des noms de colonnes
Noms de colonnes
Nettoyage automatique
Nettoyage automatique puis manuel des noms de colonnes
Supprimer des colonnes et lignes vides
Suppression des colonnes vides
Suppression des lignes vides
Suppression des lignes en double
janitor::get_dups()
dplyr::distinct()
Transformations
Correction des chaînes de caractères
Nettoyage des types de données
Mettre tout ensemble
En Résumé
Answer Key

Introduction

Dans cette leçon, nous allons explorer l'essentielle compétence du nettoyage des données, la prochaine étape cruciale pour raffiner vos jeux de données en vue d'une analyse robuste. Alors que nous passons de l'analyse exploratoire des données, où nous avons identifié des problèmes potentiels, nous nous concentrerons maintenant sur la résolution systématique de ces problèmes. Commençons !

Objectifs d'apprentissage

- Vous savez comment nettoyer automatiquement et manuellement les noms de colonnes
- Vous pouvez facilement supprimer les colonnes et lignes vides
- Vous êtes capable de supprimer les ligne en double
- Vous savez comment corriger les valeurs en caractère de chaîne
- Vous pouvez changer les types de données

Packages

Les packages ci-dessous seront nécessaires pour cette leçon :

```
# Charger les packages
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse,
                janitor,
                inspectdf,
                skimr,
                dplyr)
```

Jeu de données

Le jeu de données que nous utiliserons pour cette leçon est une version légèrement modifiée du jeu de données utilisé dans la première leçon Nettoyage des données ; ici, nous avons ajouté un peu plus d'erreurs à nettoyer ! Consultez la leçon 1 pour une explication de ce jeu de données.

```
non_adherence <- read_csv(here("data/non_adherence_desordre.csv"))
```

```
non_adherence
```

```
## # A tibble: 5 × 15
##   id_patient District `Unite de sante` Sexe Age_35
##       <dbl>     <dbl>           <dbl> <chr> <chr>
## 1      10037      1             1 Homme plus de 35 ans
## 2      10537      1             1 F     plus de 35 ans
## 3       5489       2             3 F     Moins de 35 ans
## 4       5523       2             3 Homme Moins de 35 ans
## 5       4942       2             3 F     plus de 35 ans
## # i 10 more variables: `Age a l'initiation du ARV` <dbl>,
## # EDUCATION_DU_PATIENT <chr>, ...
```

Nettoyage des noms de colonnes

Noms de colonnes

En règle générale, les noms de colonnes doivent avoir une syntaxe “propre” et standardisée afin que nous puissions facilement travailler avec eux et que notre code soit lisible par d’autres codeurs.

Idéalement, les noms de colonnes :



- devraient être courts
- ne devraient pas contenir d’espaces ou de points (remplacer les espaces et les points par des tirets bas “_”)
- ne devraient pas contenir de caractères inhabituels (&, #, <, >)
- devraient avoir un style similaire

Pour voir nos noms de colonnes, nous pouvons utiliser la fonction `names()` de base R.

```
names(non_adherence)
```

```
## [1] "id_patient"                  "District"  
"Unite de sante"  
## [4] "Sexe"                        "Age_35"  
"Age a l'initiation du ARV"  
## [7] "EDUCATION_DU_PATIENT"        "OCCUPATION_DU_PATIENT"  
"Estat...civil"  
## [10] "Statut OMS a l'initiation du ARV" "IMC_initiation_ARV"  
"CD4_initiation_ARV"  
## [13] "regime.1"                      "Nmbr_comprimes_jour"  
"NA"
```

Ici, nous pouvons voir que :

- certains noms contiennent des espaces
- certains noms contiennent des caractères spéciaux comme ...
- certains noms sont en majuscules alors que d’autres non

Nettoyage automatique

Une fonction pratique pour standardiser les noms de colonnes est `clean_names()` du package `janitor`.

La fonction `clean_names()` :

KEY POINT



- Convertit tous les noms pour qu'ils ne contiennent que des underscores, des chiffres et des lettres.
- Analyse les casse et séparateurs selon un format cohérent. (par défaut `snake_case`)
- Gère les caractères spéciaux (&, #, <, >) ou les caractères accentués.

```
non_adherence %>%  
  clean_names() %>%  
  names()
```

```
## [1] "id_patient"                  "district"  
"unite_de_sante"  
## [4] "sexe"                        "age_35"  
"age_a_linitiation_du_arv"  
## [7] "education_du_patient"        "occupation_du_patient"  
"etat_civil"  
## [10] "statut_oms_a_linitiation_du_arv" "imc_initiation_arv"  
"cd4_initiation_arv"  
## [13] "regime_1"                      "nmbr_comprimes_jour"  
"na"
```

D'après cette sortie, nous pouvons voir que :

- les noms de variables en majuscules ont été convertis en minuscules (par ex. `EDUCATION_DU_PATIENT` est devenu `education_du_patient`)
- les espaces dans les noms de variables ont été convertis en underscores (par ex. `Age a l'initiation du ARV` est devenu `age_a_linitiation_du_arv`)
- les points (.) ont tous été remplacés par des underscores (par ex. `Etat...civil` est devenu `etat_civil`)

** Q: Nettoyage automatique **

L'ensemble de données suivant a été adapté d'une étude qui a utilisé des données rétrospectives pour caractériser les dynamiques temporelles et spatiales des épidémies de fièvre typhoïde à Kasene, Ouganda.

```
typhoide <- read_csv(here("data/typhoid_uganda.csv"))
```

```
## Rows: 215 Columns: 31
## — Column specification

## Delimiter: ","
## chr (18): Householdmembers, Positioninthehousehold,
Watersourcedwithinhousehold, Borehole, River, Tap, Rainwa...
## dbl (11): UniqueKey, CaseorControl, Age, Sex, Levelofeducation,
Below10years, N1119years, N2035years, N3644ye...
## lgl (2): NA, NAN
##
## 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.
```

Utilisez la fonction `clean_names()` de `janitor` pour nettoyer les noms de variables dans le jeu de données `typhoide`.

Nettoyage automatique puis manuel des noms de colonnes

Parfois, le nettoyage automatique seul ne suffit pas pour s'assurer que nos noms de colonnes sont propres et ont un sens pour notre jeu de données. Une combinaison de nettoyage automatique et manuel peut être nécessaire. Nous pouvons commencer par le nettoyage automatique, puis vérifier s'il reste des noms de colonnes étranges qui nécessiteraient un nettoyage manuel.

Voici un exemple combinant le nettoyage automatique et manuel : un nettoyage plus approprié serait de standardiser d'abord les noms de colonnes, racourcir les noms de certaines variables, puis de supprimer la fin `_du_patient` de toutes les colonnes. Créons un nouveau dataframe appelé `non_adh_col_propre`.

```
non_adh_col_propre <- non_adherence %>%
  # standardiser la syntaxe des noms de colonnes
  clean_names() %>%
  # racourcir manuellement les noms de deux variables
  rename("age_initiation_arv" = "age_a_linitiation_du_arv",
         "statut_oms_initiation_arv" =
  "statut_oms_a_linitiation_du_arv") %>%
  # enlever manuellement "_de_patient" de toutes les colonnes
  rename_with(str_replace, pattern = "_du_patient", replacement = "")
```

```
## # A tibble: 5 × 15
##   id_patient district unite_de_sante sexe  age_35
```

```

##      <dbl>    <dbl>      <dbl> <chr> <chr>
## 1    10037     1          1 Homme plus de 35 ans
## 2    10537     1          1 F     plus de 35 ans
## 3     5489     2          3 F     Moins de 35 ans
## 4     5523     2          3 Homme Moins de 35 ans
## 5     4942     2          3 F     plus de 35 ans
## # i 10 more variables: age_initiation_arv <dbl>,
## #   education <chr>, occupation <chr>, etat_civil <chr>, ...

```

Cela semble parfait ! Nous utiliserons ce jeu de données avec les colonnes propres dans la prochaine section.

Q: Nettoyage complet des noms de colonnes

Standardisez les noms de colonnes dans le jeu de données typhoide puis:

- remplacez or_ par _
 - remplacez of par _
 - renommez les variables below10years n1119years n2035years n3644years, n4565years above65years en num_below_10_yrs num_11_19_yrs num_20_35_yrs num_36_44_yrs, num_45_65_yrs num_above_65_yrs
-
-

Supprimer des colonnes et lignes vides

Une ligne/colonne **VIDE** est une ligne/colonne où toutes les valeurs sont NA. Lorsque vous chargez un jeu de données, vous devez toujours vérifier s'il contient des lignes ou colonnes vides et les supprimer. Le but est que **chaque ligne soit un point de données significatif** et que **chaque colonne soit une variable significative**. Commençons par nos colonnes.

Suppression des colonnes vides

Pour identifier les colonnes vides, nous allons utiliser la fonction `inspect_na()` du package `inspectdf` pour identifier les colonnes vides.

```
inspectdf::inspect_na(non_adh_col_propre)
```

D'après la sortie, nous voyons que le pcnt indique 100% de vide (c'est-à-dire valeurs NA) pour notre colonne na : il y a une valeur NA dans chaque ligne.

Pour supprimer les colonnes vides du dataframe, nous utiliserons la fonction `remove_empty()` du package `janitor`. Cette fonction supprime toutes les colonnes d'un dataframe composées entièrement de valeurs NA.

Nous appliquerons la fonction sur le jeu de données `non_adh_clean_names` et supprimerons la colonne vide identifiée précédemment.

```
ncol(non_adh_col_propre)
```

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

```
non_adh_col_propre <- non_adh_col_propre %>%
  remove_empty("cols")
```

```
ncol(non_adh_col_propre)
```

```
## [1] 14
```

Ici, nous pouvons voir que la colonne `na` a été supprimée des données.

Q: Supprimer des colonnes vides

Supprimez les colonnes vides du jeu de données `typhoide`.

Suppression des lignes vides

Bien qu'il soit relativement facile d'identifier les colonnes vides à partir de la sortie `skim()`, ce n'est pas aussi facile pour les lignes vides. Heureusement, la fonction `remove_empty()` fonctionne également s'il y a des lignes vides dans les données. Le seul changement dans la syntaxe est de spécifier `"rows"` au lieu de `"cols"`.

```
nrow(non_adh_col_propre)
```

```
## [1] 1420
```

```
non_adh_col_propre <- non_adh_col_propre %>%
  remove_empty("rows")
```

```
nrow(non_adh_col_propre)
```

```
## [1] 1417
```

Le nombre de lignes est passé de 1420 à 1417, ce qui suggère qu'il y avait des lignes vides dans les données qui ont été supprimées.

Q: Supprimer des colonnes et lignes vides

Supprimez à la fois les lignes et colonnes vides du jeu de données typhoide.

Suppression des lignes en double

Très souvent dans vos jeux de données, il y a des situations où vous avez des valeurs en double, lorsqu'une ligne a exactement les mêmes valeurs qu'une autre ligne. Cela peut se produire lorsque vous combinez des données de sources multiples ou lorsque vous avez reçu plusieurs réponses à un sondage.

Il est donc nécessaire d'identifier et de supprimer toutes les valeurs en double de vos données afin de garantir des résultats précis. Pour cela, nous utiliserons deux fonctions : `janitor::get_dupes()` et `dplyr::distinct()`.

`janitor::get_dupes()`

Une façon simple de vérifier rapidement les lignes qui ont des doublons est la fonction `get_dupes()` du package `janitor`.



VOCAB

La syntaxe est `get_dupes(x)`, où `x` est **un dataframe**.

Essayons-le sur notre dataframe `non_adh_clean_names`.

```
non_adh_col_propre %>%  
  get_dupes()
```

```
## No variable names specified – using all columns.
```

La sortie est composée de 8 lignes : il y a 2 lignes pour chaque paire de doublons. Vous pouvez facilement voir que ce sont des doublons d'après la variable `patient_id` qui identifie de façon unique nos observations.

Q: Supprimer des lignes en double

Identifiez les éléments qui sont des doublons dans le jeu de données typhoide.

La fonction `get_dupes()` est utile pour extraire vos données en double avant de les supprimer. Parfois, si vous avez beaucoup de doublons, cela peut indiquer un problème qui nécessite une enquête plus approfondie, comme un problème de collecte de données

ou de fusion. Une fois que vous avez examiné vos doublons, vous pouvez les supprimer avec la fonction suivante.

dplyr::distinct()

`distinct()` est une fonction du package `dplyr` qui ne conserve que les lignes uniques/distinctes d'un datafram. S'il y a des lignes en double, seule la première ligne est conservée.

Vérifions le nombre de lignes avant et après avoir appliqué la fonction `distinct()`.

```
nrow(non_adh_col_propre)
```

```
## [1] 1417
```

```
non_adh_distinct <- non_adh_col_propre %>%
  distinct()

nrow(non_adh_distinct)
```

```
## [1] 1413
```

Initialement, notre jeu de données avait 1417 lignes. L'application de la fonction `distinct()` réduit les dimensions du jeu de données à 1413 lignes. Cela fait sens, car comme nous l'avons vu précédemment, 4 de nos valeurs étaient des doublons.

Q: Affichage des décomptes

Supprimez les lignes en double du jeu de données `typhoide`. Assurez-vous que seules des lignes uniques restent dans le jeu de données.

Transformations

Correction des chaînes de caractères

Il y a souvent des moments où vous devez corriger certaines incohérences dans les chaînes de caractères qui pourraient interférer avec l'analyse des données, y compris les fautes d'orthographe et les erreurs de casse.

Ces problèmes peuvent être corrigés manuellement dans la source de données brute ou nous pouvons apporter la modification dans le pipeline de nettoyage. Cette dernière option est plus transparente et reproductible pour toute personne cherchant à comprendre ou à répéter votre analyse.

Changer les valeurs de chaînes avec case_match()

Nous pouvons utiliser la fonction `case_match()` dans la fonction `mutate()` pour changer des valeurs spécifiques et pour concilier des valeurs orthographiées différemment.

VOCAB



La syntaxe est `case_match(nom_colonne, ancienne_valeur_colonne ~ nouvelle_valeur_colonne)`

Tout d'abord, regardons la colonne sex :

```
non_adh_distinct %>% count(sexe)
```

```
## # A tibble: 2 × 2
##   sexe     n
##   <chr> <int>
## 1 F       1084
## 2 Homme   329
```

Dans cette variable, nous pouvons voir qu'il y a des incohérences dans la façon dont les niveaux ont été codés. Utilisons la fonction `case_match()` pour que F soit changé en Female.

```
non_adh_distinct %>%
  mutate(sexe = case_match(sexe, "F" ~ "Femme", .default=sexe)) %>%
  count(sexe)
```

```
## # A tibble: 2 × 2
##   sexe     n
##   <chr> <int>
## 1 Femme   1084
## 2 Homme   329
```

Cela semble parfait ! `case_match()` est un moyen facile de corriger les valeurs qui ne correspondent pas dans votre jeu de données. Parfois, nous avons des modèles plus complexes à corriger, comme le recodage conditionnel et la suppression de caractères spéciaux. Pour plus d'informations sur la gestion des chaînes de caractères, consultez la leçon sur les chaînes de caractères !

WATCH OUT



Si vous ne spécifiez pas l'argument `.default=nom_colonne`, toutes les valeurs de cette colonne qui ne correspondent pas à celles que vous changez et que vous mentionnez explicitement dans votre fonction

WATCH OUT

`case_match()` seront renvoyées comme NA. Dans le cas ci-dessus, cela signifie que tous les Males auraient été définis comme manquants.

Q: Corriger les chaînes de caractères

La variable `householdmembers` du jeu de données `typhoide` devrait représenter le nombre de personnes dans un ménage. Affichez les différentes valeurs de la variable.

Transformer en minuscule

Il y a une valeur `01-May` dans la variable `householdmembers` du jeu de données `typhoide`. Recodez cette valeur en `1-5`.

Homogénéiser toutes les chaînes dans l'ensemble du jeu de données

Vous vous souviendrez peut-être qu'avec la sortie `skim` de la première leçon sur le pipeline de nettoyage des données, nous avions des cas où nos caractères de chaînes n'étaient pas cohérents en ce qui concerne la casse. Par exemple, pour notre variable `occupation`, nous avions à la fois `Professor` et `professor`.

Pour résoudre ce problème, nous pouvons transformer toutes nos chaînes en minuscules à l'aide de la fonction `tolower()`. Nous sélectionnons toutes les colonnes de type caractère avec `where(is.character)`

```
non_adh_distinct %>% count(occupation)
```

```
non_adh_distinct %>%
  mutate(across(where(is.character),
               ~ tolower(.x))) %>%
  count(occupation)
```

Comme nous pouvons le constater, nous sommes passés de 51 à 49 niveaux uniques pour la variable `occupation` lorsque nous transformons tout en minuscules !

Transformez toutes les chaînes de caractères du jeu de données `typhoide` en minuscules.

Nettoyage des types de données

Les colonnes contenant des valeurs qui sont des nombres, des facteurs ou des valeurs logiques (TRUE/FALSE) se comportent comme prévu uniquement si elles sont correctement classées. En tant que tel, vous devrez peut-être redéfinir le type ou la classe de votre variable.

KEY POINT

R a 6 types de données de base/classes.

- **character** : chaînes ou caractères individuels, entre guillemets
- **numeric** : n'importe quel nombre réel (comprend les décimales)
- **integer** : tout(s) nombre(s) entier(s)
- **logical** : variables composées de TRUE ou FALSE
- **factor** : variables catégorielles/qualitatives
- **Date/POSIXct** : représente les dates et heures du calendrier

KEY POINT



En plus de ceux répertoriés ci-dessus, il y a aussi **raw** qui ne sera pas abordé dans cette leçon.

Vous vous souvenez peut-être que dans la dernière leçon, notre jeu de données contenait 5 variables de type caractère et 9 variables numériques (ainsi qu'une variable logique NA qui a depuis été supprimée car elle était complètement vide). Jetons un coup d'œil rapide à nos variables à l'aide de la fonction **skim()** de la dernière leçon :

```
skim(non_adh_distinct) %>%
  select(skim_type) %>%
  count(skim_type)
```

Dans la dernière leçon, on a vu que toutes nos variables sont catégorielles, sauf **age_initiation_arv**, **imc_initiation_arv**, **cd4_initiation_arv** et **nmbr_comprises_jour**. Transformons toutes les autres en variables factorielles à l'aide de la fonction **as.factor()** !

```
non_adh_distinct %>%
  mutate(across(!c(age_initiation_arv,
                  imc_initiation_arv,
                  cd4_initiation_arv,
                  nmbr_comprises_jour),
              ~ as.factor(.x))) %>%
  skim() %>%
  select(skim_type) %>%
  count(skim_type)
```

SIDE NOTE



SIDE NOTE

pour indiquer où les variables fournies dans `across()` sont utilisées.

Q: Changer les types de données

Convertissez les variables aux positions 13 à 29 dans le jeu de données `typhoide` en facteurs.

Parfait, c'est exactement ce que nous voulions !

Mettre tout ensemble

Maintenant, appliquons ensemble les transformations de chaînes de caractères et de types de données !

```
non_adherence_final <- non_adh_distinct %>%
  # Recodage des valeurs de sexe
  mutate(sexe = case_match(sexe, "F" ~ "Femme", .default=sexe)) %>%
  # Tout en minuscules
  mutate(across(where(is.character),
               ~ tolower(.x))) %>%
  # Changement des types de variables
  mutate(across(!c(age_initiation_arv,
                  imc_initiation_arv,
                  cd4_initiation_arv,
                  nmbr_comprimes_jour),
               ~ as.factor(.x)))

non_adherence_final
```

En Résumé

Félicitations pour avoir terminé cette leçon en deux parties sur le pipeline de nettoyage des données ! Vous êtes désormais mieux équipé pour relever les complexités des jeux de données réels. N'oubliez pas, le nettoyage des données ne consiste pas seulement à arranger des données en désordre ; il s'agit de garantir la fiabilité et la précision de vos analyses. En maîtrisant des techniques telles que la gestion des noms de colonnes, l'élimination des entrées vides, le traitement des doublons, l'affinage des valeurs de chaînes et la gestion des types de données, vous avez perfectionné vos capacités à transformer des données de santé brutes en une base propre pour des informations significatives !

Answer Key

Q: Nettoyage automatique

```
clean_names(typhoide)
```

Q: Nettoyage complet des noms de colonnes

```
typhoide %>%
  clean_names %>%
  rename("num_below_10_yrs" = "below10years",
         "num_11_19_yrs" = "n1119years",
         "num_20_35_yrs" = "n2035years",
         "num_36_44_yrs" = "n3644years",
         "num_45_65_yrs" = "n4565years",
         "num_above_65_yrs" = "above65years") %>%
  rename_with(str_replace_all, pattern = "or_", replacement = " ") %>%
  rename_with(str_replace_all, pattern = "of", replacement = "_") %>%
  names()
```

```
## [1] "unique_key"                      "case_control"                  "age"
## [4] "sex"                           "level_education"
"householdmembers"
## [7] "num_below_10_yrs"                 "num_11_19_yrs"
"num_20_35_yrs"
## [10] "num_36_44_yrs"                   "num_45_65_yrs"
"num_above_65_yrs"
## [13] "positioninthehousehold"        "watersourcedwithinhousehold"
"borehole"
## [16] "river"                         "tap"
"rainwatertank"
## [19] "unprotectedspring"              "protectedspring"                "pond"
## [22] "shallowwell"                  "stream"
"jerrycan"
## [25] "bucket"                        "county"
"subcounty"
## [28] "parish"                        "village"                       "na"
## [31] "nan"
```

Q: Supprimer des colonnes vides

```
typhoide %>%
  remove_empty("cols")
```

Q: Supprimer des colonnes et lignes vides

```
typhoide %>%  
  remove_empty("cols") %>%  
  remove_empty("rows")
```

Q: Supprimer des lignes en double

```
# Identify duplicates  
get_dupes(typhoide)
```

No variable names specified – using all columns.

```
# Remove duplicates  
typhoide_distinct <- typhoide %>%  
  distinct()  
  
# Ensure all distinct rows left  
get_dupes(typhoide_distinct)
```

No variable names specified – using all columns.

No duplicate combinations found of: UniqueKey, CaseorControl, Age, Sex, Levelofeducation, Householdmembers, Below10years, N1119years, N2035years, ... and 22 other variables

Q: Affichage des décomptes

```
typhoide %>%  
  count(Householdmembers)
```

Q: Corriger les chaînes de caractères

```
typhoide %>%  
  mutate(Householdmembers = case_match(Householdmembers, "01-May" ~ "1-5",  
    .default=Householdmembers)) %>%  
  count(Householdmembers)
```

Q: Transformer en minuscule

```
typhoide %>%
  mutate(across(where(is.character),
    ~ tolower(.x)))
```

Q: Changer les types de données

```
typhoide %>%
  mutate(across(13:29, ~as.factor(.)))
```

Contributors

The following team members contributed to this lesson:



KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement



AMANDA MCKINLEY

R Developer and Instructor, the GRAPH Network
