CASSION · FORMATION

R et le tidyverse pour les données de programme

Le même terrain que le cours Python, mais en R — projets et renv, readr et haven, les verbes dplyr, forcats et tidyr — enseigné sur des exports d'enquêtes et de registres, en nommant les réglages par défaut qui diffèrent de pandas plutôt qu'en les passant sous silence.

Formateur	Pierre Robentz Cassion
Niveau	Débutant
Heures apprenant	20 h
Enseignée en	R
Mise à jour	27 juillet 2026
Secteurs	Santé publique · Nutrition
Normes et méthodologies	Gestion axée sur les résultats (GAR) · Normes OMS de croissance de l'enfant

Consulter et exécuter la formation en ligne sur data-analysis.cassion.dev/fr/cours/r-for-programme-data

Ce que vous saurez faire

— Mettre en place un projet R qui se rouvre un an plus tard sur une autre machine, versions de paquets consignées et aucun chemin absolu dans le code

— Lire des exports CSV, Excel, Stata et SPSS sans perdre un zéro initial, une date ou une étiquette de valeur, et dire ce que chaque lecteur fait d'un code de valeur manquante

— Employer les verbes dplyr délibérément, et expliquer ce que group_by() et summarise() font des clés manquantes, des niveaux inutilisés et du regroupement qui survit à l'appel

— Ordonner les catégories avec forcats pour qu'un tableau et un graphique se lisent comme le rapport l'exige plutôt qu'alphabétiquement

— Restructurer un groupe répété aplati avec pivot_longer(), et envelopper tout le nettoyage dans une fonction qui s'applique telle quelle à l'export du trimestre suivant

Place de cette formation dans le programme

Fondamentaux — Passer d'un export brut de programme à un tableau exploitable, dans celui des deux langages — Python ou R — que votre équipe utilise déjà.

Prérequis

Aucun. Cette formation ne suppose aucune expérience préalable de la programmation.

Sommaire

À propos de cette formation

Ceci est un cours de R, non un cours d'analyse. Il suppose que vous savez déjà ce qu'est un indicateur et pourquoi un dénominateur compte — c'est ce qu'enseigne *Fondamentaux de l'analyse de données* — et consacre son temps au langage : ce que font réellement les verbes du tidyverse, lesquels des réglages par défaut de R vous surprendront, et comment écrire quelque chose qui s'exécute encore quand vous n'êtes plus là.

R uniquement. Son pendant, *Python pour les données de programme*, couvre le même terrain en Python. Ailleurs sur cette plateforme, chaque technique est montrée dans les deux langages ; ici la séparation est le principe, car une équipe hérite le plus souvent d'une base de code plutôt qu'elle ne choisit entre deux.

Si vous avez lu le cours Python, ce que celui-ci apporte de plus utile est de nommer les endroits où le comportement par défaut de R est l'**inverse** de celui de pandas. Une valeur manquante fait renvoyer `NA` à `mean()` au lieu d'être ignorée. Une clé de groupe manquante devient son propre groupe au lieu de disparaître. Un niveau de facteur inutilisé subsiste au lieu d'être écarté. Aucun de ces choix n'est meilleur ou pire ; chacun est un pari différent sur ce que le silence doit signifier, et savoir dans quel pari on se trouve constitue l'essentiel du métier.

Tout s'exécute sur de vrais jeux de données de la plateforme — le registre de dépistage par PB de douze communes de l'Artibonite, un extrait de vaccination au format DHIS2, une enquête ménages EAH — avec les zéros initiaux, les codes sentinelles et les codages incohérents que ces fichiers portent réellement.

Première partie

Un projet qui se rouvre

CHAPITRE 1

Une analyse qui se rouvre un an plus tard

Leçon 1 sur 8 · 80 min

Projets RStudio, here(), renv, et les deux habitudes — setwd() et un espace de travail sauvegardé — qui font qu'une analyse R ne tourne que sur une machine, un seul jour.

1.1 L'échec que cette leçon prévient

On vous demande d'expliquer le chiffre du trimestre dernier. Vous ouvrez l'analyse, l'exécutez, et obtenez un chiffre différent — ou rien du tout, parce que le paquet qui l'avait produit a avancé de deux versions.

Ce n'est pas un défaut de l'analyse. C'est l'environnement, et en R trois habitudes précises en sont la cause.

1.2 Les trois habitudes à abandonner

1.2.1 `setwd()`

```
setwd("/Users/marie/Bureau/nutrition")
```

Cela s'exécute sur exactement un ordinateur. C'est la raison la plus fréquente pour laquelle le script d'un collègue échoue sur votre machine, et le correctif n'est pas de modifier le chemin — c'est de ne plus en avoir.

1.2.2 `rm(list = ls())` en tête de script

Cela ressemble à un départ propre et n'en est pas un. Cette instruction supprime les objets de votre espace de travail et laisse tout le reste : paquets attachés, options, répertoire de travail, tout ce qu'un `.Rprofile` a chargé. Un script qui en a besoin est un script non reproductible, et l'exécuter dans une session neuve est le seul véritable test.

Redémarrez R à la place — Ctrl/Cmd + Maj + F10 sous RStudio. C'est le départ propre que `rm(list = ls())` prétend être.

1.2.3 Un espace de travail sauvegardé

RStudio propose d'enregistrer `.RData` à la fermeture et de le recharger au démarrage. Désactivez les deux :

```
Tools -> Global Options -> General
  Restore .RData into workspace at startup    [ ]
  Save workspace to .RData on exit            Never
```

Un espace de travail rechargé signifie que votre session contient des objets dont vous ne voyez pas le code. L'analyse paraît fonctionner parce que quelque chose de la semaine dernière est encore en mémoire, et elle cesse de fonctionner dès qu'une autre personne l'exécute.

1.3 Le projet

Un projet RStudio est un répertoire contenant un fichier `.Rproj`. Ouvrir le projet fixe le répertoire de travail sur ce répertoire — c'est tout le mécanisme, mais il suffit à rendre chaque chemin de votre code relatif au projet plutôt qu'à votre dossier personnel.

```
muac-analysis/
  muac-analysis.Rproj
  data/
    raw/            l'export tel qu'il est arrive, jamais modifie
    interim/
  outputs/
    tables/
    figures/
  R/
    read_register.R
    indicator_table.R
  renv.lock
  README.md
  .gitignore
```

`data/raw` est en lecture seule. L'export tel qu'il est arrivé est la seule chose que vous ne pouvez pas reproduire ; tout le reste est régénéré en réexécutant le code. Il n'est donc jamais modifié, jamais trié dans Excel « juste pour regarder », et une correction se pose à côté de l'original sous un nouveau nom versionné plutôt que de le remplacer.

1.4 `here()` plutôt qu'un chemin relatif

Un chemin relatif comme `"data/raw/muac.csv"` fonctionne depuis la racine du projet et casse dans un notebook, dans un document R Markdown compilé depuis un sous-répertoire, ou lorsqu'on exécute une ligne à la fois depuis un autre emplacement.

```
library(here)

muac <- readr::read_csv(here("data", "raw", "muac-screening-artibonite-2024.v1.csv"))
```

`here()` trouve la racine du projet — le répertoire contenant le `.Rproj`, ou un fichier `.here`, ou un répertoire `.git` — et construit le chemin à partir de là. Le résultat est le même d'où que le code soit lancé.

```
here()
#> [1] "/home/marie/muac-analysis"
```

Appelez `here()` une fois en tête de script et utilisez-le partout. Un chemin construit par `paste0()` avec un `/` ne fonctionnera pas sous Windows ; `here()` gère le séparateur.

1.5 renv : les versions de paquets, consignées

`here()` règle les chemins. Il ne fait rien pour les paquets, et ce sont les paquets qui font bouger le chiffre.

```
install.packages("renv")

renv::init()       # une fois par projet
```

`renv::init()` donne au projet sa propre bibliothèque et écrit `renv.lock`, un relevé de chaque paquet et de sa version exacte. Ensuite :

```
renv::snapshot()   # apres toute installation ou mise a jour
renv::restore()    # sur une autre machine, ou un an plus tard
```

Versionnez `renv.lock`. C'est la différence entre « installez le tidyverse » et « installez les versions qui ont produit ce tableau ».

1.5.1 Installer là où il n'y a pas d'internet

La partie que la plupart des tutoriels sautent, et celle qui compte sur un déploiement.

Sur une machine connectée, téléchargez les sources une fois :

```
renv::init()
renv::snapshot()

# Remplir un cache local avec tout ce que nomme le fichier de verrouillage.
renv::install()
renv::isolate()
```

Copiez ensuite le répertoire du projet — `renv/library` compris — vers le portable de terrain. `renv::restore()` utilisera ce qui est déjà présent plutôt que de chercher un dépôt qu'il ne peut pas voir.

Pour une machine ayant besoin de paquets non encore installés, le mécanisme général est un dépôt local :

```
# Sur la machine connectee
dir.create("pkgs")
download.packages(c("dplyr", "readr", "haven"), destdir = "pkgs", type = "source")

# Sur la machine hors ligne
install.packages(
  c("dplyr", "readr", "haven"),
  repos = NULL,
  type = "source",
  contriburl = paste0("file://", normalizePath("pkgs"))
)
```

Les paquets sources exigent un compilateur pour tout ce qui contient du C ou du C++, ce qui est fréquent. Si les machines de terrain sont sous Windows, téléchargez plutôt les binaires (`type = "win.binary"`) depuis une machine Windows de même version de R.

1.6 Charger les paquets

```
library(readr)
library(dplyr)
```

Deux choses à ne pas faire :

N'utilisez pas `require()` dans un script. Il renvoie `FALSE` et poursuit quand le paquet est absent : le script échoue donc plus loin, sur une erreur déroutante concernant un objet inexistant. `library()` s'arrête sur place et vous dit quel paquet manque.

N'appelez pas `install.packages()` depuis un script. Un script qui installe des logiciels comme effet de bord est un script que personne ne peut relancer sans risque. L'installation,

c'est `renv::restore()`, une fois, délibérément.

Là où deux paquets exportent le même nom — `dplyr::filter()` et `stats::filter()`, `dplyr::lag()` et `stats::lag()` — dites lequel vous voulez :

```
muac |> dplyr::filter(muac_mm < 125)
```

La forme `::` vaut ses quelques caractères dans un script qui survivra au souvenir que vous avez de ce qui était attaché.

1.7 Vérifier l'environnement avant de lui faire confiance

```
sessionInfo()
```

Pour une analyse dont les chiffres comptent, préférez l'assertion à l'inspection :

```
stopifnot(getRversion() >= "4.2.0")
stopifnot(requireNamespace("dplyr", quietly = TRUE))
```

`|>`, le tube natif employé dans tout ce cours, exige R 4.1 ou plus récent. Si votre équipe est sur un R plus ancien, `%>%` de magrittr fait le même travail et le code de ces leçons fonctionne tel quel avec lui.

1.8 Ce qu'il faut remettre à un collègue

À versionner	À ne pas versionner
`R/`, `.Rproj`	`renv/library/`
`renv.lock`	`data/raw/` — voir ci-dessous
`README.md`	`outputs/`
`.gitignore`	`.Rhistory`, `.RData`

```
.Rproj.user/
.Rhistory
.RData
renv/library/
data/raw/
outputs/
```

Ne versionnez jamais de données brutes de bénéficiaires, identifiées ou pseudonymisées. Git conserve chaque version de chaque fichier pour toujours, et un dépôt interne le lundi peut être partagé le vendredi. Les identifiants de connexion — jeton DHIS2, clé KoboToolbox — n'apparaissent jamais dans un script :

```
token <- Sys.getenv("DHIS2_TOKEN")
stopifnot(nzchar(token))
```

Gardez la valeur dans un fichier `.Renviron` couvert par `.gitignore`, et documentez le nom de la variable — non sa valeur — dans le README.

1.9 Ce qui vient ensuite

Le projet se rouvre et les paquets sont épinglés. La leçon suivante y fait entrer un export : `readr` pour le CSV, `readxl` pour Excel, et la spécification de colonnes qui empêche un code de formation sanitaire de devenir un nombre.

CHAPITRE 2

Lire un export avec readr et readxl

Leçon 2 sur 8 · 85 min

Des spécifications de colonnes plutôt que des devinettes, problems() comme habitude et non comme sauvetage, et les pièges d'Excel — feuilles multiples, en-tête qui n'est pas la ligne un, cellules fusionnées et dates en numéro de série.

2.1 readr devine, et vous dit ce qu'il a deviné

```
library(readr)
library(here)

PATH <- here("data", "raw", "muac-screening-artibonite-2024.v1.csv")
muac <- read_csv(PATH)
```

`read_csv()` affiche la spécification qu'il a inférée. Ce message n'est pas un bruit à masquer — c'est le seul moment où le lecteur vous dit ce qu'il a décidé, et sur ce registre il décide huit choses :

```
child_id       character
commune        character
screening_date date
age_months     numeric
sex            character
muac_mm        numeric
oedema         character
outcome        character
```

L'essentiel est juste. Deux points méritent que vous en repreniez le contrôle.

2.2 La devinette porte sur les 1 000 premières lignes

`guess_max` vaut 1 000 par défaut. Une colonne vide sur les mille premières lignes puis numérique ensuite est devinée comme logique — car `NA` est logique — et chaque valeur ultérieure devient `NA`.

C'est l'équivalent readr de l'avertissement de types mixtes de pandas, et il échoue plus discrètement : vous obtenez une colonne pleine de valeurs manquantes et un tableau `problems()` que vous n'avez pas regardé.

```
# Deliberement : tout lire en texte, regarder, puis convertir.
peek <- read_csv(PATH, col_types = cols(.default = col_character()), n_max = 200)
print(peek)
```

Deux cents lignes suffisent à voir la forme et coûtent peu sur un gros fichier.

2.3 Déclarez les colonnes

```
muac <- read_csv(
  PATH,
  col_types = cols(
    child_id       = col_character(),
    commune        = col_character(),
    screening_date = col_date(format = "%Y-%m-%d"),
    age_months     = col_integer(),
    sex            = col_character(),
    muac_mm        = col_integer(),
    oedema         = col_character(),
    outcome        = col_character()
  ),
  na = c("", "NA", "-99")
)
```

Trois avantages.

Les zéros initiaux survivent. `col_character()` sur un identifiant est toute la leçon : un code de formation sanitaire `007` lu comme un nombre devient `7`, la jointure à la liste des formations échoue exactement pour les codes qui portaient un zéro initial, et la défaillance ressemble à des formations manquantes plutôt qu'à une erreur de type.

La règle se généralise : **si vous n'allez jamais faire d'arithmétique dessus, ce n'est pas un nombre.** Codes de formations, numéros de téléphone, identifiants de grappes, numéros de ménages et codes administratifs sont du texte qui s'écrit avec des chiffres.

Le format de date est énoncé. Un format inféré peut changer d'un fichier à l'autre selon la composition des valeurs. `01/02/2024` est soit le 1er février, soit le 2 janvier, et le fichier ne vous dira pas lequel.

La sentinelle est déclarée. Ce registre code un PB non mesuré par `-99`. Lu sans `na`, c'est un nombre :

```
mean(muac$muac_mm)
#> [1] NA
```

Ce qui nous amène à la différence la plus importante avec pandas.

2.4 `mean()` renvoie NA, et c'est une qualité

pandas ignore les valeurs manquantes par défaut ; R non.

```
mean(muac$muac_mm)
#> [1] NA

mean(muac$muac_mm, na.rm = TRUE)
#> [1] 139.92
```

R est bruyant là où pandas est silencieux. Le `NA` est une question : *savez-vous qu'il y a ici des valeurs manquantes, et avez-vous décidé de ce qu'elles signifient ?* Y répondre par `na.rm = TRUE` est parfaitement légitime — y répondre sans avoir remarqué qu'on vous interrogeait, c'est ainsi qu'un dénominateur change en silence.

`na.rm = TRUE` retire les manquants du numérateur **et** du dénominateur. C'est généralement juste pour une moyenne et généralement faux pour un taux de couverture, où l'enfant non mesuré reste un enfant dépisté. Comptez-les :

```
sum(is.na(muac$muac_mm))
#> [1] 72
```

2.5 `na` s'applique au fichier, non à une colonne

```
muac <- read_csv(PATH, na = c("", "NA", "-99"))
```

readr applique `na` à toutes les colonnes. pandas sait restreindre une sentinelle à une colonne avec `na_values = {"muac_mm": ["-99"]}` ; readr non. Si une autre colonne porte légitimement `-99`, traitez-la ensuite :

```
muac <- read_csv(PATH, na = c("", "NA")) |>
  dplyr::mutate(muac_mm = dplyr::na_if(muac_mm, -99L))
```

`na_if()` est la forme restreinte, à préférer dès que plus d'une colonne est numérique.

2.6 `problems()` est une habitude, non un sauvetage

Quand une valeur ne correspond pas à son type déclaré, readr ne s'arrête pas. Il consigne la ligne, la colonne, ce qu'il attendait et ce qu'il a trouvé.

```
issues <- problems(muac)
nrow(issues)
#> [1] 0
```

Zéro sur ce fichier, et c'est justement l'intérêt de la vérification : **le chiffre n'a de sens que si vous le regardez à chaque fois.** Une exécution qui a discrètement transformé 200 valeurs en `NA` ressemble en tout point à une exécution propre tant que vous ne demandez pas.

```
if (nrow(problems(muac)) > 0) {
  print(problems(muac))
  stop("L'entree ne correspond pas a sa specification de colonnes.")
}
```

Dans un script produisant un chiffre rapporté, ce `stop()` est correct. Refuser de s'exécuter vaut mieux que produire un tableau à colonne silencieusement vide.

2.7 Vérifier la lecture avant d'en faire quoi que ce soit

Quatre contrôles, une minute, et ils attrapent presque tout :

```
library(dplyr)

check <- function(df, expected_rows = NULL) {
  cat("lignes/colonnes ", nrow(df), ncol(df), "\n")
  print(dplyr::glimpse(df))
  print(colSums(is.na(df)))
  if (!is.null(expected_rows)) {
    stopifnot(nrow(df) == expected_rows)
  }
}

check(muac, expected_rows = 4218)
```

L'assertion sur le nombre de lignes est celle qu'on saute. Un fichier arrivant avec 4 190 lignes au lieu de 4 218 a perdu quelque chose entre le serveur et vous, et le script doit le dire plutôt que de rapporter discrètement une charge de cas plus faible.

2.8 Séparateurs, encodages et milliers

Un fichier produit sur une machine Windows française ou espagnole est très souvent séparé par des points-virgules, car la virgule y est le séparateur décimal.

```
epi <- read_csv2(PATH)                       # separateur ; decimale ,
epi <- read_delim(PATH, delim = "\t")        # tabulations
```

`read_csv2()` est la variante européenne — point-virgule et virgule — et elle existe précisément parce que le cas est assez courant pour mériter sa propre fonction.

L'encodage se manifeste dans les noms de communes : Gonaïves, L'Estère, Anse-Rouge.

```
muac <- read_csv(PATH, locale = locale(encoding = "UTF-8"))
muac <- read_csv(PATH, locale = locale(encoding = "latin1"))   # ce qu'Excel ecrit souvent
```

`latin1` ne lève jamais d'erreur, car tout octet est un caractère latin1 valide. Cela en fait un mauvais diagnostic et un repli correct : si un fichier lu en `latin1` affiche `GonaÃ¯ves`, il était en UTF-8 depuis le début et vous avez dit le contraire au lecteur.

Les séparateurs de milliers transforment un nombre en texte :

```
epi <- read_csv(PATH, locale = locale(grouping_mark = ","))
```

Sans cela, `target_population` arrive en `"1,480"` — une colonne de caractères sur laquelle toute opération arithmétique échoue ou concatène.

2.9 Excel

```
library(readxl)

excel_sheets(path)
#> [1] "Cover" "Data" "Codebook" "Sheet3"

data <- read_excel(path, sheet = "Data")
```

`read_excel()` sans `sheet` renvoie la **première** feuille, qui dans un classeur de programme est le plus souvent une page de garde. Listez toujours les feuilles d'abord.

2.9.1 L'en-tête n'est pas la ligne un

```
data <- read_excel(path, sheet = "Data", skip = 3)
```

Les classeurs de programme portent couramment un titre, une ligne de logo et une ligne vide au-dessus du véritable en-tête. Le symptôme, ce sont des noms de colonnes comme `...1`, `...2`.

2.9.2 Les cellules fusionnées

Une cellule fusionnée porte sa valeur en haut à gauche et rien dans le reste. Quand un nom de district est fusionné sur ses formations, seule la première ligne de chaque district en porte un :

```
data <- data |> tidyr::fill(district, .direction = "down")
```

`fill()` est correct ici et dangereux en général : il n'est juste que parce que les blancs viennent d'une fusion, non d'une non-réponse. Ne l'appliquez jamais à une colonne où un blanc pourrait signifier « non renseigné ».

2.9.3 Les dates en numéro de série

Excel stocke les dates comme un nombre de jours depuis le 30/12/1899. `read_excel()` les convertit généralement, mais une colonne typée en texte dans le classeur arrive sous forme de numéro de série brut.

```
data$screening_date <- as.Date(as.numeric(data$screening_date), origin = "1899-12-30")
```

Si une colonne de dates arrive en nombres à cinq chiffres, voilà pourquoi.

2.9.4 Les types sous Excel

```
data <- read_excel(
  path,
  sheet = "Data",
  col_types = c("text", "text", "date", "numeric", "text", "numeric", "text", "text")
)
```

`readxl` attend un vecteur positionnel plutôt qu'une spécification nommée : une colonne ajoutée à l'export décale donc tout ce qui suit. Vérifiez `ncol()` face à la longueur de votre vecteur, ou lisez en texte et convertissez avec dplyr.

2.10 La fonction de lecture, assemblée

Tout ce qui précède appartient à une fonction unique que les scripts partagent :

```
# R/read_register.R
read_register <- function(path) {
  muac <- readr::read_csv(
    path,
    col_types = readr::cols(
      child_id       = readr::col_character(),
      commune        = readr::col_character(),
      screening_date = readr::col_date(format = "%Y-%m-%d"),
      age_months     = readr::col_integer(),
      sex            = readr::col_character(),
      muac_mm        = readr::col_integer(),
      oedema         = readr::col_character(),
      outcome        = readr::col_character()
    ),
    na = c("", "NA", "-99")
  )

  if (nrow(readr::problems(muac)) > 0) {
    print(readr::problems(muac))
    stop("L'entree ne correspond pas a sa specification de colonnes.")
  }
  stopifnot(!any(is.na(muac$child_id)))

  muac
}
```

Une fonction, un seul endroit à corriger quand le prochain export changera.

2.11 Ce qui vient ensuite

Le CSV est entré et ses types sont justes. La leçon suivante traite les fichiers qui transportent plus que des valeurs — les exports Stata et SPSS, où `1 = Oui` voyage avec les données et où `haven` est la raison de les lire sous R même si vous les analyserez ailleurs.

Deuxième partie

Des données qui gardent leur sens

CHAPITRE 3

Les fichiers Stata et SPSS gardent leurs étiquettes

Leçon 3 sur 8 · 75 min

haven, la classe labelled, et pourquoi le .dta d'un institut de sondage doit être lu sous R même si l'analyse se fait ailleurs — plus les deux façons de perdre un dictionnaire de codes sans s'en apercevoir.

3.1 Ce qu'un CSV jette

Un institut de sondage livre un `.dta` ou un `.sav`. À l'intérieur, une colonne n'est pas seulement des valeurs — elle porte une **étiquette de variable** (« Le chef de ménage a consenti à l'entretien ») et des **étiquettes de valeurs** (`1 = Oui`, `2 = Non`, `9 = Ne sait pas`).

Exportez-le en CSV et les deux disparaissent. Vous obtenez une colonne de `1`, `2` et `9`, un dictionnaire de codes en PDF que quelqu'un doit conserver, et une moyenne de 1,7 qui ne signifie rien.

```
library(haven)

survey <- read_dta(here("data", "raw", "household-survey.dta"))
```

`haven` lit les étiquettes en même temps que les valeurs. C'est la raison d'être de cette leçon, et la raison de lire un fichier d'enquête sous R même si l'analyse se fera en Python : **R préserve davantage du fichier que pandas**, et un CSV accompagné d'un dictionnaire que vous avez écrit vous-même est un moins bon artefact que l'original.

3.2 La classe `labelled`

`read_dta()` vous donne des colonnes de classe `haven_labelled` : les valeurs sous-jacentes, avec leurs étiquettes attachées.

```
class(survey$consent)
#> [1] "haven_labelled" "vctrs_vctr" "double"

attr(survey$consent, "labels")
#>        Oui        Non Ne sait pas
#>          1          2          9

attr(survey$consent, "label")
#> [1] "Le chef de menage a consenti a l'entretien"
```

Les valeurs sont toujours là — l'arithmétique fonctionne — et c'est justement le piège. Une colonne étiquetée est un nombre du point de vue de `mean()` :

```
mean(survey$consent, na.rm = TRUE)
#> [1] 1.7
```

Ce chiffre est la moyenne de `1`, `2` et `9`. Il n'a aucun sens et il n'avertit de rien.

3.3 Convertir délibérément

`haven` propose trois gestes, et le bon dépend de ce qu'est la colonne.

Une variable catégorielle devient un facteur.

```
library(dplyr)

survey <- survey |>
  mutate(consent = as_factor(consent))

levels(survey$consent)
#> [1] "Oui" "Non" "Ne sait pas"
```

`as_factor()` — celui de haven, non le `as.factor()` de base — utilise les étiquettes de valeurs comme niveaux, dans l'ordre que le fichier déclare plutôt qu'alphabétiquement. Cet ordre est un cadeau : celui du dictionnaire de codes est généralement celui qu'un rapport attend.

Un véritable nombre perd ses étiquettes.

```
survey <- survey |> mutate(hh_size = zap_labels(hh_size))
```

`zap_labels()` retire l'étiquetage et laisse les valeurs. À employer là où la colonne est réellement une quantité qui portait accessoirement une étiquette.

Tout d'un coup, quand le fichier est uniformément catégoriel :

```
survey <- read_dta(path) |> as_factor()
```

Appliquer `as_factor()` au tibble entier convertit chaque colonne étiquetée. Rapide, et faux pour tout fichier mêlant catégories et quantités — vérifiez avec `glimpse()` avant d'y recourir.

3.4 Les deux façons de perdre un dictionnaire de codes

3.4.1 Lire avec les étiquettes déjà appliquées

```
survey <- read_dta(path, .name_repair = "unique")
survey <- as_factor(survey)          # les valeurs ont disparu, restent les etiquettes
```

Les codes sous-jacents sont désormais indisponibles. Cela compte plus qu'il n'y paraît : le dictionnaire, le questionnaire, le plan de tabulation et toute analyse antérieure renvoient tous à `1` et `2`. À un collègue qui demande « combien ont été codés 9 ? », un facteur ne répond pas.

Gardez les deux :

```
survey <- read_dta(path)

codes <- survey |> mutate(across(where(is.labelled), zap_labels))
labels <- survey |> as_factor()
```

3.4.2 Écrire le CSV et passer à autre chose

```
readr::write_csv(as_factor(survey), "survey.csv")
```

Le CSV porte désormais `"Oui"` et `"Non"`, ce qui est lisible et perd dans une autre direction : les codes ont disparu, et avec eux la distinction entre « Ne sait pas » et une véritable valeur manquante.

Si vous devez remettre un CSV à une analyse Python, remettez un dictionnaire avec lui, généré depuis le fichier plutôt que saisi :

```
codebook <- tibble::tibble(
  variable = names(survey),
  label    = vapply(survey, function(x) attr(x, "label") %||% NA_character_, character(1))
)

readr::write_csv(codebook, "survey-codebook.csv")
```

Les étiquettes de valeurs demandent une ligne par niveau, ce que `labelled::look_for()` produit directement si vous disposez de ce paquet. Dans tous les cas, le dictionnaire est **dérivé**, non transcrit — un dictionnaire transcrit s'écarte du fichier qu'il décrit.

3.5 Les valeurs manquantes définies par l'utilisateur

Stata et SPSS distinguent des sortes de manquant : refus, sans objet, non posé. `haven` les préserve sous forme de **NA étiquetés** plutôt que de les fondre en un seul.

```
survey$income
#> <labelled<double>[3]>
#> [1]  4500 NA(a) NA(b)
```

`NA(a)` et `NA(b)` restent des `NA` pour `is.na()` : rien ne devient donc silencieusement un nombre. Mais la distinction survit, et elle est souvent le constat — une question refusée par 40 % des répondants est un problème différent d'une question qui ne les concernait pas.

```
survey <- survey |> mutate(income = zap_missing(income))
```

`zap_missing()` les aplatit en `NA` ordinaires une fois que vous les avez comptés. Comptez d'abord.

3.6 SPSS et les autres formats

```
read_sav(path)     # SPSS .sav
read_por(path)     # SPSS portable
read_dta(path)     # Stata
read_xpt(path)     # SAS transport
```

Tous renvoient la même structure étiquetée : tout ce qui précède s'applique tel quel.

L'écriture est symétrique, et parfois nécessaire quand un ministère attend un `.dta` :

```
write_dta(survey, "cleaned.dta")
```

Stata impose aux noms de variables des règles que R n'a pas — pas de points, 32 caractères — et `write_dta()` lèvera une erreur plutôt que de les déformer silencieusement. C'est le comportement souhaitable.

3.7 Où cela rejoint le cours Python

Le cours *Python pour les données de programme* dit, dans sa leçon sur les codes et les sentinelles, que c'est le seul endroit où la réponse est « utilisez l'autre langage ». C'est cette leçon-ci qu'il désigne.

pandas lit un `.dta` et vous donne soit les codes, soit les étiquettes, jamais les deux, et ne préserve pas du tout les valeurs manquantes étiquetées. Si les étiquettes comptent — et sur un fichier d'enquête, c'est toujours le cas — lisez sous R, convertissez délibérément, et exportez à la fois les codes et un dictionnaire dérivé.

3.8 Ce qui vient ensuite

Les étiquettes sont attachées et converties en facteurs. La leçon suivante porte sur ce qu'est réellement un facteur en R, et sur le problème d'ordre que `as_factor()` a résolu ici pour vous et ne résoudra pas pour une colonne que vous construisez vous-même.

CHAPITRE 4

Les facteurs, et l'ordre qu'exige un rapport

Leçon 4 sur 8 · 75 min

Ce qu'est réellement un facteur, le piège de as.numeric() qui renvoie silencieusement des positions de niveaux, et les verbes forcats qui mettent une échelle JMP dans l'ordre de l'échelle plutôt que dans celui de l'alphabet.

4.1 Un facteur est un entier assorti d'une table de correspondance

```
service <- factor(c("basic", "limited", "unimproved", "basic"))

levels(service)
#> [1] "basic"      "limited"    "unimproved"

as.integer(service)
#> [1] 1 2 3 1
```

En dessous, R stocke `1 2 3 1` et un vecteur de caractères contenant les niveaux. Tout ce qui suit — le problème d'ordre, le piège de `as.numeric()`, les niveaux inutilisés — découle de ce seul fait.

Remarquez que les niveaux sont **alphabétiques**. Rien dans `basic`, `limited`, `unimproved` n'indique que ce soit l'ordre, et c'est exactement l'inverse pour une échelle de service du JMP, où non améliorée est le pire et où le service de base vaut mieux que le service limité. Laissé tel quel, chaque tableau et chaque graphique bâti sur cette colonne se lit dans un ordre qui ne signifie rien.

4.2 Le piège de `as.numeric()`

C'est l'erreur de facteur la plus coûteuse, et elle n'avertit pas.

```
age <- factor(c("10", "9", "8"))

as.numeric(age)
#> [1] 1 3 2
```

Ce sont des **positions de niveaux**, non des valeurs. Les niveaux sont alphabétiques — `"10"`, `"8"`, `"9"` — donc `"10"` est en position 1 et `"9"` en position 3.

```
as.numeric(as.character(age))
#> [1] 10 9 8
```

`as.character()` d'abord, systématiquement. Le cas survient dès qu'une colonne numérique a été lue comme du texte — une colonne Excel avec un `"n/a"` égaré, un CSV lu en `col_character()` — puis convertie en facteur en chemin.

Méfiez-vous d'une moyenne suspectement proche du nombre de catégories.

4.3 Fixer l'ordre voulu

```
library(forcats)

ladder <- fct_relevel(service, "unimproved", "limited", "basic")
levels(ladder)
#> [1] "unimproved" "limited"    "basic"
```

`fct_relevel()` nomme l'ordre explicitement. Déclarez-le une fois, près de l'endroit où la colonne est créée, et chaque tableau, graphique et modèle en aval en hérite.

Deux verbes forcats gagnent immédiatement leur place :

```
fct_infreq(source)        # le niveau le plus frequent en premier
fct_reorder(name, value)  # ordonner un facteur par une autre colonne
```

`fct_reorder()` est ce qui transforme un graphique en barres illisible en graphique lisible — les communes triées par leur taux de MAG plutôt que par l'alphabet — et il le fait sans un `levels =` manuel à mettre à jour quand les données changent.

```
library(dplyr)

by_commune |>
  mutate(commune = fct_reorder(commune, gam_rate)) |>
  ggplot2::ggplot(ggplot2::aes(gam_rate, commune)) +
  ggplot2::geom_col()
```

4.4 Les facteurs ordonnés

`fct_relevel()` fixe l'ordre des niveaux. Il ne rend **pas** le facteur comparable :

```
service[1] >= service[2]
#> Warning: '>=' not meaningful for factors
#> [1] NA
```

Pour une échelle, où « au moins de base » est une vraie question, déclarez-la ordonnée :

```
ladder <- factor(
  c("basic", "limited"),
  levels = c("unimproved", "limited", "basic"),
  ordered = TRUE
)

ladder[1] >= ladder[2]
#> [1] TRUE
```

`>= "basic"` calcule alors directement l'indicateur de couverture du JMP. Sans `ordered = TRUE`, l'expression renvoie `NA` avec un avertissement — qui, s'il n'est pas lu, devient un chiffre de couverture à `NA` ou, pire, un filtre qui ne sélectionne rien.

Employez les facteurs ordonnés avec parcimonie. Ils changent la façon dont les modèles traitent la variable (contrastes polynomiaux plutôt qu'indicatrices), ce qui est rarement souhaité dans une régression. Pour une échelle de service, une bande de sévérité ou une phase IPC, l'ordre en vaut la peine. Pour une commune, non.

4.5 Les niveaux inutilisés, et les deux comportements qui divergent

Un niveau sans ligne reste un niveau. Son apparition dans votre sortie dépend de la fonction interrogée, et R n'est pas cohérent avec lui-même sur ce point.

```
d <- tibble::tibble(k = factor(c("a", "b", "a"), levels = c("a", "b", "c")))

nrow(dplyr::count(d, k))
#> [1] 2

length(table(d$k))
#> [1] 3
```

`count()` écarte le niveau vide ; `table()` le conserve. Les deux sont justes pour des questions différentes, et la question est celle-là même que le cours *Python pour les données de programme* soulève à propos d'`observed=` : rapporter sur les formations ayant transmis des données veut le niveau vide en moins, et rapporter la couverture face à une liste de formations qui *auraient dû* transmettre le veut présent, car une formation à zéro ligne est justement le constat.

Dites lequel vous voulez :

```
dplyr::count(d, k, .drop = FALSE)   # conserver le niveau vide
droplevels(d$k)                     # retirer les niveaux sans ligne
forcats::fct_drop(d$k)              # la meme chose, en forcats
```

4.6 Réduire une longue traîne

Dix sources d'eau, c'est trop de lignes pour un tableau de rapport et trop de couleurs pour un graphique.

```
source <- factor(wash$water_source)
length(levels(source))
#> [1] 10

levels(fct_lump_n(source, 4))
#> [1] "borehole" "piped-into-dwelling" "piped-into-yard" "public-tap" "Other"
```

`fct_lump_n()` conserve les quatre plus fréquents et replie le reste dans `Other`.

Ne repliez pas avant de calculer un indicateur. `Other` mêle ici les sources protégées, qui sont améliorées, et l'eau de surface, qui ne l'est pas : un taux de couverture calculé après repli est donc faux. Repliez pour la présentation, sur une copie, une fois les chiffres arrêtés.

```
fct_recode(source,
  improved = "borehole",
  improved = "protected-well",
  unimproved = "surface-water"
)
```

`fct_recode()` est l'alternative explicite et c'est ce qu'il faut pour une classification : elle nomme chaque correspondance, si bien qu'une source apparaissant au prochain export sans figurer dans la liste reste elle-même au lieu de rejoindre silencieusement un groupe.

4.7 Facteurs et valeurs manquantes

`NA` n'est pas un niveau : il survit donc à tout réordonnancement et n'apparaît dans `table()` que si vous le demandez :

```
table(muac$outcome, useNA = "ifany")
```

Pour faire de l'absence une catégorie explicite — ce qui est souvent juste dans un rapport, où « non renseigné » est un constat et non un trou :

```
muac <- muac |> mutate(outcome = fct_na_value_to_level(outcome, level = "non renseigne"))
```

Une fois devenue un niveau, elle sera comptée, tracée et sommée comme les autres. C'est le but, et c'est aussi le risque : un taux calculé sur un facteur incluant « non renseigné » n'a pas le même dénominateur qu'un taux qui l'exclut.

4.8 Là où les facteurs mordent dans une jointure

```
left_join(muac, sites, by = "commune")
```

Si `commune` est un facteur dans un tableau et du caractère dans l'autre, dplyr convertit et avertit. Si c'est un facteur dans les deux **avec des niveaux différents**, la jointure fonctionne quand même — dplyr compare les étiquettes, non les entiers — mais les niveaux du résultat sont l'union des deux, ce qui peut discrètement réintroduire des catégories vides.

L'habitude la plus simple : **joignez sur des colonnes de caractères, convertissez en facteur ensuite.** Les facteurs sont un dispositif de présentation et de modélisation, non un type de clé.

4.9 Ce qui vient ensuite

Les colonnes portent désormais leur sens et leur ordre. L'unité suivante les travaille : les verbes dplyr, puis l'appel de regroupement où naissent la plupart des erreurs d'indicateurs.

Troisième partie

Les verbes

CHAPITRE 5

Les verbes dplyr, employés délibérément

Leçon 5 sur 8 · 85 min

filter, select, mutate et arrange — ce que chacun fait d'une valeur manquante, pourquoi if_else() refuse ce que ifelse() accepte en silence, et le across() qui applique une règle à plusieurs colonnes.

5.1 Cinq verbes et un tube

```
library(dplyr)

muac |>
  filter(!is.na(muac_mm)) |>
  mutate(gam = muac_mm < 125) |>
  group_by(commune) |>
  summarise(assessed = n(), cases = sum(gam)) |>
  arrange(desc(cases / assessed))
```

Chaque verbe prend un tableau de données et en renvoie un. Le tube passe le résultat au suivant. Voilà toute la grammaire ; le reste de cette leçon porte sur ce que fait chaque verbe quand les données ne sont pas propres — c'est-à-dire toujours.

`|>` est le tube natif de R, disponible depuis la 4.1. `%>%` de magrittr le précède et se comporte de façon quasi identique pour ce type de code ; si les scripts de votre équipe l'emploient, tout ce qui suit fonctionne tel quel.

5.2 `filter()` écarte les valeurs manquantes, en silence

```
nrow(muac)
#> [1] 4218

nrow(filter(muac, muac_mm < 125))
#> [1] 319

sum(is.na(muac$muac_mm))
#> [1] 72
```

Une comparaison avec `NA` vaut `NA`, et `filter()` ne conserve que les lignes `TRUE`. Ces 72 enfants non mesurés ont donc disparu, sans que rien ne le dise.

C'est le même comportement que pandas, et cela mérite d'être énoncé dans les deux langages parce que c'est la façon la plus rapide de changer un dénominateur par accident. Soyez explicite sur ce que vous vouliez dire :

```
# Enfants mesures en dessous de 125 mm.
filter(muac, muac_mm < 125)

# Enfants dont on ne sait pas qu'ils sont a 125 mm ou plus -- non mesures compris.
filter(muac, is.na(muac_mm) | muac_mm < 125)
#> 391 lignes
```

Soixante-douze lignes séparent ces deux questions. Laquelle vous voulez dépend de savoir si un enfant non mesuré est un non-cas ou un inconnu, et c'est une décision qui relève d'un journal de nettoyage, non d'un comportement par défaut.

5.3 `select()` et ses assistants

```
muac |> select(child_id, commune, muac_mm)
muac |> select(-oedema)
muac |> select(starts_with("muac"), ends_with("_date"))
muac |> select(where(is.numeric))
```

`select()` renomme en sélectionnant, ce qui est la manière tidy de corriger une fois pour toutes les noms de colonnes d'un export :

```
epi |> select(facility = facility_id, month = period, doses = doses_administered)
```

Deux habitudes à prendre. **Sélectionnez avant de joindre**, pour que le résultat ne traîne pas quarante colonnes dont vous ne vouliez pas. Et **ne sélectionnez jamais par position** — `select(3:5)` casse le jour où l'export gagne une colonne, et il casse en silence puisque le code s'exécute toujours.

5.4 `mutate()` et les deux fonctions conditionnelles

```
muac <- muac |>
  mutate(
    gam = muac_mm < 125,
    sam = muac_mm < 115,
    band = case_when(
      muac_mm < 115 ~ "severe",
      muac_mm < 125 ~ "moderate",
      .default = "normal"
    )
  )
```

`case_when()` évalue dans l'ordre et la première correspondance l'emporte : `< 115` doit donc venir en premier. Son `.default` attrape **tout le reste, y compris les `NA`**, ce qui n'est presque jamais souhaité :

```
case_when(c(110, 130, NA) < 125 ~ "case", .default = "not")
#> [1] "case" "not"  "not"
```

La mesure manquante est devenue `"not"`. Traitez-la explicitement :

```
band = case_when(
  is.na(muac_mm) ~ NA_character_,
  muac_mm < 115  ~ "severe",
  muac_mm < 125  ~ "moderate",
  .default = "normal"
)
```

Placer la branche `is.na()` en premier est l'habitude à prendre. C'est le même piège que le `default` de `np.select` dans le cours Python, et il vaut d'être reconnu aux deux endroits.

5.4.1 `if_else()` plutôt que `ifelse()`

```
ifelse(c(TRUE, FALSE), 1L, "x")
#> [1] "1" "x"
```

Le `ifelse()` de R de base a silencieusement fondu un entier et une chaîne en un vecteur de caractères. Une colonne que vous croyiez numérique est désormais du texte, et vous l'apprendrez trois étapes plus loin quand une somme échouera.

```
if_else(c(TRUE, FALSE), 1L, "x")
#> Error: Can't combine `true` <integer> and `false` <character>.
```

Le `if_else()` de dplyr refuse. **Préférez-le partout**, pour la raison sur laquelle ce cours revient sans cesse : une erreur là où la faute est commise vaut mieux qu'une mauvaise réponse trois étapes plus loin.

`if_else()` accepte aussi un argument `missing =`, qui permet de dire ce qu'une condition `NA` doit produire plutôt que de la laisser se propager.

5.5 `across()` : une règle, plusieurs colonnes

```
muac |>
  summarise(across(c(muac_mm, age_months), ~ mean(.x, na.rm = TRUE)))
#> # A tibble: 1 × 2
#>   muac_mm age_months
#> 1    140.       30.9
```

`across()` est ce qui évite qu'un script de nettoyage soit quinze lignes presque identiques.

```
survey |> mutate(across(where(is.character), stringr::str_squish))

survey |> mutate(across(starts_with("fcs_"), ~ if_else(.x > 7, NA_real_, .x)))
```

Cette seconde ligne est la règle sécurité alimentaire du cours *Python pour les données de programme*, en une instruction : une valeur de consommation supérieure à sept jours est impossible et doit devenir manquante plutôt qu'être plafonnée, sur les huit groupes alimentaires à la fois.

La forme `~ .x` abrège une fonction à un argument. `\(x) ...` est l'écriture plus récente de R de base et fonctionne à l'identique :

```
survey |> mutate(across(starts_with("fcs_"), \(x) if_else(x > 7, NA_real_, x)))
```

5.6 `arrange()`

```
muac |> arrange(commune, desc(screening_date))
```

`NA` se classe **en dernier** quel que soit le sens, ce qui est généralement souhaitable et masque parfois un problème — une colonne entièrement manquante paraît triée.

Trier range des lignes. Cela ne dit rien de la réalité des écarts entre lignes voisines, point que développe le projet *Tableau de bord de programme nutritionnel*, où neuf communes sur douze ont des intervalles de confiance qui se recouvrent.

5.7 `distinct()` et le comptage

```
n_distinct(muac$commune)
#> [1] 12

muac |> distinct(child_id, .keep_all = TRUE)
```

`distinct()` sans `.keep_all = TRUE` ne renvoie que les colonnes nommées, ce qui surprend qui attendait des lignes dédupliquées. Avec, c'est la **première** ligne de chaque groupe qui est conservée — triez donc d'abord si le choix de la survivante compte.

```
muac |> count(commune, sort = TRUE)
```

`count()` équivaut à `group_by() |> summarise(n = n()) |> ungroup()`, et c'est le bon outil chaque fois que c'est tout ce que vous voulez.

5.8 Les jointures, et le contrôle qui les accompagne

```
daily <- attendance |>
  left_join(roster, by = "student_id")
```

La défaillance à prévenir est une jointure qui multiplie les lignes parce que le côté droit a des clés dupliquées. dplyr avertit, mais un avertissement dans un long script défile :

```
before <- nrow(attendance)
daily <- attendance |> left_join(roster, by = "student_id")
stopifnot(nrow(daily) == before)
```

Cette assertion est l'équivalent R du `validate="many_to_one"` de pandas, et elle a attrapé deux élèves inscrits deux fois dans le projet *Assiduité scolaire* — un transfert non enregistré ayant laissé les deux inscriptions actives.

`anti_join()` est la façon la plus rapide de voir ce qui n'a pas correspondu :

```
attendance |> anti_join(roster, by = "student_id") |> distinct(student_id)
```

Un résultat vide signifie que chaque ligne de présence a trouvé son élève. Un résultat non vide est une liste à rapporter à qui tient la liste des élèves.

5.9 Ce qui vient ensuite

Les verbes sont acquis. La leçon suivante est celle vers laquelle ils mènent et celle où naissent la plupart des erreurs d'indicateurs : `group_by()` et `summarise()`, et les trois choses que R fait des clés manquantes, des niveaux inutilisés et du regroupement résiduel — à chaque fois à l'inverse de pandas.

CHAPITRE 6

Là où naissent la plupart des erreurs d'indicateurs

Leçon 6 sur 8 · 90 min

group_by() et summarise() — le regroupement qui survit à l'appel, la clé manquante qui devient son propre groupe, et les trois comportements par défaut exactement inverses de ceux de pandas.

6.1 Un indicateur, ce sont deux nombres

Presque tout chiffre rapporté dans ce secteur est un numérateur sur un dénominateur, et presque toute dispute sur un chiffre est une dispute sur le dénominateur. La conséquence pratique : **calculez les deux dans le même appel.**

```
library(dplyr)

by_commune <- muac |>
  mutate(
    assessed = !is.na(muac_mm) | !is.na(oedema),
    gam = assessed & (muac_mm < 125 | oedema == "true")
  ) |>
  group_by(commune) |>
  summarise(
    screened  = n(),
    assessed  = sum(assessed, na.rm = TRUE),
    gam_cases = sum(gam, na.rm = TRUE),
    .groups = "drop"
  ) |>
  mutate(gam_rate = round(gam_cases / assessed, 3))
```

Trois colonnes, et chacune porte quelque chose. `screened` est le nombre d'enfants venus ; `assessed` le nombre ayant produit une mesure exploitable, et c'est le dénominateur ; `gam_cases` est le numérateur. Deux calculs séparés divergent — l'un reçoit un filtre que l'autre n'a pas — et le rapport devient un chiffre que personne ne peut reconstituer.

6.2 `n()` et les deux façons de compter

```
summarise(muac, rows = n(), measured = sum(!is.na(muac_mm)))
```

`n()` compte les lignes du groupe, y compris celles à valeur manquante. Sommer un booléen compte les `TRUE`. Cette distinction fait la différence entre les deux dénominateurs ci-dessus, et mérite d'être énoncée à voix haute chaque fois que vous employez l'un des deux.

`sum()` sur un booléen contenant `NA` renvoie `NA` — R, encore une fois, bruyant :

```
sum(c(TRUE, NA))
#> [1] NA

sum(c(TRUE, NA), na.rm = TRUE)
#> [1] 1
```

Recourez à `na.rm = TRUE` délibérément, en sachant qu'il retire la ligne du décompte au lieu de la compter comme `FALSE`. Ce sont deux affirmations différentes.

6.3 Le regroupement survit, et c'est l'erreur la plus fréquente

C'est le point à intérioriser. `summarise()` retire la **dernière** variable de regroupement et laisse les autres :

```
out <- muac |>
  mutate(month = format(screening_date, "%Y-%m")) |>
  group_by(commune, month) |>
  summarise(n = n())
```

```
`summarise()` has regrouped the output.
i Summaries were computed grouped by commune and month.
i Output is grouped by commune.
i Use `summarise(.groups = "drop_last")` to silence this message.
```

```
class(out)[1]
#> [1] "grouped_df"

dplyr::group_vars(out)
#> [1] "commune"
```

Le résultat est **encore groupé par commune**. Chaque verbe qui suit opère silencieusement par commune :

```
out |> mutate(share = n / sum(n))     # part dans la commune, non dans le district
out |> arrange(desc(n))               # trie a l'interieur de chaque commune
out |> slice_max(n, n = 5)            # top 5 par commune, non au total
```

Chacune de ces lignes produit un chiffre plausible qui répond à une autre question que la vôtre. Rien ne lève d'erreur.

Dites ce que vous voulez, à chaque fois :

```
summarise(..., .groups = "drop")        # resultat degroupe -- la reponse habituelle
summarise(..., .groups = "drop_last")   # garder tout sauf la derniere (le defaut)
summarise(..., .groups = "keep")        # tout garder
```

Ou évitez complètement cet état avec le regroupement par opération, apparu dans dplyr 1.1 :

```
muac |>
  summarise(n = n(), .by = c(commune, month))
```

`.by` groupe pour ce seul appel et renvoie un résultat dégroupé. **Préférez-le** dans du code neuf : sans état résiduel, il n'y a pas d'erreur de cette forme.

6.4 Une clé manquante devient son propre groupe

Ici, R et pandas prennent des paris opposés.

```
d <- tibble::tibble(g = c("a", NA, "a"), v = c(1, 2, 3))
```

```
d |> group_by(g) |> summarise(s = sum(v), .groups = "drop")
```

```
# A tibble: 2 × 2
  g         s
  <chr> <dbl>
1 a         4
2 <NA>      2
```

Le groupe `NA` est **conservé**. pandas l'écarte par défaut : la même opération dans les deux langages vous donne donc un ensemble de lignes différent et un total différent.

Le comportement de R est le plus sûr, et dans ce secteur la clé manquante est généralement la plus intéressante : une formation sanitaire sans district saisi, un ménage sans code de site, un enfant sans commune. Mais cela signifie qu'un tableau peut arriver dans un rapport avec une ligne `<NA>` que personne n'avait prévu de publier.

```
d |> filter(!is.na(g)) |> group_by(g) |> summarise(s = sum(v), .groups = "drop")
```

L'exclure est légitime. L'exclure **sans le remarquer**, c'est ainsi que les parties cessent de faire le tout. Préférez une assertion là où la clé devrait être complète :

```
stopifnot(!any(is.na(muac$commune)))
```

6.5 Niveaux de facteur inutilisés : `count()` écarte, `table()` conserve

Le second renversement, et R n'est pas cohérent avec lui-même ici — raison pour laquelle la leçon sur les facteurs l'aborde et celle-ci le répète.

```
d <- tibble::tibble(k = factor(c("a", "b", "a"), levels = c("a", "b", "c")))

nrow(count(d, k))                # 2   -- le niveau vide a disparu
nrow(count(d, k, .drop = FALSE)) # 3   -- conserve
length(table(d$k))               # 3   -- conserve
```

Les deux sont justes pour des questions différentes, et c'est la question même que le cours Python soulève à propos d'`observed=`. Rapporter sur les formations ayant transmis des données veut le niveau vide en moins ; rapporter la couverture face à une liste de formations qui *auraient dû* transmettre le veut présent, car une formation à zéro ligne est justement le constat.

Passez `.drop` explicitement dans tout ce qui produit un tableau rapporté.

6.6 `mutate()` groupé : une statistique de groupe sur chaque ligne

`summarise()` réduit ; `mutate()` sur un tableau groupé renvoie une valeur par ligne d'origine. C'est ainsi qu'on compare une ligne à son propre groupe sans jointure :

```
muac |>
  group_by(commune) |>
  mutate(
    commune_mean = mean(muac_mm, na.rm = TRUE),
    vs_commune   = muac_mm - commune_mean
  ) |>
  ungroup()
```

C'est le contrôle qu'emploie le projet *Analyse d'enquête SMART* pour repérer une équipe mesurant un demi-kilo trop léger.

Noter le `ungroup()`. Un tableau groupé qui s'échappe dans la suite d'un script est la même erreur que ci-dessus, arrivant par une autre porte.

6.7 `count()` quand c'est tout ce que vous voulez

```
muac |> count(commune, outcome, sort = TRUE)
muac |> count(commune, wt = muac_mm)          # sommer une colonne plutot que compter
```

`count()` groupe, compte et dégroupe en un appel : il ne peut donc pas laisser d'état derrière lui. Là où un décompte suffit, c'est le meilleur verbe.

6.8 Deux clés, et le tableau qu'attend un rapport

```
by_commune_month <- muac |>
  mutate(month = format(screening_date, "%Y-%m")) |>
  summarise(cases = sum(gam, na.rm = TRUE), .by = c(commune, month))

wide <- by_commune_month |>
  tidyr::pivot_wider(names_from = month, values_from = cases, values_fill = 0)
```

`values_fill = 0` est sans risque ici, car une commune-mois sans cas en avait réellement zéro. Ce serait faux pour un taux, où l'absence signifie « non calculé » et non « zéro pour cent » — distinction à vérifier chaque fois que vous y recourez.

6.9 Écrire le résultat

```
readr::write_csv(by_commune, here::here("outputs", "tables", "gam_by_commune.csv"))
```

Pour un tableau destiné à être lu plutôt que calculé, écrivez la définition avec lui :

```
definitions <- tibble::tibble(
  indicateur   = "MAG",
  numerateur   = "PB < 125 mm ou oedemes bilateraux prenant le godet",
  denominateur = "enfants avec une mesure de PB ou une evaluation d'oedemes saisie"
)

readr::write_csv(definitions, here::here("outputs", "tables", "definitions.csv"))
```

Un nombre et sa définition voyagent ensemble, sinon la dispute mensuelle sur le dénominateur recommence.

6.10 Les trois renversements, en un seul endroit

	pandas	**R / dplyr**
Valeur manquante dans `mean()`	ignorée en silence	renvoie `NA` jusqu'à `na.rm`
Clé de groupe manquante	écartée	conservée comme groupe à part
Catégorie inutilisée	écartée par `observed=True`	conservée par `table()`, écartée par `count()`

Aucun n'est meilleur. Chacun est un pari différent sur ce que le silence doit signifier, et savoir dans quel pari on se trouve constitue l'essentiel du métier.

6.11 Ce qui vient ensuite

Le tableau agrège correctement et les deux moitiés de chaque taux proviennent d'un seul appel. L'unité suivante restructure : `pivot_longer()` sur le groupe répété aplati sous lequel arrive un export CommCare ou Kobo, puis la fonction qui fait tourner tout le nettoyage au trimestre suivant.

Quatrième partie

Restructurer et répéter

CHAPITRE 7

Défaire un groupe répété aplati

Leçon 7 sur 8 · 80 min

pivot_longer() avec .value, l'appel unique qui retransforme un export CommCare ou Kobo en une ligne par personne — plus l'avertissement de pivot_wider() qui signifie que votre clé n'en est pas une.

7.1 Ce qu'une plateforme de collecte fait d'un groupe répété

Un questionnaire ménage interroge sur chaque membre. Sous CommCare, KoboToolbox ou ODK, les réponses reviennent **aplaties** — une ligne par ménage, le groupe répété étalé sur des colonnes numérotées :

```
hh    member_1_age  member_1_sex  member_2_age  member_2_sex
H1              34  f                        8  m
```

Rien dans cette forme n'est analysable. Vous ne pouvez ni calculer un âge moyen, ni filtrer sur les enfants, ni joindre quoi que ce soit, parce qu'une personne n'est pas une ligne.

7.2 `pivot_longer()` avec `.value`

```
library(tidyr)

long <- wide |>
  pivot_longer(
    starts_with("member_"),
    names_to = c("member", ".value"),
    names_pattern = "member_(\\d+)_(.*)"
  )
```

```
# A tibble: 2 × 4
  hh    member   age sex
  <chr> <chr>  <dbl> <chr>
1 H1    1         34 f
2 H1    2          8 m
```

Une ligne par personne, et `age` et `sex` sont des colonnes distinctes avec leurs propres types. C'est la forme qu'attend tout ce qui suit.

La sentinelle `.value` est la pièce à comprendre. Dans `names_to`, un nom ordinaire — `"member"` — devient une nouvelle colonne portant cette partie de l'ancien nom de colonne. `".value"` est différent : il dit *cette partie du nom est le nom de la colonne de sortie*. Ainsi `member_1_age` se scinde en `member = "1"` et une valeur appartenant à une colonne nommée `age`.

Sans `.value`, vous obtiendriez une unique colonne `value` mêlant nombres et texte, et une colonne `name` qu'il faudrait ensuite scinder et pivoter à nouveau.

7.2.1 Le motif

`names_pattern` est une expression régulière comportant un groupe de capture par entrée de `names_to`, dans l'ordre. `"member_(\\d+)_(.*)"` capture le numéro puis tout ce qui suit le second tiret bas.

Deux variantes lorsque les noms sont plus simples :

```
# Separateur fixe, sans expression reguliere.
pivot_longer(wide, starts_with("member_"),
             names_to = c("member", ".value"), names_sep = "_(?=[a-z]+$)")

# Une seule colonne de valeurs, nom conserve entier.
pivot_longer(wide, starts_with("crop_"),
             names_to = "slot", names_prefix = "crop_", values_to = "crop",
             values_drop_na = TRUE)
```

Le `values_drop_na = TRUE` de la seconde compte : un ménage avec trois emplacements de culture et deux cultures en a un vide, et une ligne `NA` par emplacement inutilisé est un bruit qui change tous les décomptes.

7.3 Vérifiez le nombre de lignes obtenu

Un pivot qui produit le mauvais nombre de lignes est l'erreur la plus facile à commettre et la plus difficile à voir, parce que le résultat paraît correct.

```
stopifnot(nrow(long) == sum(!is.na(wide$member_1_age)) + sum(!is.na(wide$member_2_age)))
```

Plus concrètement, comptez les personnes par ménage et comparez à la taille de ménage que le formulaire a enregistrée :

```
long |>
  count(hh, name = "members_found") |>
  left_join(households |> select(hh, hh_size), by = "hh") |>
  filter(members_found != hh_size)
```

Un résultat vide, c'est le contrôle qui passe. Un résultat non vide est une liste de ménages où la composition et la taille déclarée divergent — un constat de qualité des données avant d'être un problème de pivot.

7.4 `pivot_wider()` et l'avertissement qui veut dire quelque chose

L'opération inverse construit le tableau large qu'attend un rapport :

```
by_commune_month |>
  pivot_wider(names_from = month, values_from = cases, values_fill = 0)
```

Si plusieurs lignes partagent la même clé, tidyr ne lève pas d'erreur :

```
Warning: Values from `v` are not uniquely identified; output will contain
list-cols.
```

Cet avertissement signifie que votre clé n'en est pas une. La cellule contient désormais une liste des valeurs entrées en collision, et toute opération en aval échoue étrangement ou opère silencieusement sur une liste.

N'ajoutez pas `values_fn = sum` pour faire taire l'avertissement sans avoir d'abord cherché la cause des doublons. Il s'agit généralement de l'un de trois cas : un formulaire resoumis, une formation sanitaire rapportant deux fois dans le mois, ou une variable de regroupement oubliée. Seul le premier se somme sans risque.

```
by_commune_month |> count(commune, month) |> filter(n > 1)
```

Voilà le diagnostic, et il doit précéder le pivot plutôt que suivre l'avertissement.

7.4.1 `values_fill`

```
pivot_wider(..., values_fill = 0)
```

Sans risque pour un décompte, où une combinaison absente valait réellement zéro. **Faux pour un taux**, où l'absence signifie « non calculé » et non « zéro pour cent » — une commune sans dépistage en mars a un taux de MAG indéfini, non un taux de 0 %, et le remplir par zéro tire toutes les moyennes vers le bas.

7.5 Séparer et réunir des colonnes

```
tibble(x = c("2024-01", "2024-02")) |>
  separate_wider_delim(x, delim = "-", names = c("year", "month"))
```

```
# A tibble: 2 × 2
  year  month
  <chr> <chr>
1 2024  01
2 2024  02
```

`separate_wider_delim()` a remplacé l'ancien `separate()`, et le progrès tient à sa rigueur : une ligne au mauvais nombre de morceaux est une erreur plutôt qu'une troncature silencieuse. Là où une entrée irrégulière est attendue, dites-le :

```
separate_wider_delim(x, "-", names = c("year", "month"), too_few = "align_start")
```

L'opération inverse est `unite()`, utile pour bâtir une clé composite avant une jointure :

```
unite(df, "key", district, community, sep = "-", remove = FALSE)
```

`remove = FALSE` conserve les colonnes sources, dont vous avez presque toujours encore besoin.

7.6 L'imbrication, brièvement

Un groupe répété peut aussi rester imbriqué plutôt qu'aplati :

```
by_commune <- muac |>
  tidyr::nest(.by = commune)

by_commune$data[[1]]      # le tibble complet de la premiere commune
```

C'est la forme adaptée pour exécuter le même modèle ou le même résumé par commune sans boucle. Il vaut la peine de savoir que cela existe ; pour l'essentiel du rapportage de programme, le `.by` de `summarise()` y mène avec moins de machinerie.

7.7 Où cela rejoint le cours Python

Le cours Python couvre la même restructuration sous `melt` et `pivot`. Le vocabulaire diffère, les modes de défaillance non : un groupe répété qui doit devenir des lignes, une clé qui s'avère non unique, et une valeur de remplissage juste pour un décompte et fausse pour un taux.

7.8 Ce qui vient ensuite

Les données ont la bonne forme. La dernière leçon transforme tout ce qui précède en une fonction qu'un autre chargé pourra exécuter sur l'export du trimestre prochain sans rien modifier d'autre que ses arguments.

CHAPITRE 8

Une fonction qu'un autre chargé peut exécuter

Leçon 8 sur 8 · 90 min

Transformer un script en fonctions avec arguments et stopifnot(), l'évaluation tidy nécessaire pour passer un nom de colonne, et le script en ligne de commande qui s'exécute tel quel sur l'export du trimestre suivant.

8.1 Le test de la transmission

Une analyse est terminée quand quelqu'un d'autre peut l'exécuter sur l'export du trimestre prochain sans modifier le code. Non pas « sans trop le modifier » — sans le modifier.

Ce critère élimine les trois habitudes que tout script R accumule : un chemin écrit pour une machine, un seuil changé à la main entre deux exécutions, et une étape qui ne fonctionne que si l'on a déjà exécuté les lignes précédentes dans le bon ordre.

8.2 Du script à la fonction

Le nettoyage des leçons précédentes, en une fonction :

```r
# R/clean_register.R

clean_register <- function(muac, gam_mm = 125, sam_mm = 115) {
  stopifnot(is.data.frame(muac))
  required <- c("child_id", "commune", "muac_mm", "oedema")
  missing <- setdiff(required, names(muac))
  if (length(missing) > 0) {
    stop("Colonnes absentes de l'entree : ", paste(missing, collapse = ", "))
  }

  muac |>
    dplyr::mutate(
      # Un PB sous 40 est en centimetres dans un champ en millimetres. Applique
      # seulement sous un seuil qu'aucune mesure plausible n'atteint.
      muac_mm = dplyr::if_else(muac_mm < 40, muac_mm * 10L, muac_mm),
      muac_mm = dplyr::if_else(dplyr::between(muac_mm, 80L, 220L), muac_mm, NA_integer_),
      oedema  = dplyr::recode(
        tolower(trimws(oedema)),
        "true" = TRUE, "y" = TRUE, "yes" = TRUE,
        "false" = FALSE, "n" = FALSE, "no" = FALSE,
        .default = NA
      ),
      assessed = !is.na(muac_mm) | !is.na(oedema),
      gam = assessed & (muac_mm < gam_mm | oedema),
      sam = assessed & (muac_mm < sam_mm | oedema)
    )
}
```

Quatre aspects de cette forme sont délibérés.

Elle prend un tableau de données et en renvoie un. Aucune lecture de fichier, aucune écriture, aucun «-. C'est ce qui la rend testable et ce qui permet à un notebook et à un script de la partager.

Les seuils sont des arguments avec valeurs par défaut. Un seuil qu'on ne change qu'en éditant une ligne finira par être changé et non remis. Un seuil qui est un argument est visible dans l'appel qui a produit la sortie.

Elle valide son entrée. Un `stop()` nommant les colonnes absentes vaut mieux qu'un `NULL` se propageant dans un mutate et refaisant surface en erreur sans rapport deux fonctions plus loin.

Rien n'est groupé en sortie. Un tableau groupé qui s'échappe d'une fonction est l'erreur de la leçon sur le regroupement, arrivant par une porte nouvelle.

8.3 `stopifnot()` pour ce qui ne peut pas arriver

```
stopifnot(
  "chaque ligne exige un identifiant" = !any(is.na(muac$child_id)),
  "le PB doit etre plausible apres nettoyage" =
    all(is.na(muac$muac_mm) | dplyr::between(muac$muac_mm, 80, 220))
)
```

Les expressions nommées deviennent le message d'erreur, ce qui fait la différence entre un échec utile et `Error: ... is not TRUE`.

Employez `stop()` pour « cette entrée n'est pas celle qu'on m'avait promise » et `stopifnot()` pour « ceci ne peut pas arriver à moins que le code soit faux ». La distinction compte pour qui lira la défaillance à sept heures du matin.

8.4 Passer un nom de colonne : l'évaluation tidy

C'est le seul point de R qui surprenne les personnes venant de Python, et il n'y a qu'une chose à apprendre.

```
# Ne fonctionne pas.
rate_by <- function(df, group_col) {
  df |> dplyr::summarise(n = dplyr::n(), .by = group_col)
}

rate_by(muac, commune)
#> Error: object 'commune' not found
```

Les verbes dplyr évaluent leurs arguments **à l'intérieur du tableau de données**. Le `group_col` de votre fonction est une variable de la fonction, non une colonne : dplyr cherche donc une colonne nommée `group_col` et échoue.

Le correctif tient en un opérateur :

```
rate_by <- function(df, group_col) {
  df |> dplyr::summarise(n = dplyr::n(), .by = {{ group_col }})
}

rate_by(muac, commune)
```

`{{ }}` — l'« embrassement » — signifie *prends ce que l'appelant a écrit et évalue-le dans les données*. C'est la réponse pour presque toute fonction que vous écrirez autour de dplyr.

Pour un nom de colonne arrivant sous forme de **chaîne**, ce que donne un argument de ligne de commande :

```
rate_by_name <- function(df, group_col) {
```

```
  df |> dplyr::summarise(n = dplyr::n(), .by = dplyr::all_of(group_col))
}

rate_by_name(muac, "commune")
```

`all_of()` lève une erreur si la colonne n'existe pas ; `any_of()` la saute en silence. Pour un script dont la sortie est rapportée, `all_of()` est celui qu'il vous faut.

Pour nommer une colonne de sortie à partir d'un argument :

```
count_as <- function(df, group_col, out_name) {
  df |> dplyr::summarise("{out_name}" := dplyr::n(), .by = {{ group_col }})
}
```

La forme `"{...}" :=` est la syntaxe glue à l'intérieur d'un verbe du tidyverse. Elle mérite d'être reconnue ; vous en aurez rarement besoin.

8.5 La fonction d'indicateur

```
indicator_table <- function(muac, gam_mm = 125, sam_mm = 115) {
  clean_register(muac, gam_mm, sam_mm) |>
    dplyr::summarise(
      screened  = dplyr::n(),
      assessed  = sum(assessed, na.rm = TRUE),
      gam_cases = sum(gam, na.rm = TRUE),
      sam_cases = sum(sam, na.rm = TRUE),
      .by = commune
    ) |>
    dplyr::mutate(
      assessment_rate = round(assessed / screened, 3),
      gam_rate = round(gam_cases / assessed, 3),
      sam_rate = round(sam_cases / assessed, 3)
    ) |>
    dplyr::arrange(dplyr::desc(gam_rate))
}
```

Notez ce qu'elle ne fait **pas** : elle ne supprime pas les communes à faible taux d'évaluation. En supprimer une changerait le dénominateur du district et rien dans la sortie ne le dirait. Le signalement revient à l'appelant, qui peut voir le taux.

8.6 Tester la partie qui calcule

Une fois le calcul devenu une fonction prenant un tableau, un test tient en trois lignes :

```
# tests/testthat/test-indicator-table.R
test_that("le denominateur exclut les enfants non mesures", {
  muac <- tibble::tibble(
    child_id = c("a", "b", "c"),
    commune  = "X",
    muac_mm  = c(110L, 130L, NA_integer_),
    oedema   = c("false", "false", NA_character_)
  )

  out <- indicator_table(muac)

  expect_equal(out$screened, 3)
  expect_equal(out$assessed, 2)     # l'enfant non mesure n'est pas au denominateur
```

```
  expect_equal(out$gam_rate, 0.5)
})
```

Ce test encode la décision de la leçon sur le regroupement — quels enfants entrent au dénominateur — sous une forme qui échoue si quelqu'un la change. À écrire pour tout indicateur dont la définition a fait débat.

`usethis::use_testthat()` met en place le répertoire ; `devtools::test()` l'exécute.

8.7 Le script en ligne de commande

```
#!/usr/bin/env Rscript
# scripts/indicator_table.R

suppressPackageStartupMessages({
  library(optparse)
  library(readr)
})

source(here::here("R", "clean_register.R"))
source(here::here("R", "indicator_table.R"))

option_list <- list(
  make_option("--input", type = "character", help = "CSV du registre de depistage."),
  make_option("--output", type = "character", help = "Repertoire des tableaux."),
  make_option("--gam-threshold", type = "integer", default = 125L,
              help = "Seuil de PB en mm pour la MAG [defaut %default].")
)

opt <- parse_args(OptionParser(option_list = option_list))
if (is.null(opt$input) || is.null(opt$output)) {
  stop("--input et --output sont requis.")
}

muac <- read_register(opt$input)
message("Lu ", nrow(muac), " lignes depuis ", opt$input)

table <- indicator_table(muac, gam_mm = opt$`gam-threshold`)

thin <- table$commune[table$assessment_rate < 0.8]
if (length(thin) > 0) {
  warning("Taux d'evaluation sous le plancher : ", paste(thin, collapse = ", "))
}

dir.create(opt$output, recursive = TRUE, showWarnings = FALSE)
destination <- file.path(opt$output, "gam_by_commune.csv")
write_csv(table, destination)
message("Ecrit ", destination, " (", nrow(table), " communes)")
```

```
Rscript scripts/indicator_table.R \
  --input data/raw/muac-screening-artibonite-2024.v1.csv \
  --output outputs/tables
```

`message()` plutôt que `print()` : les messages partent sur la sortie d'erreur, si bien que la narration du script ne se mêle pas à une sortie redirigée. `warning()` plutôt que `message()` pour les communes minces, car c'est une condition qu'un appelant peut vouloir durcir avec `options(warn = 2)`.

8.8 Consigner ce qui a produit la sortie

```
run_metadata <- function(opt) {
  list(
    generated_at  = format(Sys.time(), "%Y-%m-%dT%H:%M:%S%z"),
    input         = opt$input,
    gam_threshold = opt$`gam-threshold`,
    r_version     = as.character(getRversion()),
    dplyr         = as.character(packageVersion("dplyr"))
  )
}

jsonlite::write_json(run_metadata(opt), file.path(opt$output, "run.json"),
                     auto_unbox = TRUE, pretty = TRUE)
```

Six mois plus tard, « quel seuil a produit ce tableau » a une réponse qui ne dépend de la mémoire de personne.

Puis le véritable test, issu de la première leçon :

```
rm -rf outputs/
Rscript scripts/indicator_table.R --input data/raw/muac.csv --output outputs/tables
git status
```

Si cela ne restaure pas les sorties à l'identique, quelque chose n'est pas reproductible.

8.9 Où ce cours s'arrête

Vous savez construire un projet qui se rouvre un an plus tard avec ses versions de paquets consignées, lire n'importe quel export sans le corrompre, conserver les étiquettes qu'un fichier d'enquête transporte, mettre les catégories dans l'ordre qu'exige un rapport, agréger vers un numérateur et un dénominateur issus d'un seul appel, restructurer un groupe répété aplati, et remettre le résultat à quelqu'un d'autre sous forme de script plutôt que de service rendu.

Ce que ce cours n'a délibérément pas enseigné, c'est *quoi* calculer — quel indicateur, quel dénominateur, quel seuil, et comment le défendre. C'est l'objet de *Fondamentaux de l'analyse de données*, et les cours sectoriels du module **Analyses sectorielles** de la feuille de route le poussent plus loin, jusqu'aux classifications que votre cluster vous impose.

Si votre équipe travaille en Python plutôt qu'en R, *Python pour les données de programme* couvre le même terrain dans ce langage — et le tableau des renversements de la leçon 6 est le moyen le plus rapide de passer de l'un à l'autre.

À propos de cette édition

Ce PDF est produit à partir de la même source que la version web de la formation ; les deux ne peuvent donc pas diverger. L'édition en ligne comporte le code exécutable, les jeux de données téléchargeables et les corrections publiées depuis la production de ce fichier.

> Tous les jeux de données cités dans cette formation sont synthétiques ou entièrement anonymisés. Aucun enregistrement ne renvoie à une personne réelle.

Cette formation est publiée en anglais et en français. L'édition française est une adaptation professionnelle reprenant la terminologie employée par l'UNICEF, l'OMS, le PAM, OCHA et les ministères nationaux — et non une traduction littérale.

Consulter et exécuter la formation en ligne sur data-analysis.cassion.dev/fr/cours/r-for-programme-data

Généré le 27 juillet 2026.