CASSION · FORMATION

Jointures et restructuration des données de programme

Listes de ménages et de membres, groupes répétés, admissions et sorties, et agrégats mensuels qui doivent s'aligner sur un dénominateur de population.

Formateur	Pierre Robentz Cassion
Niveau	Intermédiaire
Heures apprenant	14 h
Enseignée en	Python · R
Mise à jour	27 juillet 2026
Secteurs	Santé publique · Réponse d'urgence
Normes et méthodologies	Définitions d'indicateurs de l'UNICEF · Gestion axée sur les résultats (GAR)

Consulter et exécuter la formation en ligne sur data-analysis.cassion.dev/fr/cours/joining-and-reshaping

Ce que vous saurez faire

— Choisir un type de jointure à partir de ce que vous voulez voir vérifié dans le résultat, et prouver ensuite que ce l'était

— Énoncer la granularité de chaque table que vous manipulez, et agréger à cette granularité avant la jointure plutôt qu'après

— Passer délibérément du format long au format large, et défaire le groupe répété aplati qu'exporte une plateforme de collecte

— Rattacher un dénominateur de population issu d'une base administrative, et défendre le dénominateur retenu

— Construire une grille de périodes complète pour qu'un mois jamais rapporté soit visible au lieu d'être absent

— Rapprocher un registre ligne à ligne de l'agrégat remonté, et publier l'écart

Prérequis

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

Sommaire

À propos de cette formation

Presque aucune question de programme ne trouve sa réponse dans un seul fichier. Les enfants sont dans le registre de dépistage et la population dans la projection du recensement. Les ménages sont dans un export et leurs membres dans un autre. Le registre journalier est dans la tablette et le chiffre mensuel dans DHIS2, et les deux ne concordent pas.

Ce cours porte sur leur assemblage. Il est plus court que le cours de nettoyage et plus difficile, car le mode de défaillance diffère — un défaut de nettoyage rend une valeur fausse, et on peut généralement le voir. **Un défaut de jointure rend un nombre de lignes faux, et il n'y a rien à voir** : ni erreur, ni avertissement, ni valeur implausible. Une jointure à gauche qui ajoute 120 lignes et un plusieurs-à-plusieurs qui transforme 70 245 lignes en 3,5 millions produisent l'un comme l'autre une table qui ressemble exactement à une table.

La discipline est donc énoncée d'emblée et répétée dans chaque leçon — **dites ce que la jointure doit faire avant de l'exécuter, et prouvez ensuite qu'elle l'a fait.** Les deux langages vérifient la relation pour vous si vous le demandez, et le demander coûte un argument.

Chaque technique est présentée en Python et en R, sur des jeux de données de la plateforme où le problème est réel — une liste d'élèves qui n'est pas unique sur son propre identifiant, une extraction de type DHIS2 dont la grille complète de douze mois masque quelles formations sanitaires sont restées silencieuses, et une enquête EAH dont l'analyse se fait par personne alors que les données sont par ménage.

Vous devriez avoir suivi *Nettoyage et validation des données*, ou avoir au moins pris l'habitude d'affirmer une clé avant de l'utiliser. Ce cours-là se termine là où celui-ci commence.

Première partie

Des jointures démontrables

CHAPITRE 1

Quatre jointures, et la question à laquelle chacune répond

Leçon 1 sur 8 · 90 min

Interne, à gauche, complète et anti — choisies d'après ce qui doit être vrai du résultat plutôt que par habitude. Plus l'arithmétique des lignes qui explique tous les gonflements que vous rencontrerez.

1.1 Choisissez la jointure d'après la phrase que vous allez écrire

La plupart des gens choisissent une jointure par habitude — la jointure à gauche, parce que c'est celle qui marche d'ordinaire. C'est l'inverse qu'il faut faire. La jointure est une affirmation sur les lignes qui ont leur place dans le résultat, et cette affirmation vient de la phrase que vous comptez publier.

La phrase que vous écrirez	La jointure
« La présence de chaque élève inscrit »	à gauche, registre à gauche
« La présence des élèves ayant à la fois une trace et une ligne de liste »	interne
« Tout des deux côtés, pour ne rien perdre »	complète
« Les élèves marqués présents qui ne sont pas sur la liste »	anti

Lisez-les dans l'autre sens et chaque jointure appelle une défaillance. Une jointure interne **écarte en silence** les lignes non appariées, et ces lignes sont fréquemment le constat. Une jointure à gauche **multiplie en silence** lorsque le côté droit n'est pas unique. Une jointure complète produit une table où une valeur manquante peut vouloir dire deux choses différentes. Et une anti-jointure ne renvoie rien lorsque tout s'est apparié, ce que l'on lit comme « le contrôle est passé » sans vérifier qu'il a bien tourné.

1.2 L'arithmétique, une fois, à la main

Toutes les surprises de ce cours découlent d'une seule règle. Une jointure émet **une ligne pour chaque couple de lignes partageant la clé.**

Si une valeur de clé apparaît `m` fois à gauche et `n` fois à droite, elle contribue `m` × `n` lignes.

Occurrences à gauche	Occurrences à droite	Lignes émises
1	1	1
60	1	60
60	2	120
50	3000	150 000

Trois conséquences découlent directement de ce tableau.

— **Nombre de lignes préservé** seulement si le côté droit est unique sur la clé. C'est la phrase sur laquelle repose toute la leçon.

— **Nombre de lignes multiplié** dans le cas contraire — en silence, sans erreur, par un facteur que personne n'a choisi.

— **Nombre de lignes réduit** uniquement par une jointure interne, ou par une clé qui ne s'apparie pas. Une jointure à gauche ne retire jamais de ligne, donc une jointure à gauche qui rétrécit signale une clé qui a changé sous vos pieds.

La troisième ligne du tableau est un défaut réel du cours précédent — deux élèves présents deux fois sur la liste, soixante jours de classe chacun, et une jointure à gauche qui ajoute 120 lignes à une table de 70 245. La quatrième est ce qui se produit quand on joint sur la mauvaise colonne, ce que nous ferons délibérément dans un instant.

1.3 Le montage

Deux fichiers. Une liste de 1 202 lignes couvrant 1 200 élèves de 24 écoles, et 70 245 marques de présence journalière.

```python
import pandas as pd

roster = pd.read_csv("school-roster-2024.v1.csv")
attendance = pd.read_csv("school-attendance-2024.v1.csv", parse_dates=["attendance_date"])

print(len(roster), roster["student_id"].nunique())
print(len(attendance))
```

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

roster     <- read_csv("school-roster-2024.v1.csv")
attendance <- read_csv("school-attendance-2024.v1.csv")

c(rows = nrow(roster), students = n_distinct(roster$student_id))
nrow(attendance)
```

1 202 lignes et 1 200 élèves. Réglez cela avant de joindre — le cours précédent a montré pourquoi, et la suite de cette leçon suppose que c'est fait.

```python
roster = roster.drop_duplicates(subset=["student_id"], keep="first")
```

```r
roster <- roster |> distinct(student_id, .keep_all = TRUE)
```

Garder la première ligne est une décision, pas un réglage par défaut. Pour ces deux élèves, c'est la mauvaise — l'une des lignes enregistre l'école quittée et l'autre l'école rejointe, et « la première » retient celle que l'export a triée en tête. Dans la vraie vie, cela remonte à la personne qui tient le registre. Ici c'est un provisoire pour que les jointures ci-dessous aient de quoi travailler, et cela a sa place dans le journal de nettoyage.

1.4 Jointure à gauche — garder la table de gauche, y accrocher des colonnes

```python
joined = attendance.merge(roster, on="student_id", how="left", validate="many_to_one")
print(len(attendance), "->", len(joined))
```

```r
joined <- attendance |>
  left_join(roster, by = "student_id", relationship = "many-to-one")
```

```
cat(nrow(attendance), "->", nrow(joined), "\n")
```

70 245 vers 70 245. C'est ce qu'une jointure à gauche est censée faire, et l'argument `validate` / `relationship` est ce qui en fait une garantie plutôt qu'un espoir. Sans lui, le même appel sur la liste non dédoublonnée renvoie 70 365 et ne vous dit rien.

1.5 Jointure interne — et les lignes qu'elle emporte

```
inner = attendance.merge(roster, on="student_id", how="inner")
print(len(inner), "rows;", len(attendance) - len(inner), "attendance rows dropped")
```

```
inner <- attendance |> inner_join(roster, by = "student_id")
cat(nrow(inner), "rows;", nrow(attendance) - nrow(inner), "dropped\n")
```

Ici rien n'est écarté, car chaque ligne de présence a une ligne de liste. C'est inhabituel et cela mérite d'être dit à voix haute quand cela arrive.

Quand ce n'est pas zéro, le nombre importe plus que la jointure. Une jointure interne qui écarte 4 % d'une liste de distribution écarte quatre pour cent des bénéficiaires de quelqu'un, et le rapport dira « 12 000 personnes atteintes » sans la moindre note de bas de page, parce que les lignes qui auraient soulevé la question sont justement celles qui sont parties.

> Utilisez une jointure interne quand vous avez déjà regardé ce qu'elle retire. L'utiliser *pour* les retirer est la manière dont les lignes gênantes sont évacuées sans décision.

1.6 Jointure complète — rien n'est perdu, et vous avez désormais deux sortes de vide

```
full = attendance.merge(roster, on="student_id", how="outer", indicator=True)
print(full["_merge"].value_counts())
```

```
full <- attendance |> full_join(roster, by = "student_id")
```

Le `indicator=True` de pandas ajoute une colonne `_merge` valant `both`, `left_only` ou `right_only`, et c'est l'argument le plus utile de cette leçon. dplyr n'a pas d'équivalent, alors ajoutez-en un.

```
full <- attendance |>
  mutate(in_attendance = TRUE) |>
  full_join(mutate(roster, in_roster = TRUE), by = "student_id") |>
  mutate(
    side = case_when(
      in_attendance & in_roster ~ "both",
      in_attendance             ~ "attendance only",
      TRUE                      ~ "roster only"
    )
  )
```

La raison de s'en donner la peine — après une jointure complète, un `grade` vide signifie soit « cet élève n'a pas de ligne de liste », soit « cet élève a une ligne de liste sans niveau

renseigné ». Ce sont deux constats totalement différents et la jointure les a rendus identiques. La colonne de provenance les maintient séparés.

1.7 Anti-jointure — le contrôle, pas la jointure

Une anti-jointure renvoie les lignes d'un côté sans partenaire de l'autre. Elle est rarement l'analyse et presque toujours le contrôle.

```python
in_roster = set(roster["student_id"])
orphans = attendance[~attendance["student_id"].isin(in_roster)]
never_seen = roster[~roster["student_id"].isin(set(attendance["student_id"]))]

print(len(orphans), "attendance rows with no roster entry")
print(len(never_seen), "enrolled students with no attendance row at all")
```

```r
orphans     <- attendance |> anti_join(roster, by = "student_id")
never_seen  <- roster |> anti_join(attendance, by = "student_id")

c(orphans = nrow(orphans), never_seen = nrow(never_seen))
```

Les deux sens, à chaque fois, et ils veulent dire des choses opposées.

— **De la présence sans ligne de liste** — on enregistre quelqu'un qui n'est pas inscrit. Un problème de données, ou une liste d'inscription périmée.

— **Des lignes de liste sans aucune présence** — des enfants inscrits jamais marqués, ni dans un sens ni dans l'autre. Dans un programme d'éducation ce n'est pas un défaut, c'est le constat : ce sont les enfants qui ne sont jamais venus.

Les deux valent zéro ici. Écrivez le contrôle quand même, car au trimestre suivant ce ne sera plus le cas.

1.8 Joindre sur la mauvaise colonne

Faites-le une fois, délibérément, pour en reconnaître la silhouette le jour où cela arrivera par accident. Les deux fichiers portent `school_id` après la première jointure, et joindre sur lui plutôt que sur `student_id` est une erreur d'un seul caractère.

```python
wrong = attendance.merge(roster, on="school_id", how="left")
print(f"{len(attendance):,} -> {len(wrong):,}")
```

```r
wrong <- attendance |> left_join(roster, by = "school_id")
format(nrow(wrong), big.mark = ",")
```

70 245 devient **3 567 105**. Chaque ligne de présence s'est appariée à tous les élèves de la même école — une cinquantaine — et le taux de présence calculé sur cette table tourne toujours autour de 88 %, parce que multiplier chaque ligne par cinquante laisse la proportion intacte.

C'est là tout le propos. **Le taux survit, les effectifs non.** Tout ce qui est rapporté en nombre d'enfants est multiplié par cinquante, et un taux qui a l'air juste est exactement la raison pour laquelle personne ne vérifie l'effectif.

1.9 La suite

Vous disposez de quatre jointures et de l'arithmétique qui les gouverne. La leçon suivante en fait une habitude qui tourne sans y penser — un tableau de rapprochement produit par chaque jointure, avec les lignes appariées, les deux comptes d'anti-jointure et une assertion qui échoue quand le nombre de lignes bouge.

CHAPITRE 2

Prouver que la jointure a fait ce que vous avez annoncé

Leçon 2 sur 8 · 90 min

Un tableau de rapprochement pour chaque jointure — lignes appariées, les deux comptes d'anti-jointure, le nombre de lignes avant et après — plus la collision de suffixes qui écrase une colonne sans rien dire.

2.1 Le contrôle doit coûter moins cher que le bug

Personne ne saute les contrôles de jointure en croyant les jointures sûres. On les saute parce que vérifier prend quatre commandes, qu'il faut lire la sortie, et que la jointure a « manifestement » fonctionné.

Le contrôle doit donc devenir un seul appel produisant un seul petit tableau. Écrivez-le une fois, placez-le dans le fichier que tous vos scripts importent, et le coût de la vérification passe sous celui du doute.

2.2 Déclarez la relation, toujours

La première ligne de défense est un argument que vous pouviez déjà passer.

```python
joined = attendance.merge(
    roster, on="student_id", how="left", validate="many_to_one",
)
```

```r
joined <- attendance |>
  left_join(roster, by = "student_id", relationship = "many-to-one")
```

Quatre valeurs, et choisir entre elles vous oblige à dire ce que vous croyez.

Ce que vous attendez	pandas `validate=`	dplyr `relationship=`
Une ligne de chaque côté	`one_to_one`	`one-to-one`
Plusieurs à gauche, une à droite	`many_to_one`	`many-to-one`
Une à gauche, plusieurs à droite	`one_to_many`	`one-to-many`
Réellement plusieurs à plusieurs	omettre	`many-to-many`

dplyr avertit d'un plusieurs-à-plusieurs inattendu même sans l'argument ; pandas non. Cette différence a décidé du sort d'un certain nombre de chiffres discrètement faux, et c'est la raison pour laquelle les exemples Python de ce cours passent toujours `validate=`.

Notez la dernière ligne. Passer `many-to-many` explicitement n'est pas une façon de faire taire le contrôle — c'est l'affirmation que le gonflement est voulu, et cela va de pair avec un commentaire disant quelle est la granularité obtenue.

2.3 Le tableau de rapprochement

L'argument de relation vous dit que la forme était bonne. Il ne vous dit pas ce qui n'a pas trouvé de partenaire, et c'est en général la question la plus intéressante.

```python
def reconcile(left, right, on, left_name="left", right_name="right"):
    left_keys = left[on].drop_duplicates()
    right_keys = right[on].drop_duplicates()
    merged = left_keys.merge(right_keys, on=on, how="outer", indicator=True)
    counts = merged["_merge"].value_counts()

    return pd.Series({
        f"{left_name} rows": len(left),
        f"{right_name} rows": len(right),
        "keys matched": int(counts.get("both", 0)),
        f"{left_name} only": int(counts.get("left_only", 0)),
        f"{right_name} only": int(counts.get("right_only", 0)),
        f"{left_name} unmatched %": round(
            100 * counts.get("left_only", 0) / max(len(left_keys), 1), 2
        ),
    })

print(reconcile(attendance, roster, ["student_id"], "attendance", "roster"))
```

```r
reconcile <- function(left, right, on, left_name = "left", right_name = "right") {
  left_keys  <- dplyr::distinct(dplyr::select(left, dplyr::all_of(on)))
  right_keys <- dplyr::distinct(dplyr::select(right, dplyr::all_of(on)))

  tibble::tibble(
    measure = c(paste(left_name, "rows"), paste(right_name, "rows"),
                "keys matched", paste(left_name, "only"), paste(right_name, "only")),
    value = c(
      nrow(left), nrow(right),
      nrow(dplyr::inner_join(left_keys, right_keys, by = on)),
      nrow(dplyr::anti_join(left_keys, right_keys, by = on)),
      nrow(dplyr::anti_join(right_keys, left_keys, by = on))
    )
  )
}

reconcile(attendance, roster, "student_id", "attendance", "roster")
```

Remarquez qu'il rapproche les **clés distinctes**, pas les lignes. Comparer des nombres de lignes de part et d'autre d'une jointure un-à-plusieurs n'apprend rien ; comparer les ensembles de clés dit exactement qui est d'un côté et pas de l'autre.

mesure	valeur
lignes de présence	70 245
lignes de liste	1 200
clés appariées	1 200
présence seulement	0
liste seulement	0

Voilà le tableau à coller dans une cellule au-dessus de chaque jointure. Quatre secondes de lecture, et les deux modes de défaillance silencieux deviennent impossibles à manquer.

2.4 Protégez le nombre de lignes quand vous vouliez le préserver

```python
def join_preserving(left, right, on, how="left", **kwargs):
    before = len(left)
    out = left.merge(right, on=on, how=how, **kwargs)
```

```
    if len(out) != before:
        raise ValueError(f"join changed row count: {before} -> {len(out)}")
    return out
```

```
join_preserving <- function(left, right, by, ...) {
  before <- nrow(left)
  out <- dplyr::left_join(left, right, by = by, ...)
  if (nrow(out) != before) {
    stop(sprintf("join changed row count: %d -> %d", before, nrow(out)))
  }
  out
}
```

C'est strictement plus fort que `validate=`, car cela attrape aussi le cas où la clé est unique des deux côtés mais où la table de *gauche* a perdu des lignes — ce qui arrive dès que quelqu'un remplace `how="left"` par `how="inner"` en déboguant autre chose et oublie de revenir en arrière.

2.5 Le taux de non-appariement est un indicateur, pas une erreur

Une jointure qui n'apparie pas 6 % d'une liste de bénéficiaires vous dit quelque chose sur l'enregistrement, pas sur votre code. Rapportez-le.

```
summary = reconcile(distributions, registration, ["beneficiary_id"])
if summary["left unmatched %"] > 5:
    print(f"WARNING: {summary['left unmatched %']}% of distribution rows "
          "have no registration record")
```

```
unmatched <- nrow(anti_join(distributions, registration, by = "beneficiary_id"))
share <- unmatched / nrow(distributions)
if (share > 0.05) warning(sprintf("%.1f%% of distributions have no registration", 100 *
    share))
```

Suivez ce nombre d'un tour à l'autre et il devient une tendance de qualité des données qui a sa place dans un rapport mensuel — le même geste que la série de constats de validation du cours de nettoyage. Un taux d'appariement qui tombe de 98 % à 91 % entre deux tours signale un processus d'enregistrement qui a changé, et il est bien plus utile de le savoir en mars que de le découvrir lors d'un audit.

2.6 Normalisez la clé avant d'accuser la jointure

Une large part des « la jointure n'a pas marché » vient d'une clé qui ne se compare pas égale.

— **Le type.** Un code de formation sanitaire lu comme entier d'un côté et comme chaîne de l'autre. `1042` n'égale jamais `"1042"`, et `"01042"` lu comme un nombre perd définitivement son zéro initial.

— **Espaces et casse.** `"FAC001 "` et `"fac001"` font trois clés différentes.

— **Dates contre horodatages.** `2024-03-01` et `2024-03-01 00:00:00` se comparent égaux dans certaines bibliothèques et pas dans d'autres ; une colonne de période est plus sûre en chaîne de caractères.

```python
def key_clean(series):
    return series.astype("string").str.strip().str.upper()

for frame in (registers, aggregates):
    frame["facility_id"] = key_clean(frame["facility_id"])
```

```r
key_clean <- function(x) toupper(stringr::str_squish(as.character(x)))

registers  <- registers  |> mutate(facility_id = key_clean(facility_id))
aggregates <- aggregates |> mutate(facility_id = key_clean(facility_id))
```

Faites-le sur les deux côtés dans la même fonction, pour qu'ils ne puissent pas diverger. Deux expressions de nettoyage séparées, c'est le même défaut que deux calculs d'indicateur séparés.

2.7 La collision de suffixes

Les deux tables ont une colonne nommée `updated_at`, ou `district`, ou `notes`. La jointure n'échoue pas, elle renomme.

```python
merged = registers.merge(aggregates, on="facility_id", suffixes=("_register", "_dhis2"))
```

```r
merged <- registers |>
  left_join(aggregates, by = "facility_id", suffix = c("_register", "_dhis2"))
```

Les deux bibliothèques ont un défaut peu utile — `_x` et `_y` en pandas, `.x` et `.y` en dplyr — et trois jointures plus loin, plus personne ne peut dire si `district_x` vient du registre ou de l'agrégat. **Nommez les suffixes d'après la source, à chaque fois.** Cela coûte un argument et fait la différence entre une table traçable et une supposition.

Pire encore, une colonne partagée dont vous ignoriez l'existence, transportée en silence par la jointure puis utilisée. Vérifiez avant de joindre.

```python
overlap = (set(registers.columns) & set(aggregates.columns)) - {"facility_id"}
print("columns in both:", sorted(overlap))
```

```r
setdiff(intersect(names(registers), names(aggregates)), "facility_id")
```

2.8 Des clés aux noms différents

Dites-le explicitement plutôt que de renommer une colonne pour faire marcher la jointure — le renommage survit à la jointure et égare le lecteur suivant.

```python
merged = cases.merge(
    facilities, left_on="site_code", right_on="facility_id", how="left",
    validate="many_to_one",
)
```

```r
merged <- cases |>
  left_join(facilities, by = c("site_code" = "facility_id"),
            relationship = "many-to-one")
```

2.9 La suite

Tous les contrôles vus jusqu'ici supposent que vous savez ce qu'est une ligne de chaque table. C'est dans cette supposition que logent les défaillances restantes — une table de ménages jointe à une table de personnes, ou un agrégat mensuel joint à un registre journalier. La leçon suivante porte sur la désignation de la granularité, et sur l'agrégation à cette granularité avant la jointure plutôt que sur la découverte après coup d'une multiplication.

Deuxième partie

Une ligne par quoi ?

CHAPITRE 3

Une ligne par quoi ?

Leçon 3 sur 8 · 90 min

La granularité d'une table décide du sens de tout nombre qu'on en tire. Ménages contre personnes, épisodes contre patients, et le plusieurs-à-plusieurs qui transforme 2 403 lignes en 13 004 sans que personne l'ait décidé.

3.1 Énoncez la granularité à voix haute

La **granularité** d'une table, c'est ce qu'est une ligne. Un entretien de ménage. Un enfant dépisté un jour donné. Un couple formation-mois-antigène. Un épisode d'admission.

Cela ressemble à une formalité. C'est la phrase la plus utile que vous puissiez écrire au-dessus d'une table, car presque toute défaillance de jointure et presque toute dispute sur un dénominateur sont en réalité deux personnes qui tiennent en tête deux granularités différentes.

```python
# grain: one household interview
wash = pd.read_csv("wash-household-survey-2024.v1.csv")

# grain: one student per school day
attendance = pd.read_csv("school-attendance-2024.v1.csv")

# grain: one facility x month x antigen
vax = pd.read_csv("vaccination-coverage-2024.v1.csv")
```

```r
# grain: one household interview
wash <- read_csv("wash-household-survey-2024.v1.csv")

# grain: one student per school day
attendance <- read_csv("school-attendance-2024.v1.csv")

# grain: one facility x month x antigen
vax <- read_csv("vaccination-coverage-2024.v1.csv")
```

Trois commentaires, et ils préviendront plus de dégâts que toutes les assertions de ce cours. Une jointure n'est sûre que si vous savez nommer la granularité des deux côtés et celle du résultat.

3.2 La granularité décide du sens du nombre

L'enquête EAH couvre 2 403 ménages et 13 004 personnes, soit une taille moyenne de 5,52. Deux phrases parfaitement exactes sortent de ce fichier.

```python
sized = wash[wash["household_size"].notna() & wash["litres_per_person_day"].notna()]

by_household = (sized["litres_per_person_day"] < 15).mean()
by_person = (
    sized.loc[sized["litres_per_person_day"] < 15, "household_size"].sum()
    / sized["household_size"].sum()
```

```
)

print(f"households below 15 L/p/d: {by_household:.1%}")
print(f"people in those households: {by_person:.1%}")
```

```
sized <- wash |> filter(!is.na(household_size), !is.na(litres_per_person_day))

sized |>
  summarise(
    by_household = mean(litres_per_person_day < 15),
    by_person = sum(household_size[litres_per_person_day < 15]) / sum(household_size)
  )
```

13,3 % des ménages, 13,5 % des personnes. Et sur la défécation à l'air libre, 13,4 % des ménages contre 13,9 % des personnes.

Les écarts sont ici faibles, et cela mérite d'être dit franchement — **sur ce fichier le choix déplace à peine le nombre.** Il déplace ce que le nombre *est*. Sphere énonce son standard de quantité d'eau par personne, donc c'est le chiffre par personne qui répond au standard ; un chiffre par ménage répond à « combien de ménages sont concernés », ce dont a besoin un plan de distribution.

Là où l'écart n'est pas faible est précisément là où cela compte le plus. Les grands ménages manquent systématiquement plus d'eau par personne, si bien que dans une enquête où la taille des ménages varie davantage, les deux chiffres se séparent — et rapporter alors le chiffre par ménage face à un standard par personne sous-estime le problème.

Indiquez la granularité dans le libellé. `share_of_households_below_15l` et `share_of_people_below_15l` ne peuvent pas être confondus ; `water_below_minimum` le peut.

3.3 Développer les ménages en personnes

Il arrive qu'il faille réellement une ligne par personne — pour joindre à une structure par âge et sexe, pour alimenter une pondération individuelle, ou pour calculer un dénominateur de population à partir de l'enquête elle-même.

```
people = wash.loc[wash["household_size"].notna()].copy()
people["household_size"] = people["household_size"].astype(int)
people = people.loc[people.index.repeat(people["household_size"])]
people["person_index"] = people.groupby("household_id").cumcount() + 1

print(len(wash), "households ->", len(people), "people")
```

```
people <- wash |>
  filter(!is.na(household_size)) |>
  tidyr::uncount(household_size, .remove = FALSE, .id = "person_index")

cat(nrow(wash), "households ->", nrow(people), "people\n")
```

2 403 lignes deviennent 13 004. Trois choses à retenir de cette opération.

— **C'est une affirmation, pas une transformation.** Chaque personne du ménage est désormais supposée avoir l'accès à l'eau, l'assainissement et l'hygiène du ménage. Pour la quantité d'eau c'est raisonnable ; pour « qui va chercher l'eau » c'est faux.

— **Rien des individus n'est réel.** `person_index` est une position, pas une personne. Ne désagrégez pas dessus, et ne le laissez pas ressembler à un identifiant.

— **Les quarante-six ménages sans taille enregistrée disparaissent.** Dites-le, ou la population développée est discrètement amputée de 46 ménages par rapport à l'enquête dont elle vient.

3.4 Pondérer vaut généralement mieux que développer

Le développement est gourmand en mémoire et facile à mal employer. Lorsque vous n'avez besoin que de statistiques pondérées par la population, pondérez.

```
sized["weight"] = sized["household_size"]

weighted_share = (
    (sized["litres_per_person_day"] < 15) * sized["weight"]
).sum() / sized["weight"].sum()
```

```
sized |>
  summarise(share = weighted.mean(litres_per_person_day < 15, w = household_size))
```

Même réponse, une seule table, et la pondération est visible en tant que colonne — ce qui permet à un relecteur de voir sur quoi vous avez pondéré. Une table développée cache la pondération dans le nombre de lignes.

3.5 Agrégez à la granularité avant de joindre

C'est la règle opérationnelle pour laquelle toute la leçon existe.

Vous avez un fichier de présence journalière et vous voulez un résumé par élève joint à la liste. Le mauvais ordre consiste à joindre d'abord et agréger ensuite — cela fonctionne, mais fait traverser la jointure à 70 245 lignes et rend la relation plusieurs-à-un alors qu'elle n'avait pas à l'être.

```
per_student = (
    attendance.assign(present=attendance["present"].map({"true": True, "false": False}))
    .groupby("student_id")
    .agg(days_marked=("present", "size"),
         days_present=("present", "sum"))
    .reset_index()
)
per_student["attendance_rate"] = per_student["days_present"] / per_student["days_marked"]

# grain: one student. Now the join is one-to-one.
summary = roster.merge(per_student, on="student_id", how="left", validate="one_to_one")
```

```
per_student <- attendance |>
  summarise(
    days_marked  = sum(present %in% c(TRUE, FALSE)),
    days_present = sum(present %in% TRUE),
    .by = student_id
  ) |>
  mutate(attendance_rate = days_present / days_marked)

summary <- roster |>
  left_join(per_student, by = "student_id", relationship = "one-to-one")
```

Deux gains, et c'est le second qui compte vraiment. La jointure est désormais `one_to_one`, donc le contrôle le plus fort possible s'applique. Et l'agrégation est écrite là où l'on voit son dénominateur — `days_marked` compte les jours où une marque existe, ce qui n'est pas la même chose que les jours d'ouverture de l'école, et l'avant-dernière leçon est entièrement consacrée à cette distinction.

3.6 Le plusieurs-à-plusieurs que personne n'a choisi

Un plusieurs-à-plusieurs n'est presque jamais voulu. Il survient quand la clé n'est pas celle que vous croyiez.

```python
# household file: one row per household. member file: one row per person.
# Both carry `community`. Joining on it instead of household_id:
bad = households.merge(members, on="community", how="inner")
```

```r
bad <- households |> inner_join(members, by = "community")
```

Chaque ménage d'une communauté s'apparie désormais à chaque personne de cette communauté. Quatre cents ménages de six personnes, dans une seule communauté, produisent $400 \times 2\,400$ lignes — neuf cent soixante mille — à partir d'un fichier dont la granularité honnête est de 2 400 personnes.

L'indice est toujours le même — un nombre de lignes qui est un produit et non une somme, et un total dont l'ordre de grandeur est suspicieusement rond. `validate="one_to_many"` l'attrape avant que vous ayez à le remarquer.

3.7 Des épisodes, pas des personnes

Une dernière granularité à nommer, car c'est là que dérapent les chiffres des programmes de traitement. Dans un registre PCIMA, une ligne est en général un **épisode d'admission**, pas un enfant. Un enfant réadmis après rechute a deux lignes.

Ce qui implique ceci.

— **La charge de cas** est un décompte d'épisodes. Deux admissions du même enfant sont deux admissions, et les rapporter comme une seule sous-estime le travail accompli.

— **Les enfants atteints** est un décompte d'enfants distincts, plus petit.

— **Le taux de guérison** a des épisodes au numérateur et au dénominateur, et mélanger un numérateur d'épisodes avec un dénominateur d'enfants produit un taux supérieur à 100 % que quelqu'un passera une après-midi à expliquer.

```python
print("episodes:", len(admissions))
print("children:", admissions["child_id"].nunique())
print("readmitted:", (admissions.groupby("child_id").size() > 1).sum())
```

```r
c(episodes = nrow(admissions),
  children = n_distinct(admissions$child_id),
  readmitted = sum(count(admissions, child_id)$n > 1))
```

Publier les deux nombres, avec leurs libellés, met fin à la discussion avant qu'elle commence.

3.8 Inscrivez la granularité dans le nom du fichier

Une dernière petite habitude qui rapporte. Quand vous enregistrez une table intermédiaire, mettez la granularité dans son nom.

```
outputs/tables/attendance_by_student.csv
outputs/tables/attendance_by_school_month.csv
outputs/tables/coverage_by_facility_period_antigen.csv
```

Six mois plus tard, `summary.csv` ne vous dit rien et chacun de ceux-ci vous dit tout. Cela rend aussi une mauvaise jointure évidente au moment où quelqu'un lit deux noms de fichiers côte à côte.

3.9 La suite

La granularité répond à la question de ce qu'est une ligne. La leçon suivante répond à celle de ce qu'est une *colonne* — le groupe répété aplati qu'une plateforme de collecte exporte en `symptom_1`, `symptom_2`, `symptom_3`, et le pivot qui le retransforme en lignes que l'on peut compter.

CHAPITRE 4

Long, large, et le groupe répété entre les deux

Leçon 4 sur 8 · 90 min

Pivoter parce que l'analyse a besoin de la forme, pas parce que l'export est arrivé ainsi. Le groupe répété aplati, le tableau croisé DHIS2, et la valeur de remplissage qui transforme « non rapporté » en zéro.

4.1 Deux formes, et l'usage de chacune

Les mêmes données, deux fois.

Long — une ligne par observation, une colonne par variable. 2 736 lignes de formation sanitaire, période, antigène, doses.

Large — une ligne par unité, une colonne par mesure. 456 lignes de formation sanitaire et période, avec six colonnes d'antigènes.

Aucune n'est correcte dans l'absolu. La règle qui fonctionne vraiment :

> **Le long sert à calculer. Le large sert à lire.**

Regroupez, filtrez, joignez et résumez sur des données longues, car chacun de ces verbes prend un nom de colonne et le format long a une colonne par concept. Pivotez vers le large en toute dernière étape avant qu'un humain le regarde, car une personne qui lit un tableau veut les six antigènes côte à côte.

L'essentiel des ennuis de cette leçon vient de l'inverse — recevoir un export large et l'analyser large, ce qui transforme « calculer la couverture de chaque antigène » d'une ligne en six.

4.2 Le groupe répété aplati

Une plateforme de collecte pose une question répétée — chaque enfant du ménage, chaque symptôme observé, chaque source d'eau utilisée — et l'exporte en colonnes numérotées.

```
household_id, member_1_age, member_1_sex, member_2_age, member_2_sex, member_3_age, ...
```

C'est l'une des formes les plus répandues dans ce secteur et elle est inexploitable telle quelle. Vous ne pouvez pas compter les membres, ni filtrer les enfants de moins de cinq ans, ni joindre à quoi que ce soit, parce que ce sur quoi vous voulez travailler est étalé en colonnes au lieu de descendre en lignes.

```
long = members_wide.melt(
    id_vars="household_id",
    var_name="field",
    value_name="value",
)
long[["slot", "attribute"]] = long["field"].str.extract(r"member_(\d+)_(\w+)")

members = (
    long.dropna(subset=["slot"])
```

```
    .pivot(index=["household_id", "slot"], columns="attribute", values="value")
    .reset_index()
)
```

```
members <- members_wide |>
  tidyr::pivot_longer(
    cols = tidyr::starts_with("member_"),
    names_to = c("slot", "attribute"),
    names_pattern = "member_(\\d+)_(\\w+)",
    values_to = "value"
  ) |>
  tidyr::pivot_wider(names_from = attribute, values_from = value)
```

Deux pivots, pas un. Le premier transforme chaque colonne numérotée en lignes ; le second retransforme les noms d'attributs en colonnes. C'est en essayant de le faire d'un seul geste qu'on se bloque, car le nom de colonne porte deux faits — quel membre et quel attribut — qu'il faut séparer avant de pouvoir utiliser l'un ou l'autre.

`names_pattern` en R et l'extraction par expression régulière en Python opèrent cette séparation. Mettez le motif au point face aux vrais noms de colonnes avant d'écrire quoi que ce soit d'autre ; un `NA` silencieux ici devient un membre qui disparaît.

4.3 Les emplacements vides, et ceux qui veulent dire quelque chose

Un formulaire prévu pour huit membres et un ménage de trois exporte cinq emplacements vides. La plupart ne sont rien. L'un d'eux ne l'est pas.

```
print(members["age"].isna().sum(), "empty slots")

members = members[members["age"].notna() | members["sex"].notna()]
```

```
members |> summarise(empty = sum(is.na(age) & is.na(sex)))

members <- members |> filter(!is.na(age) | !is.na(sex))
```

La distinction — un emplacement dont **tous** les attributs sont manquants est du remplissage et doit partir. Un emplacement dont l'âge manque mais dont le sexe est renseigné est un membre réel avec un âge manquant, et le supprimer retire une personne du ménage. Filtrez sur « toute la ligne est vide », jamais sur une seule colonne.

Confrontez le résultat à quelque chose que le fichier sait déjà.

```
counted = members.groupby("household_id").size().rename("members_found")
check = households.join(counted, on="household_id")
mismatch = check[check["members_found"] != check["household_size"]]
print(len(mismatch), "households where the roster does not match household_size")
```

```
check <- households |>
  left_join(count(members, household_id, name = "members_found"), by = "household_id") |>
  filter(members_found != household_size)
```

La taille de ménage déclarée et le nombre de lignes de membres doivent concorder. Là où ce n'est pas le cas, soit le dépliage a perdu quelqu'un, soit l'entretien.

4.4 Du long au large — le tableau croisé DHIS2

L'extraction vaccinale arrive en long — une ligne par formation sanitaire, période et antigène — et c'est la bonne forme pour calculer. Pour la poser devant un responsable sanitaire de district, pivotez.

```python
wide = vax.pivot_table(
    index=["facility_id", "period"],
    columns="antigen",
    values="doses_administered",
    aggfunc="sum",
).reset_index()

print(len(vax), "->", len(wide))
```

```r
wide <- vax |>
  tidyr::pivot_wider(
    id_cols = c(facility_id, period),
    names_from = antigen,
    values_from = doses_administered
  )

cat(nrow(vax), "->", nrow(wide), "\n")
```

2 736 lignes deviennent 456 — 38 formations sanitaires sur 12 mois — avec six colonnes d'antigènes. L'arithmétique mérite d'être vérifiée à chaque fois : 2 736 divisé par 6 antigènes fait 456, donc rien n'a été agrégé au passage. Quand la division ne tombe pas juste, la table longue avait des doublons sur la clé de pivot, et `pivot_table` les aura silencieusement sommés.

Le `pivot` de pandas lève une erreur sur les doublons ; `pivot_table` les agrège. Prenez `pivot` d'abord, précisément parce que vous voulez l'erreur. Le `pivot_wider` de tidyr avertit et produit des colonnes-listes, ce qui est plus laid et tout aussi informatif.

4.5 La valeur de remplissage qui invente des données

```python
wide = vax.pivot_table(
    index=["facility_id", "period"], columns="antigen",
    values="doses_administered", aggfunc="sum", fill_value=0,
)
```

```r
wide <- vax |>
  tidyr::pivot_wider(names_from = antigen, values_from = doses_administered,
                     values_fill = 0)
```

`fill_value=0` remplit les combinaisons absentes des données longues. Pour des doses administrées c'est souvent juste — une formation sanitaire sans ligne penta3 n'a réellement administré aucune dose de penta3.

Pour un **taux**, c'est toujours faux. Une cellule de couverture manquante signifie « non calculée », et la remplir par zéro publie une formation sanitaire à 0 % de couverture qui n'a simplement pas rapporté. Le cours de nettoyage faisait ce constat à propos du rapportage ; ici il revient par une autre porte, comme un argument de pivot que personne ne perçoit comme une décision.

Le réflexe sûr est de la laisser manquante et de la traiter explicitement.

```python
wide = wide.assign(reported=wide.notna().sum(axis=1))
```

```r
wide <- wide |> mutate(reported = rowSums(!is.na(across(bcg:penta3))))
```

4.6 Le tableau large que vous ne pouvez pas filtrer

Une fois en large, une question comme « quels couples formation-mois ont eu moins de dix doses d'un antigène quelconque » demande six comparaisons reliées par `ou`, et ajouter un septième antigène oblige à toutes les modifier. La même question sur des données longues tient en une ligne.

```python
low = vax[vax["doses_administered"] < 10]
```

```r
low <- vax |> filter(doses_administered < 10)
```

C'est le test qui dit si vous avez pivoté trop tôt. **Si votre opération suivante nomme plus d'une colonne contenant le même type de valeur, revenez au long.**

4.7 Aller-retour, et son contrôle

Toute restructuration digne de confiance est réversible. Affirmez-le une fois, pendant que vous écrivez le code.

```python
back = wide.melt(id_vars=["facility_id", "period"],
                 var_name="antigen", value_name="doses_administered").dropna()

assert len(back) == len(vax)
assert back["doses_administered"].sum() == vax["doses_administered"].sum()
```

```r
back <- wide |>
  tidyr::pivot_longer(-c(facility_id, period),
                      names_to = "antigen", values_to = "doses_administered") |>
  filter(!is.na(doses_administered))

stopifnot(nrow(back) == nrow(vax),
          sum(back$doses_administered) == sum(vax$doses_administered))
```

Nombre de lignes et total. Deux lignes, et elles attrapent un antigène silencieusement perdu, un doublon qui s'est fait sommer, et une valeur de remplissage qui a inventé des lignes.

4.8 La suite

Vous savez désormais donner à une table la forme qu'exige la question. L'unité suivante porte sur les tables qu'il faut faire venir de l'extérieur — la base de population qui fournit un dénominateur de couverture, et le calendrier qui dit quelles périodes auraient dû exister.

Troisième partie

Faire coïncider les pièces

CHAPITRE 5

Rattacher un dénominateur de population

Leçon 5 sur 8 · 90 min

D'où vient un dénominateur, comment le joindre sans apparier sur un nom, et pourquoi les populations de bassins de desserte ne totalisent pas un district. Plus le taux de couverture au-dessus de 100 % et ce qu'il dit réellement.

5.1 Le numérateur est la moitié facile

Compter ce qu'a fait votre programme relève de la comptabilité. Décider par quoi le diviser relève de l'analyse, et c'est là qu'un taux de couverture se gagne ou se perd.

L'extraction vaccinale rend la chose concrète — elle livre `target_population` dans le fichier, un chiffre par formation sanitaire, constant sur les douze mois. C'est commode et c'est aussi l'estimation de quelqu'un, faite à un moment donné, par une méthode donnée, et toute cette leçon consiste à ne pas la prendre pour un fait au seul motif qu'elle est arrivée dans une colonne.

5.2 D'où vient réellement un dénominateur

Quatre sources, à peu près par ordre décroissant de fréquence des disputes.

— **Une projection de recensement.** L'institut national de statistique publie un recensement et un taux de croissance ; toute population de district que vous employez est une projection à partir de l'année du recensement. Dans une grande partie de ce secteur, ce recensement a dix ans ou davantage.

— **Une base administrative de population.** Les dénominateurs du ministère de la santé par bassin de desserte, ce qu'est ici `target_population`. Généralement dérivés de la projection par un coefficient — 3,5 % de la population pour les moins d'un an, 20 % pour les moins de cinq ans, 4 % pour les femmes enceintes.

— **Une cible de programme.** Le nombre de personnes que le programme prévoyait d'atteindre. Dénominateur légitime pour « avons-nous fait ce que nous avions dit », jamais pour « quelle part de la population est couverte ».

— **Une enquête.** La plus défendable et la moins disponible, car une enquête donne une proportion avec un intervalle de confiance plutôt qu'un effectif.

Nommez la source dans la colonne. `target_population_moh_2024` et `target_population_census_projec`t sont deux nombres différents, et dès que deux d'entre eux se trouvent dans le même dossier sans libellé, quelqu'un en fera la moyenne.

5.3 Joignez sur le code, jamais sur le nom

Le cours de nettoyage a défendu cet argument à propos des noms de lieux. Il vaut d'être répété ici parce qu'une base de population est l'endroit où une jointure sur nom fait le plus de dégâts : le dénominateur est la seule colonne où un appariement raté ne ressemble pas à un appariement raté, il ressemble à une formation sanitaire sans couverture.

```python
population = pd.read_csv("admin-population-2024.csv")   # admin2_pcode, year, pop_total,
    pop_under1

vax["admin2_pcode"] = vax["admin2_pcode"].astype("string").str.strip().str.upper()

with_denominator = vax.merge(
    population.query("year == 2024"),
    on="admin2_pcode", how="left", validate="many_to_one",
)

missing = with_denominator["pop_under1"].isna().sum()
assert missing == 0, f"{missing} rows have no population figure"
```

```r
population <- read_csv("admin-population-2024.csv")

with_denominator <- vax |>
  mutate(admin2_pcode = toupper(stringr::str_squish(admin2_pcode))) |>
  left_join(filter(population, year == 2024),
            by = "admin2_pcode", relationship = "many-to-one")

stopifnot(!any(is.na(with_denominator$pop_under1)))
```

L'assertion est la ligne importante. Une jointure à gauche qui n'apparie pas laisse un dénominateur manquant ; un dénominateur manquant rend un taux de couverture manquant ; et un taux de couverture manquant est exclu d'une moyenne, si bien que la moyenne du district devient discrètement la moyenne des seuls endroits qui se sont appariés.

5.4 La projection, et l'année que personne n'énonce

Une projection de recensement, c'est une population de base et un taux de croissance. Calculez-la au grand jour.

```python
GROWTH = 0.024   # national annual growth rate, published with the census
CENSUS_YEAR = 2015

population["pop_2024"] = population["pop_census"] * (1 + GROWTH) ** (2024 - CENSUS_YEAR)
```

```r
GROWTH <- 0.024
CENSUS_YEAR <- 2015

population <- population |>
  mutate(pop_2024 = pop_census * (1 + GROWTH)^(2024 - CENSUS_YEAR))
```

Neuf ans à 2,4 % font un facteur de 1,24. **Un quart de votre dénominateur est une hypothèse**, et elle se compose : deux districts projetés depuis le même recensement au même taux national conserveront exactement leurs tailles relatives, ce qui est précisément ce qui ne se produit pas là où il y a eu des déplacements.

Aussi, lorsqu'un taux de couverture est contesté, la projection est en général la réponse honnête. Trois habitudes qui la maintiennent défendable.

— Écrivez l'année du recensement, le taux de croissance et sa source à côté du nombre.

— Utilisez la **même** projection pour tous les indicateurs d'un rapport. Deux indicateurs sur deux projections ne sont pas comparables, et personne ne le remarquera.

— Là où des déplacements ont eu lieu, dites que la projection n'en tient pas compte, et donnez le sens de l'erreur au lieu de prétendre la corriger.

5.5 Les bassins de desserte ne s'additionnent pas

Le bassin d'une formation sanitaire est la population qu'elle est *censée* desservir. Additionnez les bassins de toutes les formations d'un district et vous n'obtiendrez pas le district.

```python
by_facility = vax.query("antigen == 'penta3' and period == '2024-01-01'")
print("sum of facility targets:", by_facility["target_population"].sum())
```

```r
vax |>
  filter(antigen == "penta3", period == "2024-01-01") |>
  summarise(total = sum(target_population))
```

Ici cette somme vaut 11 774 pour le mois. Ce n'est un dénominateur de district utilisable que si les bassins partitionnent le district — sans chevauchement et sans trou — et ce n'est presque jamais le cas. Deux formations de part et d'autre d'une ville comptent toutes deux la ville. Une population située entre deux bassins est comptée deux fois ou pas du tout.

La conséquence est précise et fréquente : **la couverture par formation sanitaire n'est pas fiable et la couverture au niveau du district l'est.** Les gens traversent les limites de bassin pour se faire vacciner, si bien que le numérateur d'une formation inclut des enfants du dénominateur de sa voisine. Agrégez au niveau où la traversée se produit à l'intérieur de l'unité, et le bruit s'annule.

```python
district = (
    vax.query("antigen == 'penta3'")
    .groupby("period")
    .agg(doses=("doses_administered", "sum"),
         target=("target_population", "sum"))
)
district["coverage"] = district["doses"] / district["target"]
```

```r
district <- vax |>
  filter(antigen == "penta3") |>
  summarise(doses = sum(doses_administered),
            target = sum(target_population), .by = period) |>
  mutate(coverage = doses / target)
```

5.6 Une couverture supérieure à 100 %

Cela arrive constamment et ce n'est jamais un programme qui dépasse sa population. Quatre causes, dans l'ordre où il vaut la peine de les vérifier.

1. **Le dénominateur est trop petit.** Une projection périmée, ou un coefficient de bassin qui suppose moins de moins d'un an qu'il n'y en a.
2. **Le numérateur inclut des gens de l'extérieur.** Traversée de limites, ou une campagne qui a attiré des gens d'un district voisin.
3. **La granularité est fausse.** Des doses comptées là où on voulait des enfants — un enfant recevant trois doses d'un antigène à trois doses reste un enfant.

4. **Une jointure a multiplié quelque chose.** Tout ce qui est dans la leçon 1.

```python
over = district[district["coverage"] > 1]
print(over)
```

```r
district |> filter(coverage > 1)
```

Rapportez-le plutôt que de l'écrêter. Un taux de couverture plafonné à 100 % a jeté la preuve que le dénominateur est faux, et ce dénominateur faux est le constat.

5.7 Taux pour mille, et le piège de l'annualisation

```python
district["per_1000"] = 1000 * district["doses"] / district["target"]
```

```r
district <- district |> mutate(per_1000 = 1000 * doses / target)
```

Deux pièges logent au même endroit. D'abord, une couverture **mensuelle** n'est pas une couverture annuelle, et la multiplier par douze suppose que chaque mois a rapporté — ce que le cours de nettoyage a montré faux en août et septembre ici. Ensuite, un taux pour mille exige que sa période figure dans le libellé : `doses_per_1000_under1_per_month` est sans ambiguïté, `rate` ne l'est pas.

5.8 Écrivez le dénominateur

Tout indicateur que vous publiez devrait porter une fiche de référence.

Champ	Valeur
Indicateur	Couverture penta3, district
Numérateur	Doses de penta3 administrées, formations ayant rapporté
Dénominateur	Nourrissons survivants, bassins du ministère sommés au district
Source	Estimations de population du ministère 2024, projetées du recensement 2015 à 2,4 %
Désagrégation	Mois, type de formation sanitaire
Réserve	Taux de complétude de 29 % en août ; couverture calculée sur les seules formations ayant rapporté

Ce tableau prend deux minutes et règle la question définitivement. C'est le même geste que le fichier de définitions que le cours de fondamentaux écrit à côté d'une table de résultats, et c'est ce que le cours *Conception d'indicateurs et cadre logique* développera plus tard en fiche de référence complète.

5.9 La suite

Le dénominateur répond à « sur combien de personnes ». La leçon suivante répond à « sur combien de périodes » — construire un calendrier complet pour qu'une formation sanitaire n'ayant jamais rapporté en août apparaisse comme un trou plutôt que de ne pas apparaître du tout.

CHAPITRE 6

Les lignes qui n'ont jamais été écrites

Leçon 6 sur 8 · 90 min

Une ligne absente n'a aucune valeur à avoir manquante. Construire la grille que les données auraient dû remplir, retrouver où sont passées 1 755 lignes de présence, et comprendre pourquoi une grille complète ne prouve pas que tout le monde a rapporté.

6.1 Une ligne absente n'a aucune valeur à avoir manquante

Tous les contrôles du cours de nettoyage regardaient des valeurs — cases vides, sentinelles, nombres implausibles. Tous partagent une hypothèse : que la ligne est là.

La défaillance traitée ici est d'une autre nature. Une école fermée n'a aucune ligne de présence. Une formation sanitaire qui n'a pas rapporté n'a aucune ligne dans l'extraction. Un mois sans distribution n'a rien du tout. Il n'y a aucune case vide à compter, aucun `NA` à détecter, et tous les résumés que vous calculez portent sur les périodes qui se trouvent avoir été enregistrées.

Le remède a toujours la même forme — **construire la grille des périodes qui devraient exister, y joindre les données, et laisser les trous apparaître sous forme de lignes.**

6.2 Construire la grille

```python
students = roster["student_id"].drop_duplicates()
school_days = attendance["attendance_date"].drop_duplicates()

grid = pd.MultiIndex.from_product(
    [students, school_days], names=["student_id", "attendance_date"]
).to_frame(index=False)

complete = grid.merge(attendance, on=["student_id", "attendance_date"], how="left")

print(f"grid {len(grid):,}, actual {len(attendance):,}, "
      f"missing {len(grid) - len(attendance):,}")
```

```r
grid <- tidyr::expand_grid(
  student_id = unique(roster$student_id),
  attendance_date = unique(attendance$attendance_date)
)

complete <- grid |> left_join(attendance, by = c("student_id", "attendance_date"))

cat(sprintf("grid %d, actual %d, missing %d\n",
            nrow(grid), nrow(attendance), nrow(grid) - nrow(attendance)))
```

1 200 élèves par 60 jours de classe font une grille de 72 000. Le fichier en contient 70 245. **1 755 lignes n'ont jamais été écrites.**

dplyr offre un chemin plus court lorsque la trame est déjà les données.

```r
complete <- attendance |>
```

```
  tidyr::complete(student_id, attendance_date)
```

`complete()` remplit le produit cartésien des valeurs qu'il trouve, ce qui est juste quand toutes les combinaisons devraient exister et faux quand un élève a rejoint l'école en cours de trimestre — il n'inventera pas de jours pour un élève que le fichier ne mentionne pas avant mars. Construisez la grille explicitement quand l'univers est défini hors des données, ce qui est le cas habituel.

6.3 Où sont passées les 1 755

```
missing = complete[complete["present"].isna()]
by_school = (
    missing.merge(roster[["student_id", "school_id"]], on="student_id")
    .groupby("school_id")
    .agg(missing_rows=("present", "size"),
         days=("attendance_date", "nunique"))
)
print(by_school)
```

```
complete |>
  filter(is.na(present)) |>
  left_join(select(roster, student_id, school_id), by = "student_id") |>
  summarise(missing_rows = n(), days = n_distinct(attendance_date), .by = school_id)
```

school_id	lignes manquantes	jours
SCH07	915	15
SCH18	840	15

Deux écoles, quinze jours chacune, et $61 \times 15 + 56 \times 15 = 1\,755$. **La grille rend compte de chacune des lignes manquantes**, et sous une forme qui nomme la cause : deux écoles ont été fermées les mêmes quinze jours de mars.

C'est le bénéfice. Avant la grille, les lignes manquantes étaient invisibles. Après, ce sont deux écoles et une plage de dates — une grève, une inondation, une période d'examens, quelque chose qu'un collègue peut confirmer en un coup de téléphone.

6.4 Absent n'est pas absent

Vient maintenant la décision, et c'est toute la raison d'être de cette leçon.

```
naive = complete["present"].fillna(False)
print("attendance rate treating gaps as absences:", naive.mean())
```

```
complete |> summarise(rate = mean(coalesce(present, FALSE)))
```

Remplir les trous par « absent » fait ressembler SCH07 et SCH18 à une urgence d'abandon scolaire, puisqu'un quart de leur trimestre est désormais enregistré comme tous les enfants manquants tous les jours. Le traitement correct est l'inverse : ces jours n'étaient **pas des jours de classe pour ces écoles**, et ils n'ont leur place ni au numérateur ni au dénominateur.

```python
open_days = attendance.merge(roster[["student_id", "school_id"]], on="student_id")
school_calendar = open_days[["school_id", "attendance_date"]].drop_duplicates()

grid = (
    roster[["student_id", "school_id"]]
    .merge(school_calendar, on="school_id")
)
print(len(grid), "student-days the schools were actually open")
```

```r
school_calendar <- attendance |>
  left_join(select(roster, student_id, school_id), by = "student_id") |>
  distinct(school_id, attendance_date)

grid <- roster |>
  select(student_id, school_id) |>
  left_join(school_calendar, by = "school_id", relationship = "many-to-many")
```

Construisez la grille par école, pas par district. L'univers des périodes est une propriété de l'unité de rapportage, et supposer un calendrier unique et partagé est la manière dont une fermeture devient une absence.

> Décider si un trou est un zéro, une valeur manquante ou une période qui ne devrait pas exister n'est pas une question technique. C'est l'analyse, et cela a sa place dans le journal avec son motif.

6.5 Une grille complète ne prouve pas que tout le monde a rapporté

L'extraction vaccinale est le contre-exemple, et il compte parce qu'il ressemble au bon cas.

```python
expected = (
    vax["facility_id"].nunique()
    * vax["period"].nunique()
    * vax["antigen"].nunique()
)
print(expected, "expected rows;", len(vax), "actual")
```

```r
c(expected = n_distinct(vax$facility_id) * n_distinct(vax$period) * n_distinct(vax$antigen
    ),
  actual = nrow(vax))
```

$38 \times 12 \times 6 = 2\,736$, et le fichier compte 2 736 lignes. La grille est parfaite. Et 642 de ces lignes portent `report_submitted` à `false` et zéro dose.

Le contrôle de grille passe donc, et les données restent pleines de trous — ce sont simplement des trous entourés d'une ligne. **Deux contrôles distincts, et il vous faut les deux :** chaque ligne attendue est-elle présente, et chaque ligne présente contient-elle un rapport.

```python
coverage = (
    vax[vax["report_submitted"]]
    .groupby(["period", "antigen"])
    .agg(doses=("doses_administered", "sum"), target=("target_population", "sum"))
)
reporting = vax.groupby(["period", "antigen"])["report_submitted"].mean()
```

```
vax |>
  summarise(
    reporting_rate = mean(report_submitted),
    doses  = sum(doses_administered[report_submitted]),
    target = sum(target_population[report_submitted]),
    .by = c(period, antigen)
  )
```

6.6 Des clés de période qui se trient

Un petit point mécanique qui cause des ennuis disproportionnés.

```
vax["period"] = pd.to_datetime(vax["period"])
vax["month"] = vax["period"].dt.strftime("%Y-%m")     # 2024-08, sorts correctly
```

```
vax <- vax |> mutate(month = format(period, "%Y-%m"))
```

Employez des clés de période ISO — `2024-08`, `2024-Q3`, `2024-W32` — partout où une période sert de clé. `Aug`, `August` et `08/2024` se trient mal, et `08/2024` est ambigu entre deux conventions toutes deux d'usage quotidien dans ce secteur.

Le mois doit porter son année. Une grille de douze mois indexée sur `month` seul fusionne silencieusement août 2023 et août 2024 dès que deux années de données se retrouvent dans le même dossier.

6.7 Aligner un registre journalier sur un agrégat mensuel

La dernière forme de cette leçon, et celle sur laquelle la suivante s'appuie. Pour comparer un registre ligne à ligne à une remontée mensuelle, agrégez le registre à la granularité de l'agrégat — jamais l'inverse.

```
monthly = (
    attendance.assign(month=attendance["attendance_date"].dt.strftime("%Y-%m"))
    .merge(roster[["student_id", "school_id"]], on="student_id")
    .groupby(["school_id", "month"])
    .agg(marks=("present", "size"),
         present=("present", "sum"))
    .reset_index()
)
```

```
monthly <- attendance |>
  mutate(month = format(attendance_date, "%Y-%m")) |>
  left_join(select(roster, student_id, school_id), by = "student_id") |>
  summarise(marks = n(), present = sum(present %in% TRUE), .by = c(school_id, month))
```

Deux choses à remarquer, car ce sont deux choix. Le mois auquel appartient un enregistrement vient de la **date de l'événement**, non de la date de soumission du formulaire — une remontée tardive appartient toujours au mois qu'elle décrit. Et le dernier mois du fichier est très souvent partiel, si bien qu'un graphique de tendance qui l'inclut paraît toujours en baisse.

```
last = monthly["month"].max()
print(f"excluding partial period {last}")
```

```
monthly = monthly[monthly["month"] < last]
```

```
monthly <- monthly |> filter(month < max(month))
```

Écartez-le ou signalez-le, mais ne le tracez jamais en silence à côté de mois complets.

6.8 La suite

Vous disposez maintenant d'un registre agrégé à la même granularité que la remontée mensuelle. La leçon suivante met les deux côte à côte, cherche pourquoi ils divergent, et transforme l'écart en un tableau publiable plutôt qu'en une discussion à gagner.

Quatrième partie

La table que vous analysez

CHAPITRE 7

Quand le registre et le rapport divergent

Leçon 7 sur 8 · 90 min

Recalculer l'indicateur depuis le registre ligne à ligne, le poser à côté de ce qui a été remonté, et décomposer l'écart par unité de rapportage au lieu de le noyer dans une moyenne.

7.1 Deux chiffres pour la même chose

Quelque part se trouve un registre ligne à ligne — marques de présence journalières, cahier de dépistage, registre de patients. Ailleurs se trouve un chiffre mensuel qui en a été remonté. Les deux divergent, et cet écart est le nombre le plus instructif de ce cours.

Il est instructif parce qu'**il est borné**. Contrairement à la plupart des questions de qualité des données, celle-ci a une bonne réponse rangée dans un tiroir : le registre est la source, le rapport est une affirmation sur le registre, et la différence entre les deux est une quantité mesurable et non une opinion.

Les audits bailleurs appellent leur rapport le **facteur de vérification** — recompter les documents sources, diviser le recomptage par ce qui a été rapporté. Un facteur de vérification de 1,0 signifie que le rapportage est exact. Toute autre valeur est un nombre qui a une cause.

7.2 Recalculer depuis le registre

Agrégez le registre à la granularité de rapportage, exactement comme dans la leçon précédente.

```python
attendance["marked"] = attendance["present"].isin(["true", "false"])
attendance["is_present"] = attendance["present"] == "true"

from_register = (
    attendance.merge(roster[["student_id", "school_id"]], on="student_id",
                     how="left", validate="many_to_one")
    .groupby("school_id")
    .agg(marks=("marked", "sum"), present=("is_present", "sum"))
)
from_register["rate"] = from_register["present"] / from_register["marks"]
```

```r
from_register <- attendance |>
  left_join(select(roster, student_id, school_id), by = "student_id",
            relationship = "many-to-one") |>
  summarise(
    marks   = sum(present %in% c("true", "false")),
    present = sum(present == "true", na.rm = TRUE),
    .by = school_id
  ) |>
  mutate(rate = present / marks)
```

Notez ce que compte `marked`. Une marque existe là où la valeur vaut `true` ou `false`. Les valeurs `Y`, `N` et vide ne sont **pas** des marques selon cette définition — et cette définition est une décision, ce qui fait toute la section suivante.

7.3 La décision de codage qui déplace un dénominateur

Une école a enregistré la présence avec `Y` et `N` au lieu de `true` et `false`. Recalculez des deux façons.

```python
mapping = {"true": True, "false": False, "Y": True, "N": False}
attendance["is_present_mapped"] = attendance["present"].map(mapping)

both_ways = (
    attendance.merge(roster[["student_id", "school_id"]], on="student_id")
    .groupby("school_id")
    .agg(
        marks_strict=("is_present", lambda s: s.notna().sum()),
        rate_strict=("is_present", "mean"),
        marks_mapped=("is_present_mapped", lambda s: s.notna().sum()),
        rate_mapped=("is_present_mapped", "mean"),
    )
)
print(both_ways.loc["SCH09"])
```

```r
both_ways <- attendance |>
  left_join(select(roster, student_id, school_id), by = "student_id") |>
  mutate(mapped = dplyr::recode(present, "Y" = "true", "N" = "false")) |>
  summarise(
    marks_strict = sum(present %in% c("true", "false")),
    rate_strict  = mean(present[present %in% c("true", "false")] == "true"),
    marks_mapped = sum(mapped %in% c("true", "false")),
    rate_mapped  = mean(mapped[mapped %in% c("true", "false")] == "true"),
    .by = school_id
  )
```

SCH09	Strict	Y/N recodés
Marques au dénominateur	2 015	2 865
Taux de présence	86,05 %	85,72 %

Lisez les deux colonnes ensemble, car la leçon est dans le contraste. **Le dénominateur croît de 42 %. Le taux bouge d'un tiers de point.**

C'est la forme la plus courante de ce problème et la raison pour laquelle il franchit si aisément les relectures. Quiconque contrôle le *taux* conclut que le codage n'avait pas d'importance. Quiconque rapporte ensuite « nombre de journées de présence enregistrées » se trompe de 850. Un défaut qui touche à peine un rapport peut être énorme sur un effectif, et le rapportage de programme est plein d'effectifs.

Sur l'ensemble du fichier, la même opération fait passer le taux du district de 88,47 % à 88,42 % et le dénominateur de 69 101 à 69 974.

7.4 Mettez les deux sources dans une seule table

Ne comparez jamais deux chiffres en regardant deux affichages. Joignez-les, avec l'unité de rapportage et la période pour clé, et calculez l'écart comme une colonne.

```python
comparison = (
    from_register.rename(columns={"present": "register_present"})
    .join(reported.rename(columns={"present": "reported_present"}), how="outer")
)
```

```
comparison["difference"] = comparison["register_present"] - comparison["reported_present"]
comparison["verification_factor"] = (
    comparison["register_present"] / comparison["reported_present"]
)
print(comparison.sort_values("difference").head(10))
```

```
comparison <- from_register |>
  full_join(reported, by = c("school_id", "month"),
            suffix = c("_register", "_reported")) |>
  mutate(
    difference = present_register - present_reported,
    verification_factor = present_register / present_reported
  ) |>
  arrange(difference)
```

Une jointure **complète**, délibérément. Une unité présente dans le registre et absente du rapport a cessé de rapporter ; une unité présente dans le rapport sans rien dans le registre remonte des chiffres que personne ne peut retrouver. Les deux sont des constats et une jointure interne supprime les deux.

7.5 Décomposez, ne moyennez pas

L'erreur la plus fréquente ici est de calculer un facteur de vérification unique pour tout le district. Un facteur de 1,02 pour le district peut être vingt écoles à 1,00 et une à 1,40, et la moyenne a masqué la seule chose sur laquelle agir.

```
print(comparison["verification_factor"].describe())
print(comparison[comparison["verification_factor"].sub(1).abs() > 0.05])
```

```
comparison |>
  summarise(across(verification_factor, list(min = min, median = median, max = max)))

comparison |> filter(abs(verification_factor - 1) > 0.05)
```

Rapportez la distribution et les valeurs extrêmes, jamais la moyenne seule. Une bande de tolérance — tout ce qui sort de 0,95 à 1,05 est examiné — transforme ceci d'un nombre en une liste de travail, ce dont un superviseur peut effectivement se servir.

7.6 Pourquoi deux sources divergent

Quatre causes, appelant des réponses totalement différentes. Parcourez-les dans cet ordre.

— **Des définitions différentes.** Le registre compte des marques ; le rapport compte des enfants inscrits. De loin la cause la plus fréquente, la moins souvent soupçonnée, et ce n'est pas du tout un problème de qualité des données — ce sont deux indicateurs sous un seul nom.

— **Des périodes différentes.** Le rapport couvre un mois calendaire, le registre a été extrait le 28. Vérifiez les bornes avant toute chose.

— **Des couvertures différentes.** Le rapport inclut une école que l'extraction du registre excluait, ou l'inverse. L'anti-jointure de la leçon 1 y répond en une ligne.

— **La transcription.** Quelqu'un a additionné la colonne à la main. C'est la cause que tout le monde suppose et la moins fréquente des quatre.

Seule la dernière est une erreur au sens ordinaire. Les trois premières sont des problèmes de rapprochement, et les rapporter comme des « problèmes de qualité des données » fait porter à un superviseur la responsabilité de ce qui a été décidé dans une réunion de conception de formulaire.

7.7 Rapportez l'écart, ne le faites pas disparaître

La tentation est d'ajuster le chiffre du registre jusqu'à ce qu'il corresponde au rapport, ou de ne publier que celui qui a la meilleure allure. Publiez les deux, avec l'écart.

```python
summary = pd.DataFrame({
    "source": ["Line-level register", "Reported monthly returns"],
    "attendance days recorded": [69101, 68240],
    "attendance rate": ["88.5%", "89.1%"],
})
```

```r
summary <- tibble::tibble(
  source = c("Line-level register", "Reported monthly returns"),
  days   = c(69101, 68240),
  rate   = c("88.5%", "89.1%")
)
```

Deux lignes et une phrase nommant le principal contributeur à l'écart. C'est le matériau d'une évaluation de la qualité des données, et le cours *Évaluation de la qualité des données* — le suivant et le dernier de ce module — en fait une routine avec une stratégie d'échantillonnage et un plan d'action corrective.

> Un écart que vous rapportez est un constat. Un écart que vous refermez en silence est un chiffre que personne ne peut reproduire, vous compris, dans six mois.

7.8 La suite

Toutes les pièces sont en place — des jointures démontrables, une granularité énoncée, la bonne forme, un dénominateur, un calendrier complet et une source rapprochée. La dernière leçon les assemble en une table d'analyse unique : une ligne par unité d'analyse, chaque colonne traçable jusqu'à sa provenance, construite par un script qui va des fichiers bruts à la table finale en une seule commande.

CHAPITRE 8

Une seule table, assemblée à dessein

Leçon 8 sur 8 · 75 min

Partir d'une ossature des unités sur lesquelles vous devez rapporter, attacher une mesure à la fois, affirmer à chaque raccord, et porter des colonnes de provenance pour qu'un relecteur puisse remonter tout chiffre jusqu'à son fichier d'origine.

8.1 La table dont tout le reste se calcule

À l'issue d'un assemblage, vous voulez exactement une table : **une ligne par unité d'analyse, une colonne par mesure, et rien d'autre.** Chaque graphique, chaque résumé et chaque chiffre du rapport en sortent.

Cette discipline compte parce que l'alternative est ce qui se produit d'ordinaire — un notebook de onze tableaux nommés `df`, `df2`, `merged`, `merged_final` et `merged_final_v2`, où la table dont un graphique a été tiré est celle qui se trouvait en mémoire à ce moment-là. Personne ne peut reproduire cela, y compris la personne qui l'a écrit.

8.2 Décidez l'unité d'analyse avant d'écrire une ligne

L'unité d'analyse est ce sur quoi votre rapport formule des affirmations. Écrivez-la d'abord, car elle détermine toutes les jointures qui suivent.

```
# unit of analysis: one school x month
# rows: 24 schools x 3 months = 72
```


```
# unit of analysis: one school x month
# rows: 24 schools x 3 months = 72
```


Si vous ne pouvez pas l'énoncer en une ligne, l'analyse n'est pas prête à être assemblée. Et si deux chiffres de votre rapport ont des unités différentes — l'un par école, l'autre par enfant — ils appartiennent à deux tables, pas à une.

8.3 Construisez d'abord l'ossature

L'ossature est l'ensemble complet des unités sur lesquelles vous devez rapporter, bâti à partir de l'autorité et non des données. Chaque mesure y est ensuite attachée.

```
schools = roster["school_id"].drop_duplicates()
months = pd.Series(["2024-02", "2024-03", "2024-04"], name="month")

analysis = (
    pd.MultiIndex.from_product([schools, months], names=["school_id", "month"])
    .to_frame(index=False)
)
print(len(analysis), "rows in the spine")
```

```r
analysis <- tidyr::expand_grid(
  school_id = unique(roster$school_id),
  month = c("2024-02", "2024-03", "2024-04")
)
```

L'ossature d'abord est l'habitude la plus utile de cette leçon. Un couple école-mois sans données existe désormais comme une ligne aux mesures manquantes, ce qui est visible, au lieu de ne pas exister, ce qui ne l'est pas. C'est le même argument que la grille de périodes, appliqué à l'assemblage entier.

8.4 Attachez une mesure à la fois, et affirmez à chaque raccord

```python
def attach(base, addition, on, name):
    before = len(base)
    out = base.merge(addition, on=on, how="left", validate="one_to_one")
    if len(out) != before:
        raise ValueError(f"{name}: row count changed {before} -> {len(out)}")
    filled = out[addition.columns.difference(on)].notna().any(axis=1).mean()
    print(f"{name}: attached, {filled:.1%} of spine rows matched")
    return out

analysis = attach(analysis, attendance_by_school_month, ["school_id", "month"], "
    attendance")
analysis = attach(analysis, enrolment_by_school, ["school_id"], "enrolment")
analysis = attach(analysis, feeding_by_school, ["school_id"], "feeding programme")
```

```r
attach_measure <- function(base, addition, by, name) {
  before <- nrow(base)
  out <- dplyr::left_join(base, addition, by = by, relationship = "one-to-one")
  if (nrow(out) != before) {
    stop(sprintf("%s: row count changed %d -> %d", name, before, nrow(out)))
  }
  message(sprintf("%s: attached", name))
  out
}

analysis <- analysis |>
  attach_measure(attendance_by_school_month, c("school_id", "month"), "attendance") |>
  attach_measure(enrolment_by_school, "school_id", "enrolment")
```

Deux propriétés qui valent d'être acquises. Le nombre de lignes ne peut pas changer, donc un gonflement est impossible par construction. Et le **taux d'appariement est affiché à chaque attachement** — une table d'inscriptions appariée à 100 % de l'ossature et une table de présence appariée à 92 % vous disent immédiatement où sont les trous, avant qu'aucun chiffre ne soit calculé.

`attach()` a l'air d'un cérémonial jusqu'à la première fois où il se déclenche. Il se déclenchera.

8.5 Portez la provenance

Ajoutez des colonnes disant d'où viennent les choses et sur quoi elles reposent.

```python
analysis["source_register"] = "school-attendance-2024.v1.csv"
analysis["denominator_source"] = "roster 2024, enrolled students"
analysis["marks_definition"] = "true/false only; Y/N excluded"
analysis["extracted_on"] = "2026-07-27"
```

```
analysis <- analysis |>
  mutate(
    source_register    = "school-attendance-2024.v1.csv",
    denominator_source = "roster 2024, enrolled students",
    marks_definition   = "true/false only; Y/N excluded",
    extracted_on       = "2026-07-27"
  )
```

Cela paraît redondant tant que chaque ligne porte la même valeur. Cela cesse de l'être dès que deux extractions sont concaténées, qu'un deuxième district arrive, ou que quelqu'un ouvre le CSV un an plus tard sans savoir de quelle version du registre il provient. **Les colonnes constantes sont bon marché ; les nombres sans étiquette ne le sont pas.**

8.6 Affirmez ce qui doit être vrai de la table finie

```
assert analysis.duplicated(subset=["school_id", "month"]).sum() == 0
assert analysis["attendance_rate"].between(0, 1).all()
assert (analysis["days_present"] <= analysis["days_marked"]).all()
assert len(analysis) == 24 * 3

print(f"analysis table: {len(analysis)} rows, "
      f"{analysis['attendance_rate'].isna().sum()} without an attendance rate")
```

```
stopifnot(
  !any(duplicated(analysis[c("school_id", "month")])),
  all(dplyr::between(analysis$attendance_rate, 0, 1), na.rm = TRUE),
  all(analysis$days_present <= analysis$days_marked, na.rm = TRUE),
  nrow(analysis) == 24 * 3
)
```

Quatre assertions, correspondant chacune à une leçon de ce cours — la granularité est unique, le taux est un taux, le numérateur ne peut pas dépasser son dénominateur, et l'ossature est complète. Exécutez-les à la fin de l'assemblage, à chaque fois.

8.7 Ce qui n'a pas sa place dans la table

— **Les colonnes intermédiaires.** `present_x`, `_merge`, `key_clean` et les six colonnes nécessaires à un seul calcul. Supprimez-les ; c'est ainsi qu'une table devient illisible.

— **Les lignes sous votre seuil de rapportage.** Si vous supprimeriez une cellule de quatre cas dans un tableau publié, ne la transportez pas dans un fichier qui circule par courriel.

— **Tout ce qui est au niveau de la personne.** Une table d'analyse à la granularité école-mois n'a rien à faire d'un identifiant d'élève, et le jour où elle en porte un est le jour où elle ne peut plus être partagée.

8.8 Enregistrez-la avec sa granularité dans le nom

```
analysis.to_csv("outputs/tables/attendance_by_school_month.csv", index=False)
```

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

Et enregistrez le script d'assemblage à côté, car la table est le produit et le script est la méthode. Si régénérer la table demande plus d'une commande, la régénérer dans trois mois n'aura pas lieu.

```
scripts/
  01_read_and_validate.R      contract and validation   (cleaning course)
  02_clean.R                  rules and cleaning log    (cleaning course)
  03_assemble.R               spine, joins, assertions  (this course)
  04_report.R                 tables and figures
```

Quatre scripts, exécutés dans l'ordre, des fichiers bruts à la table finale. C'est l'organisation que *Chaînes d'analyse reproductibles* développera plus loin dans le programme ; l'adopter dès maintenant ne coûte rien et évite la reconstitution.

8.9 Ce que ce cours vous laisse

Vous savez prendre plusieurs fichiers décrivant le même programme et produire une table unique que vous défendriez colonne par colonne — jointures prouvées, granularité énoncée, dénominateur sourcé, calendrier complet, et l'écart avec ce qui a été remonté consigné plutôt que discuté.

Le dernier cours de ce module, *Évaluation de la qualité des données*, prend ce dernier point et en fait une routine — facteurs de vérification sur un échantillon de formations sanitaires, les cinq dimensions de la qualité, et un rapport qui nomme l'action corrective, le responsable et l'échéance, ce qui transforme un écart trouvé en un écart qui se referme.

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/joining-and-reshaping

Généré le 27 juillet 2026.