CASSION · FORMATION

Nettoyage et validation des données

Valeurs manquantes, doublons, mesures impossibles et codages incohérents — comment repérer chaque cas, décider du traitement et laisser une trace qu'un auditeur de données peut suivre.

Formateur	Pierre Robentz Cassion
Niveau	Débutant
Heures apprenant	16 h
Enseignée en	Python · R
Mise à jour	27 juillet 2026
Secteurs	Santé publique · Nutrition · EAH
Normes et méthodologies	Normes OMS de croissance de l'enfant · Norme humanitaire fondamentale (CHS) · Standards Sphère

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

Ce que vous saurez faire

— Profiler un export neuf avant d'y toucher, et conserver ce profil comme trace de l'état du fichier à son arrivée

— Montrer si la non-réponse se concentre sur un site ou une semaine, et dire ce que la suppression des lignes incomplètes ferait au classement que vous publiez

— Prouver qu'un tableau possède bien la clé que vous lui supposez, et retrouver les enregistrements en double qui ne partagent aucune clé

— Écrire des règles de plausibilité à partir des seuils du secteur, et distinguer une erreur d'unité récupérable d'une valeur définitivement perdue

— Rapprocher des noms de sites et de villages saisis en texte libre d'une liste administrative sans inventer d'appariement

— Livrer un journal de nettoyage énonçant chaque règle, le nombre de lignes concernées, ce qui a été modifié et qui l'a validé

Prérequis

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

Sommaire

À propos de cette formation

Fondamentaux de l'analyse de données vous a montré que les données de routine arrivent avec des défauts et vous a donné cinq contrôles pour les repérer. Ce cours pose la question suivante : **que faire concrètement de chacun d'eux, et comment le prouver ensuite ?**

La question est plus lourde qu'il n'y paraît. Presque toute décision de nettoyage déplace un chiffre publié, la plupart le déplacent dans un sens qui arrange, et presque aucune n'est consignée. Un classement de districts qui change quand on supprime les lignes incomplètes, une charge de cas qui baisse de 2 % après dédoublonnage, un taux de couverture qui monte parce que onze villages étaient orthographiés de quatre façons et que trois de ces graphies ont été écartées faute d'appariement — chacun de ces cas est une décision défendable et un silence qui ne l'est pas.

Le cours est donc bâti sur une discipline : **chaque modification est une règle, chaque règle est comptée, et le compte accompagne le chiffre.** À la fin de la dernière unité, vous disposerez d'un script de validation qui tourne sur l'export du trimestre suivant sans être retouché, et d'un journal de nettoyage qu'un auditeur peut lire sans avoir à vous poser de question.

Chaque technique est présentée en Python et en R. Les exemples s'exécutent sur des jeux de données de la plateforme qui portent réellement ces défauts — le registre de dépistage par PB de douze communes de l'Artibonite, l'enquête ménage EAH où un nom de district est écrit de quatre façons, et l'extraction vaccinale de type DHIS2 où une formation sanitaire qui n'a pas rapporté et une formation sanitaire qui n'a vacciné personne sont indiscernables.

Vous devriez avoir suivi *Fondamentaux de l'analyse de données*, ou être à l'aise pour charger un export dans un tableau et le regrouper. Les statistiques ne sont pas nécessaires ; le seul cours qui en exige le dit.

Première partie

Avant de modifier quoi que ce soit

CHAPITRE 1

Le profil que l'on exécute avant d'y toucher

Leçon 1 sur 8 · 90 min

Un export neuf est une pièce à conviction tant que vous n'y touchez pas. Le lire d'abord comme du texte brut, inventorier chaque colonne, prouver ou réfuter la clé, et enregistrer le résultat comme trace de ce qui est arrivé.

1.1 Le fichier est une pièce à conviction tant que vous ne l'éditez pas

Dans six mois, quelqu'un vous demandera si le chiffre d'août a toujours été ainsi. Cette conversation a deux issues possibles. Soit vous ouvrez un fichier enregistré le jour de l'arrivée de l'export et vous y lisez la réponse, soit vous la reconstituez de mémoire, à partir d'un script modifié onze fois depuis.

La première chose à faire d'un export neuf n'est donc pas de le nettoyer. C'est de **le profiler et d'écrire ce profil**, avant qu'une seule valeur ne change. Le profil coûte vingt lignes, et c'est le seul artefact capable de prouver un jour l'état du fichier à son arrivée.

Cette leçon construit ce profil. Le cours *Fondamentaux* enseignait les cinq contrôles qui repèrent les défauts ; celui-ci en fait quelque chose que l'on conserve.

1.2 Lisez-le d'abord comme du texte

Tout lecteur de fichier devine. `read_csv` regarde les premiers milliers de lignes et décide que chaque colonne est un nombre, une date ou une chaîne, et chacune de ces suppositions peut détruire une information irrécupérable — un zéro initial sur un code de formation sanitaire, une sentinelle `-99` moyennée, une date lue à l'américaine.

Lisez une première fois avec tout en chaîne de caractères. Cette lecture sert à regarder, pas à calculer.

```python
import pandas as pd

PATH = "muac-screening-artibonite-2024.v1.csv"
raw = pd.read_csv(PATH, dtype="string", keep_default_na=False)

print(raw.shape)
print(raw.dtypes)
```

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

PATH <- "muac-screening-artibonite-2024.v1.csv"
raw <- read_csv(PATH, col_types = cols(.default = col_character()))

dim(raw)
```

Deux arguments travaillent réellement dans l'appel Python. `dtype="string"` désactive l'inférence de type ; `keep_default_na=False` empêche pandas de transformer les chaînes littérales `NA`, `N/A`, `null` et `nan` en valeurs manquantes avant que vous ne les ayez vues. Dans ce secteur,

cela compte plus qu'il n'y paraît — une colonne où un enquêteur a tapé `NA` pour « sans objet » n'est pas la même colonne que celle où le champ a été sauté, et pandas confond les deux par défaut.

La lecture R impose délibérément `col_character()` partout, ce dont `problems()` a besoin pour être utile — en lisant strictement plus tard, toute valeur non convertible est signalée au lieu d'être silencieusement transformée en `NA`.

1.3 Ce à quoi un profil doit répondre

Six questions, et un profil qui ne répond pas aux six n'en est pas un.

— **Combien est arrivé ?** Lignes et colonnes, face à ce que vous attendiez.

— **Qu'est réellement chaque colonne ?** Ses valeurs brutes, pas le type deviné.

— **Où sont les trous ?** Par colonne et par site — jamais un chiffre global.

— **La clé en est-elle une ?** Non pas « la colonne existe-t-elle » mais « est-elle unique ».

— **Qu'est-ce qui sort des bornes ?** Face aux limites du secteur, pas à celles des données.

— **Quels codes circulent ?** Chaque valeur distincte de chaque colonne catégorielle.

1.4 La forme, face à une attente que vous écrivez

```python
EXPECTED_COLUMNS = [
    "child_id", "commune", "screening_date", "age_months",
    "sex", "muac_mm", "oedema", "outcome",
]

print(f"{len(raw)} rows, {raw.shape[1]} columns")
print("missing columns:", set(EXPECTED_COLUMNS) - set(raw.columns))
print("unexpected columns:", set(raw.columns) - set(EXPECTED_COLUMNS))
```

```r
EXPECTED_COLUMNS <- c(
  "child_id", "commune", "screening_date", "age_months",
  "sex", "muac_mm", "oedema", "outcome"
)

cat(nrow(raw), "rows,", ncol(raw), "columns\n")
setdiff(EXPECTED_COLUMNS, names(raw))
setdiff(names(raw), EXPECTED_COLUMNS)
```

Ce registre compte 4 218 lignes et huit colonnes. L'intérêt n'est pas le décompte, c'est que vous ayez écrit ce que vous attendiez. Une colonne qui disparaît discrètement d'un export à l'autre est la rupture la plus fréquente que livrent les plateformes de collecte, et elle reste invisible tant que rien ne compare à une liste.

1.5 L'inventaire des colonnes

Pour chaque colonne — combien de lignes vides, combien de valeurs distinctes, et les quelques plus fréquentes. Ce seul tableau remplace l'essentiel de ce que l'on fait en faisant défiler le fichier.

```python
def inventory(df):
```

```
    rows = []
    for column in df.columns:
        values = df[column]
        counts = values.value_counts(dropna=False)
        rows.append({
            "column": column,
            "blank": int((values == "").sum()),
            "distinct": int(values.nunique(dropna=False)),
            "top": counts.index[0] if len(counts) else None,
            "top_n": int(counts.iloc[0]) if len(counts) else 0,
        })
    return pd.DataFrame(rows)

print(inventory(raw).to_string(index=False))
```

```
inventory <- function(df) {
  purrr::map_dfr(names(df), function(column) {
    values <- df[[column]]
    counts <- sort(table(values, useNA = "ifany"), decreasing = TRUE)
    tibble::tibble(
      column   = column,
      blank    = sum(values == "" | is.na(values)),
      distinct = dplyr::n_distinct(values),
      top      = names(counts)[1],
      top_n    = as.integer(counts[1])
    )
  })
}

print(inventory(raw), n = Inf)
```

Exécutez-le sur ce registre et deux lignes méritent qu'on s'y arrête.

Colonne	Vides	Distinctes	Remarque
age_months	226	55	5,4 % de vides — la leçon 2 demande où
oedema	44	4	quatre valeurs distinctes dans une colonne booléenne

Quatre valeurs distinctes dans une colonne booléenne, voilà le constat. Le registre contient `true`, `false`, une chaîne vide et `N` — vingt-cinq lignes où un enquêteur a utilisé les conventions `Y`/`N` dans une colonne que le formulaire attendait en `true`/`false`. Convertissez cette colonne en booléen et les lignes `N` deviennent manquantes, silencieusement, et vous avez perdu vingt-cinq *négatifs* enregistrés en les traitant comme non renseignés.

C'est tout l'argument en faveur d'une première lecture en texte. Après la conversion, il n'y a plus rien à trouver.

1.6 Les sentinelles ne sont pas encore des valeurs manquantes

`muac_mm` a l'air complète — aucune case vide. Elle ne l'est pas.

```
print(raw["muac_mm"].value_counts().head())
print("rows coded -99:", int((raw["muac_mm"] == "-99").sum()))
```

```
raw |> count(muac_mm, sort = TRUE) |> head()
sum(raw$muac_mm == "-99")
```

Soixante-douze lignes portent `-99`, le code du registre pour « non mesuré ». Lisez la colonne comme un nombre sans déclarer ce code et la moyenne perd environ deux millimètres, la charge de cas est sous-estimée, parce que vous avez moyenné soixante-douze enfants à moins quatre-vingt-dix-neuf.

Les codes sentinelles sont documentés dans le dictionnaire des données et nulle part ailleurs. **Consultez le dictionnaire avant la première lecture numérique, à chaque fois** — `-99`, `-1`, `999`, `9999` et `88` sont tous en usage dans ce secteur, et aucun n'a l'air faux dans un résumé.

1.7 La clé en est-elle une ?

La colonne nommée `child_id` est un identifiant. Savoir si elle est *unique* est une autre question, et la réponse est ici non.

```python
duplicated_ids = raw["child_id"].duplicated(keep=False)
print("rows sharing a child_id:", int(duplicated_ids.sum()))
print("distinct ids involved:", raw.loc[duplicated_ids, "child_id"].nunique())
```

```r
raw |>
  group_by(child_id) |>
  filter(n() > 1) |>
  ungroup() |>
  summarise(rows = n(), ids = n_distinct(child_id))
```

Vingt-quatre lignes réparties sur douze identifiants. Toute jointure que vous écrirez désormais fait une hypothèse sur cette colonne, et cette hypothèse est actuellement fausse. L'unité 2 traite de cela ; le rôle du profil se borne à le faire apparaître dès le premier jour plutôt qu'au milieu d'une jointure qui double discrètement une charge de cas.

1.8 Les bornes, avant de croire un résumé

Un résumé en cinq nombres masque précisément les valeurs que vous cherchez, car une ligne aberrante déplace à peine un quartile. Regardez les extrêmes directement.

```python
muac = pd.to_numeric(raw["muac_mm"], errors="coerce")
muac = muac.where(muac != -99)

print(muac.describe())
print(muac.nsmallest(10).tolist())
print(muac.nlargest(10).tolist())
```

```r
muac <- suppressWarnings(as.numeric(raw$muac_mm))
muac[muac == -99] <- NA

summary(muac)
head(sort(muac), 10)
head(sort(muac, decreasing = TRUE), 10)
```

Les dix plus petites valeurs donnent 13, 14, 14, 15, 16, 17, 18 puis sautent à

1. Les sept premières sont des centimètres jamais convertis — un PB de 13,4 cm

saisi 13. La moyenne ne les remarque pas. La queue de distribution les nomme immédiatement.

Regardez toujours les dix plus petites et les dix plus grandes valeurs de toute colonne de mesure. Cela coûte une ligne, et c'est le contrôle le plus rentable de cette leçon.

1.9 Enregistrez le profil à côté des données

Le profil ne vaut d'être écrit que s'il survit à la session.

```python
from pathlib import Path
import json

profile = {
    "file": PATH,
    "rows": len(raw),
    "columns": list(raw.columns),
    "blank_by_column": {c: int((raw[c] == "").sum()) for c in raw.columns},
    "distinct_by_column": {c: int(raw[c].nunique()) for c in raw.columns},
    "duplicate_key_rows": int(raw["child_id"].duplicated(keep=False).sum()),
    "categorical_values": {
        c: sorted(raw[c].unique().tolist())
        for c in ["commune", "sex", "oedema", "outcome"]
    },
}

Path("outputs/profiles").mkdir(parents=True, exist_ok=True)
Path("outputs/profiles/muac-2024-q4.json").write_text(json.dumps(profile, indent=2))
```

```r
profile <- list(
  file    = PATH,
  rows    = nrow(raw),
  columns = names(raw),
  blank_by_column    = sapply(raw, function(x) sum(x == "" | is.na(x))),
  distinct_by_column = sapply(raw, dplyr::n_distinct),
  duplicate_key_rows = sum(duplicated(raw$child_id) | duplicated(raw$child_id, fromLast =
    TRUE)),
  categorical_values = lapply(
    raw[c("commune", "sex", "oedema", "outcome")],
    function(x) sort(unique(x))
  )
)

dir.create(here::here("outputs", "profiles"), recursive = TRUE, showWarnings = FALSE)
jsonlite::write_json(
  profile,
  here::here("outputs", "profiles", "muac-2024-q4.json"),
  pretty = TRUE, auto_unbox = TRUE
)
```

Du JSON plutôt qu'un tableau imprimé, parce que toute la section suivante consiste à en comparer deux.

1.10 Comparer cet export au précédent

La plupart des exports sont le même export, un trimestre plus tard. La question intéressante n'est donc jamais « qu'y a-t-il dans ce fichier » mais **« qu'est-ce qui a changé »**.

```python
previous = json.loads(Path("outputs/profiles/muac-2024-q3.json").read_text())
```

```
for column, values in profile["categorical_values"].items():
    was, now = set(previous["categorical_values"][column]), set(values)
    if was != now:
        print(f"{column}: new {sorted(now - was)}, gone {sorted(was - now)}")

growth = (profile["rows"] - previous["rows"]) / previous["rows"]
print(f"row count moved {growth:.1%}")
```

```
previous <- jsonlite::read_json(
  here::here("outputs", "profiles", "muac-2024-q3.json"),
  simplifyVector = TRUE
)

for (column in names(profile$categorical_values)) {
  was <- previous$categorical_values[[column]]
  now <- profile$categorical_values[[column]]
  if (!setequal(was, now)) {
    cat(column, ": new", setdiff(now, was), "| gone", setdiff(was, now), "\n")
  }
}

sprintf("row count moved %.1f%%", 100 * (profile$rows - previous$rows) / previous$rows)
```

Une valeur nouvelle dans une colonne catégorielle est le moyen le plus courant pour qu'une analyse se mette à produire des réponses fausses sans jamais échouer. Un formulaire gagne une modalité de réponse, une treizième commune est ajoutée, un antigène est renommé — et tout `case_when` écrit avant ce jour envoie désormais la valeur nouvelle dans sa branche par défaut. Comparer les profils l'attrape à l'arrivée, le seul moment où c'est bon marché.

> Un profil que vous n'avez pas enregistré est un profil que vous n'avez pas exécuté. Toute sa valeur tient à la possibilité de le rouvrir plus tard.

1.11 La suite

Vous savez désormais que ce registre a perdu 5,4 % de ses âges. Ce chiffre seul ne veut presque rien dire — il importe énormément de savoir si ces 226 lignes sont dispersées sur douze communes ou concentrées sur une seule. La leçon suivante décompose la non-réponse jusqu'à ce qu'elle cesse d'avoir l'air d'un accident ou qu'elle prouve qu'elle en est un, et chiffre ce que la suppression de ces lignes ferait au classement que vous publiez.

CHAPITRE 2

Une non-réponse qui n'est pas un accident

Leçon 2 sur 8 · 90 min

Un taux de complétude global masque la seule chose qui compte. Décomposer la non-réponse par site et par semaine, nommer le mécanisme, et chiffrer ce que la suppression des lignes incomplètes fait à votre classement.

2.1 5,4 %, ce n'est pas un constat

Le registre n'a pas d'`age_months` sur 226 de ses 4 218 lignes. Présenté comme « le jeu de données est complet à 94,6 % », ce chiffre est pire qu'inutile — il invite le lecteur à conclure que le problème est petit, et il ne l'est pas, parce qu'il n'est pas réparti.

Un taux de complétude ne signifie quelque chose qu'une fois qu'on sait où sont les trous. Une non-réponse dispersée coûte de la précision. Une non-réponse concentrée coûte la réponse. Cette leçon montre comment savoir dans quel cas vous êtes, et quoi en dire.

2.2 Décomposez par groupe avant toute chose

```python
by_commune = (
    muac.assign(missing_age=muac["age_months"].isna())
    .groupby("commune")
    .agg(missing=("missing_age", "mean"), n=("missing_age", "size"))
    .sort_values("missing", ascending=False)
)
print((by_commune["missing"] * 100).round(1))
```

```r
muac |>
  group_by(commune) |>
  summarise(missing = mean(is.na(age_months)), n = n()) |>
  arrange(desc(missing)) |>
  mutate(missing = round(100 * missing, 1))
```

Commune	Âge manquant	Lignes
Gros-Morne	17,7 %	361
Anse-Rouge	5,7 %	229
Saint-Michel	5,1 %	335
Dessalines	4,9 %	450
Ennery	3,0 %	263

Onze communes se situent entre 3 % et 6 %. Une seule est à 17,7 %. Ce n'est pas une distribution à longue traîne, ce sont onze communes avec une non-réponse ordinaire et **une commune avec un problème**.

2.3 Puis par période, à l'intérieur du groupe qui ressort

```python
gros_morne = muac[muac["commune"] == "Gros-Morne"].copy()
gros_morne["week"] = gros_morne["screening_date"].dt.to_period("W").dt.start_time

print(
    gros_morne.groupby("week")["age_months"]
    .apply(lambda s: s.isna().mean())
    .sort_values(ascending=False)
    .head()
)
```

```r
muac |>
  filter(commune == "Gros-Morne") |>
  mutate(week = lubridate::floor_date(screening_date, "week", week_start = 1)) |>
  group_by(week) |>
  summarise(missing = mean(is.na(age_months)), n = n()) |>
  arrange(desc(missing)) |>
  head()
```

La semaine du 10 juin — 54 dépistages sur 85, soit 63,5 %. Toutes les autres semaines de cette commune sont sous les 10 %.

Ce n'est plus une statistique de qualité des données. C'est **une équipe, une semaine, un formulaire de tablette dont le champ âge était mal configuré**, et cela porte un nom, une date et probablement une personne qui s'en souvient. Une non-réponse qui se ramène à un événement est une non-réponse sur laquelle on peut poser une question, et souvent la corriger à la source.

2.4 Nommez le mécanisme, dans des mots que le rapport peut porter

La littérature statistique distingue trois catégories. Vous n'avez pas besoin de la notation, mais bien de la distinction, car elle décide de ce que vous avez le droit de faire.

— **Manquant complètement au hasard.** Les trous ne dépendent de rien, pas même de la valeur absente. Supprimer ces lignes coûte de la précision et rien d'autre.

— **Manquant au hasard.** Les trous dépendent de quelque chose d'*observé* — une commune, une semaine, un enquêteur. Les supprimer biaise le résultat, mais on peut ajuster sur ce dont ils dépendent, puisqu'on le possède.

— **Manquant non au hasard.** Les trous dépendent de la valeur absente elle-même. Les enfants les plus malades sont ceux que la file n'a jamais atteints ; les ménages ayant refusé l'analyse d'eau sont ceux dont l'eau est la plus sale. Rien dans les données ne corrige cela, et la seule réponse honnête est de le dire.

La non-réponse sur l'âge est ici **manquante au hasard** — elle dépend de la commune et de la semaine, toutes deux enregistrées. Les codes `-99` du PB sont peut-être pires — si la mesure a été sautée lorsque l'enfant était agité, et que l'agitation est corrélée à la maladie, alors c'est du manquant non au hasard, et aucun code ne vous le dira.

> La catégorie n'est pas une subtilité statistique. C'est la différence entre « nous avons supprimé 226 lignes » et « nous avons supprimé 226 lignes, dont 54 d'une seule commune en une seule semaine, ce qui déplace le taux de cette commune de 1,1 point ».

2.5 Chiffrez ce que leur suppression coûte

C'est l'étape que presque personne ne fait, et elle prend six lignes. Calculez l'indicateur deux fois — une fois sur tout, une fois sur les seuls cas complets.

```python
def gam_rate(df):
    assessed = df["muac_mm"].notna() | df["oedema"].notna()
    cases = assessed & ((df["muac_mm"] < 125) | (df["oedema"] == True))
    return pd.Series({
        "assessed": int(assessed.sum()),
        "cases": int(cases.sum()),
        "rate": round(cases.sum() / assessed.sum(), 3),
    })

all_rows = muac.groupby("commune").apply(gam_rate)
complete = muac[muac["age_months"].notna()].groupby("commune").apply(gam_rate)

comparison = all_rows.join(complete, rsuffix="_complete")
comparison["shift"] = comparison["rate_complete"] - comparison["rate"]
print(comparison.sort_values("shift"))
```

```r
gam_rate <- function(df) {
  df |>
    mutate(
      assessed = !is.na(muac_mm) | !is.na(oedema),
      case     = assessed & (muac_mm < 125 | oedema)
    ) |>
    summarise(
      assessed = sum(assessed, na.rm = TRUE),
      cases    = sum(case, na.rm = TRUE),
      rate     = round(cases / assessed, 3),
      .by      = commune
    )
}

comparison <- gam_rate(muac) |>
  left_join(gam_rate(filter(muac, !is.na(age_months))),
            by = "commune", suffix = c("", "_complete")) |>
  mutate(shift = rate_complete - rate) |>
  arrange(shift)
```

Commune	Toutes lignes	Cas complets	Écart
Gros-Morne	14,5 %	13,4 %	-1,1 pt
Anse-Rouge	15,6 %	16,5 %	+0,9 pt
Dessalines	5,8 %	5,4 %	-0,4 pt
Saint-Michel	7,6 %	7,6 %	0,0 pt

Lisez les deux premières lignes ensemble. Sur toutes les lignes, Gros-Morne est deuxième plus touchée à 14,5 %, avec 1,7 point d'avance sur la commune suivante. Sur les cas complets, elle est à 13,4 % et la commune suivante à 13,3 %. **L'écart entre la deuxième et la troisième place passe de 1,7 point à 0,1** — autrement dit, le classement que vous poseriez sur la table d'une réunion du cluster nutrition dépend d'un formulaire mal configuré dans une commune au mois de juin.

Remarquez aussi que rien ici n'a franchi de seuil au point de changer une décision. Dites-le également. Un test de sensibilité qui ne trouve aucun effet est un résultat, et c'est celui qui permet d'arrêter de s'inquiéter.

2.6 Les trois réponses, et le moment où chacune est honnête

Les supprimer, et déclarer la suppression. Légitime quand la non-réponse est dispersée et que vous l'avez montré. La déclaration n'est pas facultative — « n = 3 992 sur 4 218 ; 226 lignes exclues pour âge manquant » a sa place dans la note de bas de tableau, pas dans votre tête.

Les garder, dans une catégorie à part. Souvent la bonne réponse pour une variable catégorielle, car « non renseigné » est un vrai constat sur le programme. Une colonne de complétude à côté de la colonne d'indicateur en dit plus que chacune séparément.

Imputer — prudemment, et rarement. Pour une variable servant à *désagréger* plutôt qu'à calculer, remplir l'âge manquant par l'âge médian de la même commune et du même mois est défendable si vous marquez chaque ligne imputée et rapportez l'indicateur des deux façons. Pour une variable du numérateur, imputer revient à inventer le constat.

```python
muac["age_imputed"] = muac["age_months"].isna()
muac["age_months_filled"] = muac.groupby("commune")["age_months"].transform(
    lambda s: s.fillna(s.median())
)
```

```r
muac <- muac |>
  mutate(age_imputed = is.na(age_months)) |>
  group_by(commune) |>
  mutate(age_months_filled = ifelse(is.na(age_months),
                                    median(age_months, na.rm = TRUE),
                                    age_months)) |>
  ungroup()
```

La colonne de marquage est l'essentiel. Une valeur imputée qu'on ne distingue plus d'une valeur mesurée a cessé d'être une estimation pour devenir une affirmation.

2.7 Des lignes manquantes, pas seulement des valeurs

La non-réponse la plus difficile est celle sans case vide, parce que la ligne n'est pas là du tout.

FIGURE 2.1 – Part des formations sanitaires ayant transmis un rapport mensuel au cours de 2024. L'effondrement d'août et septembre bouge de concert dans tout le district, ce qui en fait un événement de rapportage et non une défaillance de formation sanitaire.

Dans l'extraction vaccinale de routine, une formation sanitaire qui n'a pas rapporté n'est

pas absente — elle est présente avec `doses_administered` à zéro et `report_submitted` à `false`. Six cent quarante-deux lignes sont dans cet état. Calculez la couverture sans les séparer et toute formation sanitaire silencieuse devient une formation sanitaire n'ayant vacciné personne.

```python
vax["reported"] = vax["report_submitted"] == True

reporting_rate = vax.groupby("period")["reported"].mean()
coverage = (
    vax[vax["reported"]]
    .groupby("period")
    .apply(lambda g: g["doses_administered"].sum() / g["target_population"].sum())
)
print(pd.DataFrame({"reporting_rate": reporting_rate, "coverage": coverage}))
```

```r
vax |>
  summarise(
    reporting_rate = mean(report_submitted),
    coverage = sum(doses_administered[report_submitted]) /
               sum(target_population[report_submitted]),
    .by = period
  )
```

Deux chiffres, toujours publiés ensemble. **La couverture calculée sur les formations sanitaires ayant rapporté, et le taux de complétude qui dit quelle part du district cela représente.** Un taux de couverture unique qui inclut discrètement les structures silencieuses à son dénominateur est le défaut le plus répandu des données sanitaires de routine, et il tire toujours dans le même sens — vers le bas.

2.8 La suite

La non-réponse est un trou là où quelque chose devrait être. L'unité suivante traite du problème inverse — quelque chose qui est là deux fois. La leçon 3 demande si la colonne que vous traitez comme une clé en est réellement une, et montre ce que fait une jointure quand elle n'en est pas une.

Deuxième partie

Identité et doublons

CHAPITRE 3

Prouvez la clé avant de joindre

Leçon 3 sur 8 · 90 min

Toute jointure formule une hypothèse d'unicité que rien ne vérifie. La tester, voir deux lignes de liste dupliquées gonfler de 120 lignes une table de 70 245, et apprendre à affirmer le nombre de lignes plutôt qu'à le regarder.

3.1 L'hypothèse que rien ne vérifie

Vous avez une liste d'élèves et un fichier de présence journalière. Vous les joignez sur `student_id`, et ce faisant vous avez affirmé quelque chose — **que `student_id` identifie exactement une ligne de la liste.**

Rien dans aucun des deux fichiers ne le dit. Aucune erreur n'est levée si c'est faux. Ce qui se produit à la place, c'est que la jointure produit discrètement plus de lignes qu'elle n'en avait, que tous les décomptes en aval sont gonflés d'une quantité invisible, et que le taux de présence obtenu est faux d'un écart trop petit pour avoir l'air faux.

Cette leçon consiste à tester cette hypothèse avant de la formuler.

3.2 Une clé est une affirmation, et elle est testable

L'affirmation a deux volets — **unique** et **complète**. Une colonne comportant des doublons n'est pas une clé. Une colonne comportant des valeurs manquantes n'en est pas une non plus, car une valeur manquante n'identifie rien.

```python
def is_key(df, columns):
    subset = df[columns]
    return {
        "rows": len(df),
        "distinct": len(subset.drop_duplicates()),
        "missing_any": int(subset.isna().any(axis=1).sum()),
        "is_key": len(subset.drop_duplicates()) == len(df)
                  and not subset.isna().any(axis=None),
    }

print(is_key(roster, ["student_id"]))
print(is_key(attendance, ["student_id", "attendance_date"]))
```

```r
is_key <- function(df, columns) {
  subset <- dplyr::select(df, dplyr::all_of(columns))
  list(
    rows        = nrow(df),
    distinct    = nrow(dplyr::distinct(subset)),
    missing_any = sum(!stats::complete.cases(subset)),
    is_key      = nrow(dplyr::distinct(subset)) == nrow(df) &&
                  !any(is.na(subset))
  )
}

is_key(roster, "student_id")
is_key(attendance, c("student_id", "attendance_date"))
```

Sur cette liste — 1 202 lignes, 1 200 `student_id` distincts. **Ce n'est pas une clé**, à deux lignes près.

Deux lignes sur 1 202 font 0,17 % et cela n'a l'air de rien. Regardez ce qu'elles sont.

```python
repeated = roster[roster["student_id"].duplicated(keep=False)]
print(repeated.sort_values("student_id"))
```

```r
roster |>
  group_by(student_id) |>
  filter(n() > 1) |>
  arrange(student_id)
```

student_id	school_id	grade	feeding_programme
STU0150	SCH09	3	false
STU0150	SCH08	3	true
STU0896	SCH01	2	true
STU0896	SCH06	2	false

Deux élèves ont changé d'école sans jamais être radiés de la première. Chacun figure désormais **une fois dans une école avec cantine et une fois dans une école sans** — et l'analyse phare de ce jeu de données porte précisément sur l'effet de l'alimentation scolaire sur la présence. Ces deux élèves apporteront tout leur historique de présence aux deux bras de la comparaison.

Telle est la forme générale de ce défaut. Le doublon est rarement fortuit ; il est en général la trace de quelque chose qui a eu lieu — un transfert, une réinscription, un formulaire envoyé deux fois — et il atterrit en général exactement dans la variable sur laquelle porte l'analyse.

3.3 Clés composites, et celle dont vous disposez vraiment

`student_id` n'est pas une clé de la liste. `student_id` plus `school_id` en est une, puisque les deux lignes diffèrent par l'école. Savoir si c'est la clé que vous *voulez* est une autre question — si l'analyse suppose « une ligne par élève », une clé composite qui admet deux lignes par élève a documenté le problème, pas résolu.

Le fichier de présence relève du cas plus courant, où la clé naturelle est composite dès le départ.

```python
print(is_key(attendance, ["student_id", "attendance_date"]))
```

```r
is_key(attendance, c("student_id", "attendance_date"))
```

70 245 lignes, 70 245 couples distincts. **Voilà une clé.** Une ligne par élève et par jour de classe, exactement ce que le fichier prétend être.

Prenez l'habitude d'écrire la clé dans le code, à côté de la lecture.

```python
ROSTER_KEY = ["student_id"]          # intended; violated by 2 rows, see cleaning log
ATTENDANCE_KEY = ["student_id", "attendance_date"]
```

```r
ROSTER_KEY <- "student_id"           # intended; violated by 2 rows, see cleaning log
ATTENDANCE_KEY <- c("student_id", "attendance_date")
```

3.4 Ce qu'une clé cassée fait à une jointure

L'arithmétique mérite d'être faite une fois à la main, car elle explique tous les gonflements que vous rencontrerez.

Une jointure apparie chaque ligne de gauche à toutes les lignes de droite partageant la clé. Si un élève a 60 lignes de présence et 1 ligne de liste, la jointure produit 60 lignes. Si cet élève a 2 lignes de liste, elle en produit **120**.

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

```r
joined <- attendance |> left_join(roster, by = "student_id")
cat(nrow(attendance), "->", nrow(joined), "\n")
```

70 245 devient 70 365. Cent vingt lignes de plus — deux élèves, soixante jours de classe chacun, comptés deux fois.

Une jointure à gauche qui ajoute des lignes est une contradiction dans les termes : « jointure à gauche » se lit d'ordinaire « garder la table de gauche et y accrocher des colonnes », et cette lecture n'est vraie que si le côté droit est unique sur la clé. Ce n'est pas un défaut de la jointure. C'est la jointure qui vous dit la vérité sur vos données, sous une forme que personne ne regarde.

3.5 Dites ce que vous attendez, et laissez la jointure échouer

Les deux langages vérifient la relation pour vous. Servez-vous-en.

```python
joined = attendance.merge(
    roster,
    on="student_id",
    how="left",
    validate="many_to_one",   # many attendance rows, one roster row
)
```

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

Les deux **échouent bruyamment** sur ces données.

```
MergeError: Merge keys are not unique in right dataset; not a many-to-one merge
```

```
Error in `left_join()`:
! Each row in `x` must match at most 1 row in `y`.
i Row 1 of `x` matches multiple rows in `y`.
```

Une erreur qu'il faut traiter aujourd'hui vaut mieux qu'un dénominateur de présence discrètement gonflé qu'on traitera en réunion de revue au mois de novembre. **Mettez `validate=` ou `relationship=` sur chaque jointure que vous écrivez.** Cela coûte un argument et transforme la classe de bugs silencieux la plus coûteuse en trace d'erreur.

3.6 Protégez le nombre de lignes quand la jointure est l'objectif

Là où l'argument de relation ne convient pas — un plusieurs-à-plusieurs qui en est réellement un — affirmez directement le décompte.

```python
before = len(attendance)
joined = attendance.merge(roster, on="student_id", how="left")
assert len(joined) == before, f"join changed row count: {before} -> {len(joined)}"
```

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

L'affirmation fait trois mots de plus que la jointure. Écrivez-la à chaque fois.

3.7 Les lignes d'un côté et pas de l'autre

Une clé peut être unique et malgré tout ne pas s'apparier. Vérifiez les deux sens avant d'accepter le résultat d'une jointure.

```python
left_only = set(attendance["student_id"]) - set(roster["student_id"])
right_only = set(roster["student_id"]) - set(attendance["student_id"])
print(f"{len(left_only)} students with attendance and no roster row")
print(f"{len(right_only)} students on the roster with no attendance")
```

```r
setdiff(attendance$student_id, roster$student_id) |> length()
setdiff(roster$student_id, attendance$student_id) |> length()
```

Ici les deux valent zéro, ce qui est la réponse souhaitée et qu'on n'obtient presque jamais. Quand ce n'est pas zéro, les deux sens veulent dire des choses totalement différentes.

— **De la présence sans ligne de liste** — un élève marqué présent sans être inscrit. Soit la liste est incomplète, soit on enregistre quelqu'un qui ne devrait pas l'être.

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

Une jointure interne fait disparaître les deux groupes et rapporte un chiffre propre. C'est pourquoi le contrôle précède la jointure et ne la suit pas.

3.8 Des colonnes qui n'étaient pas faites pour être des clés

Noms, numéros de téléphone et noms de chefs de ménage servent constamment de clés, parce qu'ils sont la seule chose que deux fichiers ont en commun. Ce ne sont pas des clés, et le mode de défaillance est asymétrique.

— **Les doublons créent de fausses fusions.** Deux ménages dont le chef s'appelle Jean Baptiste deviennent un seul ménage.

— **Les variantes créent de fausses scissions.** `Jean Baptiste`, `JEAN BAPTISTE` et `Jean Baptiste` en deviennent trois.

Si un nom est réellement le seul lien dont vous disposez, il s'agit d'un **problème d'appariement d'enregistrements**, pas d'une jointure, et la leçon suivante lui est entièrement consacrée. Ce qu'il ne faut surtout pas faire est un `merge(on="name")` suivi de rien.

3.9 Affirmez, ne regardez pas

Le motif que cette leçon enseigne réellement est plus petit que chacun de ses contrôles. Chacun d'eux peut s'écrire comme un affichage qu'on lit une fois, ou comme une assertion qui s'exécute sur chaque export à venir. Seule la seconde survit.

```python
def assert_key(df, columns, name):
    duplicated = df.duplicated(subset=columns, keep=False)
    if duplicated.any():
        offenders = df.loc[duplicated, columns].drop_duplicates()
        raise ValueError(
            f"{name}: {duplicated.sum()} rows violate the key {columns}\n"
            f"{offenders.head(10)}"
        )
```

```r
assert_key <- function(df, columns, name) {
  dup <- duplicated(df[columns]) | duplicated(df[columns], fromLast = TRUE)
  if (any(dup)) {
    stop(sprintf("%s: %d rows violate the key %s", name, sum(dup),
                 paste(columns, collapse = " + ")))
  }
  invisible(df)
}
```

> Un contrôle exécuté une fois vous renseigne sur le fichier que vous aviez. Un contrôle exécuté à la lecture vous renseigne sur le fichier que vous avez.

L'unité 4 transforme ce motif en une suite de validation. Pour l'instant, placez `assert_key` en tête de tout script qui joint quoi que ce soit.

3.10 La suite

`assert_key` trouve les élèves enregistrés deux fois sous le *même* identifiant. Il ne peut pas trouver l'enfant réinscrit sous un nouvel identifiant, le ménage interrogé deux fois par deux enquêteurs, ni le bénéficiaire figurant sur trois listes de distribution avec trois graphies de son nom. Ceux-là ne partagent aucune clé, et la leçon suivante montre comment les retrouver sans fusionner deux personnes qui se ressemblent simplement.

CHAPITRE 4

La même personne, deux fois, sous deux identifiants

Leçon 4 sur 8 · 105 min

L'appariement d'enregistrements quand rien ne joint — blocage, normalisation, distance d'édition et score. Douze paires candidates dans le registre de dépistage, six réelles, et quatre fabriquées par les âges manquants de la leçon 2.

4.1 Le doublon sans clé

Un enfant dépisté le matin, renvoyé chez lui, puis ramené l'après-midi par un autre accompagnant est enregistré deux fois sous deux identifiants. De même pour un ménage visité par deux enquêteurs dans la même rue. De même pour un bénéficiaire figurant sur trois listes de distribution sous trois graphies du même nom.

Aucun de ces cas ne partage de clé. `duplicated()` ne les trouvera jamais. Et ils comptent — dans une charge de cas ils gonflent le numérateur, dans un taux de couverture ils gonflent le numérateur sans le dénominateur, et dans un décompte de bénéficiaires ils sont précisément ce qu'un audit cherche.

Les retrouver relève de l'**appariement d'enregistrements**, discipline différente du dédoublonnage. Le dédoublonnage supprime des lignes identiques. L'appariement décide si deux lignes qui ne sont *pas* identiques décrivent la même chose — et il peut se tromper dans les deux sens, raison pour laquelle le produit de cette leçon est une liste à faire relire par un humain, et non un tableau nettoyé.

4.2 Commencez par les attributs qui ne devraient pas coïncider

Le registre de dépistage n'a ni nom ni ménage. Il a une combinaison qui devrait être quasi unique par accident — commune, date, âge, sexe et mesure.

```python
CANDIDATE_KEY = ["commune", "screening_date", "age_months", "sex", "muac_mm"]

candidates = (
    muac.groupby(CANDIDATE_KEY, dropna=False)
    .filter(lambda g: g["child_id"].nunique() > 1)
    .sort_values(CANDIDATE_KEY)
)
print(candidates[["child_id"] + CANDIDATE_KEY].to_string(index=False))
```

```r
CANDIDATE_KEY <- c("commune", "screening_date", "age_months", "sex", "muac_mm")

candidates <- muac |>
  group_by(across(all_of(CANDIDATE_KEY))) |>
  filter(n_distinct(child_id) > 1) |>
  ungroup() |>
  arrange(across(all_of(CANDIDATE_KEY)))
```

Douze groupes reviennent. Six sont le même enfant réinscrit sous un nouvel identifiant. **Six sont deux enfants différents qui se trouvent coïncider.**

4.3 Regardez les faux avant les vrais

Commune	Date	Âge	Sexe	PB	Identifiants
Desdunes	2024-01-20	36	f	134	CH03315, CH09241
Gros-Morne	2024-06-10	*manquant*	f	131	CH00576, CH02413
Gros-Morne	2024-06-10	*manquant*	m	123	CH01302, CH04050
Gros-Morne	2024-06-13	*manquant*	m	-99	CH00519, CH00755
Gros-Morne	2024-06-14	*manquant*	m	137	CH01558, CH02913
Verrettes	2024-11-09	26	f	127	CH02193, CH03129

Quatre des douze viennent de Gros-Morne, la semaine du 10 juin, et toutes ont un **âge manquant**. C'est la semaine dont le formulaire était mal configuré — la non-réponse de la leçon 2 qui revient dans une autre leçon sous un autre déguisement.

Voici pourquoi. Regrouper sur une colonne manquante fait s'apparier chaque ligne manquante avec toutes les autres sur cette colonne. Le bloc s'effondre : au lieu de comparer des enfants du même âge, vous comparez tous les enfants d'âge inconnu. Dans une commune qui dépiste quatre-vingt-cinq enfants en une semaine, deux garçons ayant le même PB au millimètre n'a rien de surprenant.

Une colonne de blocage comportant des valeurs manquantes fabrique des appariements. Excluez les lignes manquantes du bloc ou bloquez sur autre chose — ne laissez jamais un trou compter comme un accord.

```python
blocked = muac[muac["age_months"].notna() & muac["muac_mm"].notna()]
```

```r
blocked <- muac |> filter(!is.na(age_months), !is.na(muac_mm))
```

Faites-le et huit candidats subsistent — les six réinscriptions réelles et deux coïncidences authentiques à Verrettes et Saint-Marc. **Huit, c'est une liste qu'un superviseur peut confronter au registre papier en une après-midi.** Douze, dont quatre absurdes, c'est une liste qui apprend aux gens à ignorer les listes.

4.4 Le blocage, et pourquoi on ne peut pas s'en passer

Comparer chaque ligne à toutes les autres est quadratique. Sur 4 218 enregistrements cela fait 8,9 millions de paires — lent mais supportable. Sur un registre de 300 000 bénéficiaires, cela fait 45 milliards, ce qui ne l'est pas.

Le **blocage** consiste à ne comparer que les enregistrements déjà d'accord sur quelque chose de simple et fiable — la commune, le site de distribution, le mois, la première lettre du nom de famille. Les comparaisons coûteuses n'ont ensuite lieu qu'à l'intérieur de chaque bloc.

```python
blocks = blocked.groupby(["commune", "sex"])
print(sum(len(g) * (len(g) - 1) // 2 for _, g in blocks), "pairs to compare")
```

```r
blocked |>
  count(commune, sex) |>
  summarise(pairs = sum(n * (n - 1) / 2))
```

L'arbitrage est explicite : un bloc trop grossier est lent, et un bloc trop fin **rate les paires en désaccord sur la colonne de blocage**. Deux graphies d'un nom de village placées dans des blocs différents ne seront jamais comparées. C'est pourquoi la colonne de blocage doit

être celle en laquelle vous avez le plus confiance, et pourquoi bloquer sur un champ en texte libre est généralement une erreur.

4.5 Normalisez avant de comparer

Pour tout ce qui relève du texte, l'essentiel du travail se fait avant le calcul de la moindre distance.

```python
import re
import unicodedata

def normalise(series):
    return (
        series.fillna("")
        .str.normalize("NFKD")
        .str.encode("ascii", "ignore").str.decode("ascii")
        .str.lower()
        .str.replace(r"[^a-z0-9 ]", " ", regex=True)
        .str.replace(r"\s+", " ", regex=True)
        .str.strip()
    )

households["head_key"] = normalise(households["head_of_household"])
```

```r
normalise <- function(x) {
  x |>
    tidyr::replace_na("") |>
    stringi::stri_trans_general("Latin-ASCII") |>
    tolower() |>
    stringr::str_replace_all("[^a-z0-9 ]", " ") |>
    stringr::str_squish()
}

households$head_key <- normalise(households$head_of_household)
```

Casse, accents, espaces doubles et ponctuation expliquent la grande majorité des noms « différents » dans les données de ce secteur. Retirez-les et une part étonnante de votre problème d'appariement approximatif devient de l'appariement exact.

Ne retirez les accents que pour la clé de comparaison, jamais dans les données conservées. `Étienne` est le nom de la personne. `etienne` est un index.

4.6 Distance d'édition et choix d'un seuil

Pour ce qui survit à la normalisation, comparez avec une distance entre chaînes. N'importe laquelle des distances usuelles convient ; ce qui compte est de choisir un seuil délibérément et de dire lequel.

```python
from rapidfuzz import fuzz, process

matches = process.extract(
    "jean baptiste pierre",
    households["head_key"].tolist(),
    scorer=fuzz.token_sort_ratio,
    score_cutoff=88,
    limit=10,
)
```

```
scores <- stringdist::stringsim(
  "jean baptiste pierre",
  households$head_key,
  method = "jw"
)
which(scores > 0.88)
```

`token_sort_ratio` et Jaro-Winkler sont deux valeurs par défaut raisonnables ici, pour une raison qu'il vaut la peine de connaître — dans ce secteur les noms arrivent avec leurs éléments permutés (`Pierre Jean Baptiste` contre `Jean Baptiste Pierre`) et avec une faute de frappe bien plus rarement dans les premiers caractères que dans les derniers.

Le seuil est un **curseur entre précision et rappel**, pas un réglage. À 95 vous obtenez peu de paires et manquez de vrais doublons. À 80 vous en obtenez beaucoup, la plupart fausses, et le relecteur cesse de lire. Réglez-le en prenant un échantillon de cinquante paires à votre seuil proposé et en vérifiant combien sont réelles. Inscrivez le seuil retenu et le taux de justesse mesuré dans le journal de nettoyage.

4.7 Notez plusieurs champs, n'enchaînez pas les filtres

Un seul champ ne suffit jamais. Comparez-en une poignée et combinez-les, pour qu'un accord fort sur un champ compense un accord faible sur un autre.

```
def pair_score(a, b):
    name = fuzz.token_sort_ratio(a["head_key"], b["head_key"]) / 100
    same_site = 1.0 if a["community"] == b["community"] else 0.0
    size_gap = abs(a["household_size"] - b["household_size"])
    size = max(0.0, 1 - size_gap / 5)
    return 0.6 * name + 0.25 * same_site + 0.15 * size
```

```
pair_score <- function(a, b) {
  name      <- stringdist::stringsim(a$head_key, b$head_key, method = "jw")
  same_site <- as.numeric(a$community == b$community)
  size      <- pmax(0, 1 - abs(a$household_size - b$household_size) / 5)
  0.6 * name + 0.25 * same_site + 0.15 * size
}
```

Les pondérations relèvent du jugement, et elles doivent être visibles plutôt qu'enfouies dans une chaîne de conditions. Un filtre enchaîné traite chaque critère comme absolu : une faute dans le nom de la communauté et la paire disparaît. Un score laisse les indices s'additionner, ce qui est de toute façon la manière dont un relecteur humain raisonne.

4.8 Le produit est une file de relecture, pas un tableau propre

C'est ce qui sépare un appariement utile d'un appariement qui provoque un incident.

```
review = (
    pairs.sort_values("score", ascending=False)
    .loc[:, ["id_a", "id_b", "score", "head_a", "head_b", "community", "size_gap"]]
    .assign(decision="", reviewer="", reviewed_on="")
)
review.to_csv("outputs/review/duplicate-candidates.csv", index=False)
```

```
review <- pairs |>
```

```
    arrange(desc(score)) |>
    select(id_a, id_b, score, head_a, head_b, community, size_gap) |>
    mutate(decision = "", reviewer = "", reviewed_on = "")

readr::write_csv(review, here::here("outputs", "review", "duplicate-candidates.csv"))
```

Trois colonnes vides, et c'est là tout l'intérêt. Le fichier part chez la personne qui tient le registre, revient avec une décision sur chaque ligne, et **c'est ce fichier revenu que le script de nettoyage lit** — pas le score, et pas un seuil appliqué automatiquement.

> Un doublon fusionné est un enregistrement détruit. Si la fusion était fausse, la preuve qu'elle l'était a disparu avec lui.

4.9 Ce qu'il ne faut pas automatiser

Dans les données de bénéficiaires, de protection et de gestion de cas, la fusion automatique n'est pas un raccourci technique à faible taux d'erreur. C'est une décision qui porte sur une personne, et les deux modes de défaillance ne sont pas symétriques.

— **Une fausse fusion supprime quelqu'un.** Deux ménages n'en font plus qu'un ; l'un cesse de recevoir l'assistance sans aucun moyen de savoir pourquoi. Dans une charge de cas de protection, deux survivantes deviennent un seul dossier, et l'histoire de l'une est rattachée au nom de l'autre.

— **Une fausse scission compte double.** C'est coûteux et embarrassant, et cela ne prive personne de rien.

Cette asymétrie impose de sous-fusionner par défaut. Signalez, faites relire, et laissez décider la personne qui tient le registre. Lorsqu'une fusion a lieu, conservez les deux identifiants d'origine dans l'enregistrement fusionné pour pouvoir revenir en arrière — une table d'appariement avec `kept_id`, `merged_id`, `score`, `reviewer` et `date` à côté des données.

Rien de tout cela n'est de la prudence gratuite. Les principes de gestion de l'information VBG qui régissent une partie des données de ce secteur le disent directement : le préjudice d'un enregistrement mal traité retombe sur la personne qu'il décrit, pas sur l'analyste.

4.10 La suite

Vous savez désormais quelles lignes désignent la même chose et quelles colonnes les identifient. L'unité suivante s'attaque aux valeurs elles-mêmes — les mesures qui ne peuvent pas être vraies, les codes de sortie qui contredisent leur propre mesure, et les seuils sectoriels qui transforment « cela semble faux » en une règle qu'un script applique.

Troisième partie

Des valeurs qui ne peuvent pas être vraies

CHAPITRE 5

Des règles que le secteur vous a déjà données

Leçon 5 sur 8 · 90 min

Transformer « cela semble faux » en une table de règles avec une source, une gravité et un compte. Bornes dures face aux limites physiques, bornes souples face aux seuils du secteur, et la discipline du signalement plutôt que de la suppression.

5.1 « Cela semble faux » n'est pas une règle

Quiconque fait ce métier finit par acquérir l'œil. Vous faites défiler une colonne, quelque chose accroche, vous allez voir. Cela marche, et cela ne passe pas à l'échelle, ne se transmet pas à la personne qui vous succédera, et ne se défend pas en revue — car la réponse à « pourquoi avez-vous supprimé cette ligne » est « elle semblait fausse ».

Le remède consiste à écrire la règle : **une condition, une source, une gravité et un compte.** Une fois qu'elle existe sous cette forme, elle s'exécute sur le fichier du trimestre suivant, elle s'explique d'elle-même, et le nombre de lignes qu'elle a attrapées devient une ligne du journal de nettoyage.

5.2 Les seuils existent déjà

Vous n'inventez presque jamais une limite. Ce secteur est exceptionnellement bien pourvu en seuils publiés et défendables, et s'en servir est ce qui fait d'une règle une règle plutôt qu'une opinion.

Domaine	Règle	So
PB, 6-59 mois	En dessous de 80 mm ou au-dessus de 220 mm, ce n'est pas une mesure	OM
Score Z poids-taille	Hors de l'intervalle -5 à +5 de la référence	No
Anthropométrie SMART	Signaler les scores Z à plus de 3 ET de la moyenne de l'enquête	Ra
Quantité d'eau	Moins de 15 litres par personne et par jour est sous le minimum	Spl
Collecte d'eau	Un aller-retour de plus de 30 minutes est un service limité, pas élémentaire	Écl
Chlore résiduel libre	0,2 à 2,0 mg/L au point de consommation	Spl
Couverture	Des doses administrées supérieures à la population cible demandent une explication	Gu

Citez la source dans la règle. Un seuil accompagné d'une référence survit à la contestation ; le même chiffre sans référence se rediscute chaque trimestre.

5.3 Des règles comme données, pas comme conditions

Écrivez les règles sous forme d'une table que le code parcourt. Deux raisons, et c'est la seconde qui compte.

La première est évidente — ajouter une règle revient à ajouter une ligne. La seconde est qu'une table de règles peut être **comptée, rapportée et comparée** : vous pouvez afficher combien de lignes chaque règle a attrapées, comparer au trimestre précédent, et coller l'ensemble dans le journal de nettoyage sans le réécrire en prose.

```python
RULES = [
    {
        "id": "muac-range",
        "column": "muac_mm",
        "severity": "error",
        "source": "WHO / CMAM field practice",
        "test": lambda d: d["muac_mm"].between(80, 220) | d["muac_mm"].isna(),
        "message": "MUAC outside 80-220 mm",
    },
    {
        "id": "muac-unit-error",
        "column": "muac_mm",
        "severity": "warning",
        "source": "unit check, cm entered as mm",
        "test": lambda d: ~d["muac_mm"].between(1, 40),
        "message": "MUAC looks like centimetres",
    },
    {
        "id": "age-in-scope",
        "column": "age_months",
        "severity": "warning",
        "source": "CMAM screening protocol, 6-59 months",
        "test": lambda d: d["age_months"].between(6, 59) | d["age_months"].isna(),
        "message": "age outside the screening window",
    },
]
```

```r
RULES <- tibble::tribble(
  ~id,               ~column,      ~severity,  ~source,                              ~
    message,
  "muac-range",      "muac_mm",    "error",    "WHO / CMAM field practice",          "MUAC
    outside 80-220 mm",
  "muac-unit-error", "muac_mm",    "warning",  "unit check, cm entered as mm",       "MUAC
    looks like centimetres",
  "age-in-scope",    "age_months", "warning",  "CMAM screening protocol, 6-59 mo",   "age
    outside the screening window"
)

TESTS <- list(
  `muac-range`      = function(d) dplyr::between(d$muac_mm, 80, 220) | is.na(d$muac_mm),
  `muac-unit-error` = function(d) !dplyr::between(d$muac_mm, 1, 40),
  `age-in-scope`    = function(d) dplyr::between(d$age_months, 6, 59) | is.na(d$age_months
    )
)
```

Chaque test réussit sur une valeur manquante. C'est délibéré et cela prend constamment les gens en défaut : `NA < 80` n'est pas faux, c'est inconnu, et une règle qui traite l'inconnu comme une infraction rapportera votre non-réponse deux fois — une fois comme non-réponse et une fois comme valeur implausible. La non-réponse est le problème de la leçon 2. Gardez les deux séparés.

5.4 Appliquer la table

```python
def apply_rules(df, rules):
    results = []
    flags = pd.DataFrame(index=df.index)
    for rule in rules:
```

```
        ok = rule["test"](df)
        flags[rule["id"]] = ~ok
        results.append({
            "rule": rule["id"],
            "severity": rule["severity"],
            "column": rule["column"],
            "failed": int((~ok).sum()),
            "share": round((~ok).mean(), 4),
            "source": rule["source"],
        })
    return pd.DataFrame(results), flags

report, flags = apply_rules(muac, RULES)
print(report.to_string(index=False))
```

```
apply_rules <- function(df, rules, tests) {
  flags <- purrr::map_dfc(rules$id, function(id) {
    tibble::tibble(!!id := !tests[[id]](df))
  })
  report <- rules |>
    dplyr::mutate(
      failed = purrr::map_int(id, ~ sum(flags[[.x]], na.rm = TRUE)),
      share  = round(failed / nrow(df), 4)
    )
  list(report = report, flags = flags)
}

out <- apply_rules(muac, RULES, TESTS)
out$report
```

Règle	Gravité	Échecs	Part
muac-range	error	7	0,17 %
muac-unit-error	warning	7	0,17 %
age-in-scope	warning	0	0,00 %

Sept lignes échouent aux deux règles sur le PB, ce qui est la réponse souhaitée : les mêmes sept enregistrements sont hors bornes **et** ressemblent à des centimètres, donc il s'agit d'une erreur d'unité et non de sept fautes sans rapport. Deux règles qui concordent constituent un indice sur la cause ; une règle isolée dirait seulement que quelque chose ne va pas.

`age-in-scope` n'attrape rien, et cela mérite d'être conservé aussi. Une règle qui ne se déclenche jamais coûte une ligne et prouve que la contrainte tient — et le jour où elle se déclenche, elle vous dit que le programme a changé.

5.5 Bornes dures et bornes souples

Toutes les règles ne veulent pas dire la même chose, et les confondre est la façon dont un script de nettoyage se met à supprimer de vraies données.

— **Les bornes dures** sont physiquement ou logiquement impossibles. Une taille de ménage négative, un PB de 450 mm, une date de sortie antérieure à l'admission, des doses administrées à plus d'enfants qu'il n'en existe. Ce sont toujours des erreurs.

— **Les bornes souples** sont implausibles mais possibles. Dix litres par personne et par jour est sous le minimum Sphere, et c'est aussi une réalité vécue par de vrais ménages — c'est le *constat*, pas un défaut.

L'enquête EAH montre à quel point il est facile de confondre les deux. Douze ménages déclarent plus de 60 litres par personne et par jour, contre une médiane autour de quinze. Soixante litres par personne n'est pas impossible ; c'est ce que consomme réellement un ménage doté d'un branchement de cour et d'un jardin. Mais dans un fichier où quelques enquêteurs ont noté la consommation **totale** du ménage dans une colonne par personne, c'est aussi exactement l'allure de cette erreur.

```
suspect = wash[wash["litres_per_person_day"] > 60][
    ["household_id", "community", "household_size", "litres_per_person_day", "water_source
    "]
]
suspect["implied_total"] = suspect["litres_per_person_day"] * suspect["household_size"]
print(suspect)
```

```
wash |>
  filter(litres_per_person_day > 60) |>
  mutate(implied_total = litres_per_person_day * household_size) |>
  select(household_id, community, household_size, litres_per_person_day,
         water_source, implied_total)
```

Regardez `implied_total`. Si un ménage de sept personnes est enregistré à 140 litres par personne, le total implicite est de 980 litres par jour transportés à la main, ce qui n'arrive pas. **La règle qui tranche une valeur ambiguë est généralement une deuxième colonne, pas un seuil plus serré.**

Le même piège se loge dans le temps de collecte. Quarante-deux ménages déclarent un aller-retour de moins de six minutes depuis une source qui n'est pas sur leur parcelle. Certains sont un robinet au bout de la rue. D'autres sont un enquêteur qui a écrit des heures dans un champ en minutes. Rien dans la colonne ne les distingue, et la règle honnête signale les quarante-deux pour relecture au lieu de prétendre savoir lesquels sont lesquels.

5.6 Des règles qui comparent deux colonnes

Les règles les plus utiles portent rarement sur la plage d'une seule colonne. Elles portent sur deux colonnes qui doivent concorder.

```
CROSS_RULES = [
    {
        "id": "referral-without-measurement",
        "severity": "warning",
        "test": lambda d: ~(d["outcome"].str.startswith("referred") & d["muac_mm"].isna()),

        "message": "referred but no MUAC recorded",
    },
    {
        "id": "outcome-contradicts-muac",
        "severity": "warning",
        "test": lambda d: ~((d["muac_mm"] >= 125) & (d["oedema"] != True)
                            & (d["outcome"] != "no-action")),
        "message": "no-action expected from the measurement",
    },
]
```

```
CROSS_TESTS <- list(
  `referral-without-measurement` = function(d)
```

```
    !(startsWith(d$outcome, "referred") & is.na(d$muac_mm)),
  `outcome-contradicts-muac` = function(d)
    !(d$muac_mm >= 125 & !d$oedema & d$outcome != "no-action")
)
```

Dix enregistrements portent une décision de référencement sans mesure derrière, et neuf portent une sortie que leur propre mesure contredit. Aucun des deux n'est impossible — un enfant peut être référé sur jugement clinique, et un référencement peut être justifié pour une raison que le registre ne contient pas. Mais les deux **méritent un coup de téléphone**, et aucun n'est repérable en regardant l'une ou l'autre colonne isolément.

5.7 Signalez, ne supprimez pas

La règle ajoute une colonne. Elle ne retire pas de ligne.

```
muac = muac.join(flags.add_prefix("flag_"))
muac["flag_any_error"] = flags[[r["id"] for r in RULES if r["severity"] == "error"]].any(
    axis=1)
```

```
muac <- dplyr::bind_cols(muac, dplyr::rename_with(out$flags, ~ paste0("flag_", .x)))
muac$flag_any_error <- dplyr::if_any(dplyr::starts_with("flag_"), identity)
```

Trois choses que cela vous apporte et qu'un filtre ne donne pas.

— **Le compte reste disponible.** « 4 218 dépistages, 7 exclus pour mesure implausible » exige les deux nombres, et un tableau filtré en a jeté un.

— **L'exclusion est réversible.** Un relecteur en désaccord avec une règle peut relancer l'analyse sans elle, en une ligne.

— **Les signalements sont analysables.** Ce qui nous amène à la dernière section.

Appliquez l'exclusion une seule fois, au moment du calcul, et dites-le.

```
analysis = muac[~muac["flag_any_error"]]
print(f"{len(muac)} screenings, {len(analysis)} analysed, "
      f"{muac['flag_any_error'].sum()} excluded")
```

```
analysis <- muac |> filter(!flag_any_error)
sprintf("%d screenings, %d analysed, %d excluded",
        nrow(muac), nrow(analysis), sum(muac$flag_any_error))
```

5.8 Le taux de signalement est lui-même un indicateur

Une fois que les signalements sont des colonnes, regroupez-les comme n'importe quoi d'autre — et ce qui revient est une affirmation sur la collecte des données plutôt que sur les enfants.

```
print(muac.groupby("commune")[["flag_muac_unit_error", "flag_any_error"]].mean())
```

```
muac |>
  summarise(across(starts_with("flag_"), mean), .by = commune)
```

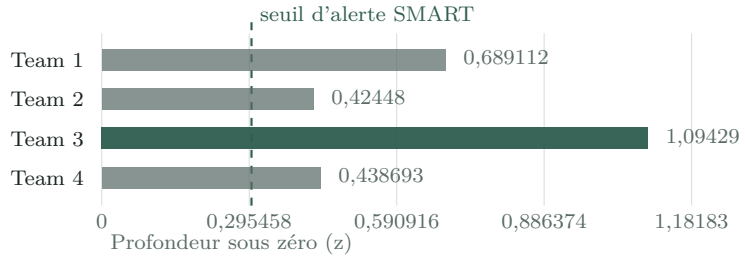

FIGURE 5.1 – Score Z poids-pour-taille moyen par équipe de mesure dans l'enquête SMART, tracé en profondeur sous zéro. Les grappes ayant été attribuées aux équipes indépendamment de l'état nutritionnel, un écart de cette ampleur entre équipes est un défaut de mesure et non une différence réelle entre populations.

Si une commune, une équipe ou un enquêteur concentre l'essentiel d'un signalement, le constat n'est pas dans les données — il est dans la supervision. C'est une chose bien plus utile à mettre dans un rapport mensuel qu'un chiffre corrigé, parce qu'elle est actionnable : l'équipe peut être reformée, et le fichier du trimestre suivant sera meilleur au lieu d'être simplement nettoyé.

> Une règle qui se déclenche partout est une règle sur le monde. Une règle qui se déclenche à un seul endroit est une règle sur une équipe.

5.9 La suite

Toutes les règles vues jusqu'ici comparent une valeur à un nombre. La leçon suivante traite des valeurs qu'il faut comparer à une *liste* — les noms de sites et de villages en texte libre qui arrivent en quatre graphies, où l'autorité est une liste administrative et où le plus dur est de refuser d'inventer un appariement.

CHAPITRE 6

Six districts, trois districts

Leçon 6 sur 8 · 90 min

Des noms de lieux en texte libre face à une liste administrative — normaliser, apparier à l'identique, relire le reliquat, et stocker le résultat dans un fichier de correspondance plutôt que dans une suite de remplacements. Plus l'assertion qui arrête la cinquième graphie du prochain tour.

6.1 Le tableau à six lignes pour trois districts

Regroupez l'enquête ménage EAH par district et voici ce qui revient.

district	Ménages	Sous 15 L/personne/jour
NORD-OUEST	33	21,2 %
nord-ouest	20	20,0 %
Sud-Est	803	16,4 %
Nord-Ouest	727	14,7 %
Nord Ouest	22	9,1 %
Centre	798	8,9 %

Six lignes. Il y a trois districts. Une équipe d'enquêteurs a écrit Nord-Ouest de quatre façons différentes, et comme les variantes se trient séparément, les 802 ménages de ce district sont découpés en morceaux de 727, 33, 22 et 20.

Lisez maintenant le tableau comme le ferait une réunion de coordination. Trié du pire au meilleur, **NORD-OUEST est le district prioritaire à 21,2 %** — un chiffre calculé sur trente-trois ménages, la plus petite cellule du tableau et la plus facilement déplacée par une poignée d'entretiens. Le vrai Nord-Ouest, ses 802 ménages, est à 15,0 % et arrive deuxième derrière Sud-Est.

Personne ne vous dira que cela s'est produit. Rien n'échoue. Le tableau a l'air correct, il répond seulement à une autre question que celle posée.

6.2 La liste administrative fait autorité

La première décision porte sur les noms qui font foi, et la réponse n'est jamais « ceux qui sont dans les données ». Chaque pays dispose d'une hiérarchie administrative officielle — admin 1, admin 2, admin 3 — publiée avec des codes. Dans les opérations humanitaires ce sont les **p-codes**, diffusés par le Humanitarian Data Exchange et utilisés par tous les clusters, et tout leur intérêt est qu'un code n'a pas de variantes orthographiques.

```
admin = pd.DataFrame([
    {"admin1_pcode": "HT07", "admin1_name": "Nord-Ouest"},
    {"admin1_pcode": "HT05", "admin1_name": "Centre"},
    {"admin1_pcode": "HT09", "admin1_name": "Sud-Est"},
])
```

```
admin <- tibble::tribble(
  ~admin1_pcode, ~admin1_name,
  "HT07",        "Nord-Ouest",
  "HT05",        "Centre",
  "HT09",        "Sud-Est"
)
```

Trois conséquences découlent du fait de traiter cette liste comme l'autorité, et elles donnent la forme du reste de la leçon.

— Une valeur des données qui ne se résout pas dans la liste est **non appariée**, pas fausse. Ce peut être une variante d'écriture, une unité administrative nouvelle, ou un lieu réel que la liste ne couvre pas.

— Le produit de l'appariement est un **code**, et tout l'aval joint sur le code.

— C'est la liste, et non les données, qui décide du nombre de districts. Un tableau à six lignes échoue à un contrôle au lieu d'être rapporté.

6.3 Normalisez, puis appariez à l'identique

La plupart des variantes ne sont pas vraiment des noms différents. Retirez ce qui ne porte pas de sens et elles se rejoignent.

```
def match_key(series):
    return (
        series.fillna("")
        .str.normalize("NFKD")
        .str.encode("ascii", "ignore").str.decode("ascii")
        .str.lower()
        .str.replace(r"[^a-z0-9]+", " ", regex=True)
        .str.strip()
    )

wash["district_key"] = match_key(wash["district"])
admin["key"] = match_key(admin["admin1_name"])

matched = wash.merge(
    admin[["key", "admin1_pcode", "admin1_name"]],
    left_on="district_key", right_on="key", how="left",
)
print(matched["admin1_pcode"].isna().sum(), "rows unmatched")
```

```
match_key <- function(x) {
  x |>
    tidyr::replace_na("") |>
    stringi::stri_trans_general("Latin-ASCII") |>
    tolower() |>
    stringr::str_replace_all("[^a-z0-9]+", " ") |>
    stringr::str_squish()
}

wash  <- wash  |> mutate(district_key = match_key(district))
admin <- admin |> mutate(key = match_key(admin1_name))

matched <- wash |> left_join(admin, by = c("district_key" = "key"))
sum(is.na(matched$admin1_pcode))
```

La casse, les accents et la distinction trait d'union / espace expliquent **chacune** des quatre variantes de Nord-Ouest ici. `NORD-OUEST`, `nord-ouest`, `Nord Ouest` et `Nord-Ouest` se normalisent tous en `nord ouest`, et la jointure est exacte. Zéro ligne non appariée.

C'est le cas courant et il vaut la peine de l'intérioriser — l'appariement approximatif vient *après* la normalisation, pas à sa place. Une grande partie de ce qui ressemble à un problème d'appariement difficile est de la ponctuation.

6.4 Le reliquat, c'est le vrai travail

Quand l'appariement exact ne solde pas tout — et sur des noms de villages cela n'arrive jamais — le reste est ce qu'il faut traiter.

```python
unmatched = (
    matched[matched["admin1_pcode"].isna()]
    .groupby("district")
    .size()
    .sort_values(ascending=False)
)
print(unmatched)
```

```r
matched |>
  filter(is.na(admin1_pcode)) |>
  count(district, sort = TRUE)
```

Traitez la liste par nombre de lignes, pas par ordre alphabétique. Les valeurs non appariées forment presque toujours une tête très courte et une longue traîne — deux ou trois variantes couvrant l'essentiel des lignes, puis des cas isolés. Solder la tête prend dix minutes et récupère la majeure partie des données.

Pour la traîne, produisez des candidats plutôt que des décisions.

```python
from rapidfuzz import process, fuzz

for value in unmatched.index:
    best = process.extract(
        match_key(pd.Series([value]))[0],
        admin["key"].tolist(),
        scorer=fuzz.ratio,
        limit=3,
    )
    print(value, "->", best)
```

```r
for (value in unique(unmatched$district)) {
  scores <- stringdist::stringsim(match_key(value), admin$key, method = "jw")
  cat(value, "->", admin$admin1_name[order(-scores)][1:3], "\n")
}
```

6.5 Les appariements qu'il faut refuser

L'appariement approximatif sur des noms de lieux est plus dangereux qu'il n'y paraît, car les unités administratives sont fréquemment nommées les unes d'après les autres, et d'après les mêmes saints, rivières et administrateurs coloniaux. Deux communes réelles et distinctes peuvent différer d'un seul caractère.

Trois habitudes qui évitent l'erreur coûteuse.

— **Contraignez par l'unité parente.** Un village n'a besoin d'être apparié qu'aux villages de sa propre commune, et une commune qu'aux communes de son propre département. Cela élimine gratuitement la plupart des faux candidats et c'est pourquoi on apparie la hiérarchie de haut en bas.

— **Refusez l'acceptation automatique en dessous d'un seuil élevé.** Une quasi correspondance sur un nom de deux syllabes n'est pas une preuve. Lorsque les deux meilleurs candidats sont à quelques points l'un de l'autre, la réponse est « à relire », pas « le premier ».

— **Laissez non apparié ce qui n'est pas apparié.** Une ligne sans district est honnête et apparaît dans le tableau de complétude. Une ligne attribuée au mauvais district est invisible et déplace deux chiffres.

> Le mode de défaillance n'est pas que le script n'arrive pas à apparier. C'est que le script apparie quelque chose, et que le ménage finit compté dans un district où il n'est pas.

6.6 La correspondance est un fichier, pas une suite de remplacements

Le réflexe est un `replace` ou un `case_when`. Résistez-y.

```python
mapping = pd.read_csv("reference/district-mapping.csv")   # raw_value, admin1_pcode,
    decided_by, decided_on
wash = wash.merge(mapping, left_on="district", right_on="raw_value", how="left")
```

```r
mapping <- readr::read_csv(here::here("reference", "district-mapping.csv"))
wash <- wash |> left_join(mapping, by = c("district" = "raw_value"))
```

Un fichier de correspondance l'emporte sur une suite de remplacements pour quatre raisons, et c'est la dernière qui compte vraiment.

— Il est **relisable** par quelqu'un qui ne lit pas le code — en général la personne qui connaît les districts.

— Il est **réutilisable** par le notebook, le tableau de bord et le script du prochain tour.

— Il est **comparable**, si bien qu'ajouter une variante apparaît en revue comme une seule ligne.

— Il **porte qui a décidé**. `decided_by` et `decided_on` transforment une correspondance d'artefact technique en trace, et lorsqu'on demandera en octobre pourquoi `Nord Ouest` a été rattaché à `HT07`, le fichier répond.

6.7 Appariez une fois, joignez sur le code pour toujours

Après l'appariement, retirez le nom en texte libre de tout l'aval et emportez le p-code.

```python
wash = wash.drop(columns=["district", "district_key"]).rename(
    columns={"admin1_pcode": "admin1"}
)
```

```r
wash <- wash |> select(-district, -district_key) |> rename(admin1 = admin1_pcode)
```

C'est l'étape qui fait que le problème reste résolu. Les dénominateurs de population, l'enquête du tour précédent, la matrice 4W du cluster et les données géographiques de la carte se joignent tous sur le code sans jamais recomparer un nom. Deux jeux de données porteurs de p-codes se joignent correctement du premier coup, ce qui est toute la raison d'être de ces codes.

6.8 L'assertion qui attrape la cinquième graphie du prochain tour

Tout ce qui précède nettoie le fichier que vous avez. Une ligne empêche le même défaut d'arriver inaperçu au trimestre suivant.

```python
EXPECTED_DISTRICTS = {"HT05", "HT07", "HT09"}

seen = set(wash["admin1"].dropna())
assert seen <= EXPECTED_DISTRICTS, f"unexpected districts: {seen - EXPECTED_DISTRICTS}"
assert wash["admin1"].isna().sum() == 0, "rows with no district after mapping"
```

```r
EXPECTED_DISTRICTS <- c("HT05", "HT07", "HT09")

stopifnot(
  all(wash$admin1 %in% EXPECTED_DISTRICTS),
  !any(is.na(wash$admin1))
)
```

Remarquez ce que cela fait lorsque le programme s'étend réellement à un quatrième district — cela échoue. C'est correct. Un nouveau district est une chose dont quelqu'un devrait vous informer, et un script qui l'absorbe en silence calculera en silence un taux de couverture contre un dénominateur qui ne l'inclut pas.

6.9 La suite

Vous disposez maintenant de règles pour les valeurs et d'une correspondance pour les noms, appliquées l'une et l'autre à la main ce trimestre. L'unité suivante les fait tourner toutes seules — une suite de validation qui s'exécute à chaque lecture, rapporte les échecs avec leurs comptes, et arrête la chaîne avant qu'un mauvais export n'atteigne un tableau de bord.

Quatrième partie

La trace qui vous survit

CHAPITRE 7

Une validation qui tourne sans vous

Leçon 7 sur 8 · 90 min

Transformer les contrôles en une suite qui s'exécute à chaque lecture — un contrat de schéma, trois gravités, un artefact de rapport, et la décision de savoir quels échecs ont le droit d'arrêter la chaîne.

7.1 Les contrôles que vous avez exécutés ne sont pas ceux que vous avez

Six leçons de contrôles, et chacun d'eux a été exécuté par une personne qui a décidé de l'exécuter. Au trimestre suivant, il se produira l'une de trois choses — vous les relancez et cela prend une après-midi, quelqu'un d'autre en lance certains, ou l'export part directement dans le tableau de bord parce que l'échéance était mardi.

La troisième est l'issue normale. **Les seuls contrôles qui survivent sont ceux qui s'exécutent que quelqu'un y pense ou non**, et cette leçon montre comment y arriver.

7.2 Un contrat, pas un script

Commencez par écrire à quoi ressemble un export valide, séparément du code qui le lit. Cet énoncé est le contrat, et tout le reste en découle.

```python
CONTRACT = {
    "name": "muac-screening",
    "key": ["child_id"],
    "columns": {
        "child_id":       {"type": "string",  "required": True},
        "commune":        {"type": "string",  "required": True, "allowed": COMMUNES},
        "screening_date": {"type": "date",    "required": True,
                           "min": "2024-01-01", "max": "2024-12-31"},
        "age_months":     {"type": "integer", "required": False, "min": 6, "max": 59},
        "sex":            {"type": "string",  "required": True, "allowed": ["f", "m"]},
        "muac_mm":        {"type": "integer", "required": False, "min": 80, "max": 220,
                           "sentinels": [-99]},
        "oedema":         {"type": "boolean", "required": False},
        "outcome":        {"type": "string",  "required": True, "allowed": OUTCOMES},
    },
    "expected_rows": (3500, 5000),
    "max_missing": {"age_months": 0.10, "muac_mm": 0.05},
}
```

```r
CONTRACT <- list(
  name = "muac-screening",
  key  = "child_id",
  columns = list(
    child_id       = list(type = "character", required = TRUE),
    commune        = list(type = "character", required = TRUE, allowed = COMMUNES),
    screening_date = list(type = "Date", required = TRUE,
                          min = as.Date("2024-01-01"), max = as.Date("2024-12-31")),
    age_months     = list(type = "integer", required = FALSE, min = 6, max = 59),
    sex            = list(type = "character", required = TRUE, allowed = c("f", "m")),
```

```
    muac_mm        = list(type = "integer", required = FALSE, min = 80, max = 220,
                          sentinels = -99),
    oedema         = list(type = "logical", required = FALSE),
    outcome        = list(type = "character", required = TRUE, allowed = OUTCOMES)
  ),
  expected_rows = c(3500, 5000),
  max_missing = list(age_months = 0.10, muac_mm = 0.05)
)
```

Deux champs y sont faciles à négliger et ce sont eux qui attrapent les surprises.

`expected_rows` est un intervalle, pas un nombre. Un export de 400 lignes là où vous en attendez quatre mille est un téléchargement tronqué ou un filtre que quelqu'un a laissé actif, et il sera sinon découvert le jour où la charge de cas paraîtra encourageante.

`max_missing` transforme le constat de la leçon 2 en limite. Dix pour cent d'âges manquants est tolérable et documenté ; trente pour cent signifie qu'un formulaire est cassé, et la différence entre les deux n'est pas une chose que l'on souhaite remarquer à l'œil.

7.3 Trois gravités, et une seule arrête l'exécution

C'est la décision de conception qui détermine si la suite sera utilisée ou désactivée.

Gravité	Signification	Effet
`error`	Le fichier n'est pas analysable en l'état	Arrêt. Rien en aval ne s'exécute.
`warning`	Quelque chose ne va pas sur certaines lignes	Les signaler, continuer, rapporter le compte
`note`	Bon à savoir, attendu	Rapport seulement

Une colonne absente est une erreur — tout calcul postérieur est faux. Sept valeurs de PB implausibles sont un avertissement : 4 211 lignes restent analysables, et arrêter la chaîne pour 0,17 % d'entre elles signifie que la chaîne s'arrête tous les mois et que quelqu'un finit par la retirer.

Les erreurs portent sur le fichier. Les avertissements portent sur des lignes. Si vous avez envie de faire d'un contrôle de ligne une erreur, ce que vous voulez en réalité est un seuil sur le nombre de lignes autorisées à échouer — c'est exactement ce qu'est `max_missing`.

7.4 Le validateur

```
from dataclasses import dataclass

@dataclass
class Finding:
    check: str
    severity: str
    count: int
    detail: str

def validate(df, contract):
    findings = []

    missing = set(contract["columns"]) - set(df.columns)
    if missing:
        findings.append(Finding("columns-present", "error", len(missing),
                                f"missing columns: {sorted(missing)}"))
        return findings                       # nothing else is meaningful
```

```
    low, high = contract["expected_rows"]
    if not low <= len(df) <= high:
        findings.append(Finding("row-count", "error", len(df),
                                f"{len(df)} rows, expected {low}-{high}"))

    duplicated = df.duplicated(subset=contract["key"], keep=False)
    if duplicated.any():
        findings.append(Finding("key-unique", "error", int(duplicated.sum()),
                                f"rows sharing a {contract['key']}"))

    for column, spec in contract["columns"].items():
        values = df[column]
        if spec.get("required") and values.isna().any():
            findings.append(Finding(f"{column}-required", "error",
                                    int(values.isna().sum()), "required column has blanks"
    ))
        if "allowed" in spec:
            unexpected = set(values.dropna().unique()) - set(spec["allowed"])
            if unexpected:
                findings.append(Finding(f"{column}-allowed", "error", len(unexpected),
                                        f"unexpected values: {sorted(unexpected)}"))
        if "min" in spec:
            out = (values < spec["min"]) | (values > spec["max"])
            if out.any():
                findings.append(Finding(f"{column}-range", "warning", int(out.sum()),
                                        f"outside {spec['min']}-{spec['max']}"))

    for column, limit in contract["max_missing"].items():
        share = df[column].isna().mean()
        if share > limit:
            findings.append(Finding(f"{column}-missing", "error", int(df[column].isna().
    sum()),
                                    f"{share:.1%} missing, limit {limit:.0%}"))

    return findings
```

```
validate <- function(df, contract) {
  findings <- list()
  add <- function(check, severity, count, detail) {
    findings[[length(findings) + 1]] <<-
      tibble::tibble(check = check, severity = severity, count = count, detail = detail)
  }

  missing <- setdiff(names(contract$columns), names(df))
  if (length(missing)) {
    add("columns-present", "error", length(missing),
        paste("missing columns:", paste(missing, collapse = ", ")))
    return(dplyr::bind_rows(findings))
  }

  if (!dplyr::between(nrow(df), contract$expected_rows[1], contract$expected_rows[2])) {
    add("row-count", "error", nrow(df), sprintf("%d rows, expected %d-%d", nrow(df),
        contract$expected_rows[1], contract$expected_rows[2]))
  }

  dup <- duplicated(df[contract$key]) | duplicated(df[contract$key], fromLast = TRUE)
  if (any(dup)) add("key-unique", "error", sum(dup), "rows sharing a key")

  for (column in names(contract$columns)) {
```

```
    spec <- contract$columns[[column]]
    values <- df[[column]]
    if (isTRUE(spec$required) && any(is.na(values))) {
      add(paste0(column, "-required"), "error", sum(is.na(values)), "required column has
    blanks")
    }
    if (!is.null(spec$allowed)) {
      unexpected <- setdiff(unique(stats::na.omit(values)), spec$allowed)
      if (length(unexpected)) {
        add(paste0(column, "-allowed"), "error", length(unexpected),
            paste("unexpected values:", paste(unexpected, collapse = ", ")))
      }
    }
    if (!is.null(spec$min)) {
      out <- values < spec$min | values > spec$max
      if (any(out, na.rm = TRUE)) {
        add(paste0(column, "-range"), "warning", sum(out, na.rm = TRUE),
            sprintf("outside %s-%s", spec$min, spec$max))
      }
    }
  }

  dplyr::bind_rows(findings)
}
```

Remarquez le **retour anticipé** après une colonne absente. Une fois la forme fausse, tout constat ultérieur est du bruit, et un validateur qui affiche quatre- vingt-dix constats alors que le vrai problème est une colonne renommée a échoué dans sa tâche même s'il a fonctionné.

7.5 Branchez-le sur la lecture, pas sur une cellule de notebook

```
def load_screening(path, contract=CONTRACT):
    df = read_typed(path, contract)
    findings = validate(df, contract)

    errors = [f for f in findings if f.severity == "error"]
    for finding in findings:
        print(f"[{finding.severity}] {finding.check}: {finding.count} - {finding.detail}")

    if errors:
        raise ValueError(f"{len(errors)} validation errors in {path}")
    return df, findings
```

```
load_screening <- function(path, contract = CONTRACT) {
  df <- read_typed(path, contract)
  findings <- validate(df, contract)

  if (nrow(findings)) print(findings, n = Inf)
  if (any(findings$severity == "error")) {
    stop(sprintf("%d validation errors in %s", sum(findings$severity == "error"), path))
  }
  list(data = df, findings = findings)
}
```

Il n'existe désormais aucun moyen de lire ce fichier sans le valider. C'est tout le mécanisme. Une fonction de validation que personne n'appelle est de la documentation ; une fonction de validation placée dans le chargeur est une garantie.

Si vous utilisez une bibliothèque — `pandera` en Python, `pointblank` ou `validate` en R — elle fait la même chose avec moins de code et de meilleurs rapports. Servez-vous-en si vous le pouvez. Si cette leçon l'écrit à la main, c'est que la forme importe plus que l'outil, et la forme est celle-ci — contrat, gravités, constats, chargeur.

7.6 Les constats sont un artefact, comme le profil

```python
import json
from pathlib import Path

Path("outputs/validation").mkdir(parents=True, exist_ok=True)
Path("outputs/validation/muac-2024-q4.json").write_text(
    json.dumps([f.__dict__ for f in findings], indent=2)
)
```

```r
jsonlite::write_json(findings,
  here::here("outputs", "validation", "muac-2024-q4.json"),
  pretty = TRUE
)
```

Enregistrés à côté du profil d'arrivée de la leçon 1, ils vous donnent quelque chose de plus précieux que chacun séparément — **une série**. Trimestre après trimestre, les mêmes contrôles sur le même fichier, et le mouvement des comptes est une tendance de qualité des données que vous pouvez rapporter sans collecter quoi que ce soit de nouveau.

```python
history = pd.concat([
    pd.read_json(p).assign(quarter=p.stem)
    for p in sorted(Path("outputs/validation").glob("*.json"))
])
print(history.pivot_table(index="check", columns="quarter", values="count", fill_value=0))
```

```r
history <- purrr::map_dfr(
  list.files(here::here("outputs", "validation"), full.names = TRUE),
  ~ jsonlite::read_json(.x, simplifyVector = TRUE) |>
      dplyr::mutate(quarter = tools::file_path_sans_ext(basename(.x)))
)

tidyr::pivot_wider(history, id_cols = check, names_from = quarter, values_from = count)
```

Un nombre d'avertissements qui baisse après une visite de formation est la preuve que la formation a servi. C'est un bien meilleur usage de ce travail qu'un tableau propre.

7.7 Ce que coûte l'échec bruyant, et pourquoi il reste juste

Une chaîne qui s'arrête a un coût réel — quelqu'un attend le chiffre, et le voilà qui vous attend. Il faut reconnaître honnêtement qu'il s'agit d'un arbitrage et non d'un gain gratuit.

L'arbitrage est favorable pour une raison. **Un chiffre faux qui est parti coûte plus cher qu'un rapport en retard**, parce que le chiffre faux se fait citer, figure dans une proposition, et se compare au trimestre suivant — et la correction, si elle a lieu un jour, doit le poursuivre à travers tous les documents qu'il a atteints.

La règle est donc : arrêter sur tout ce qui rend le résultat faux, signaler tout ce qui rend certaines lignes fausses, et rendre le message d'échec assez bon pour que la personne qui le

rencontre à 7 heures du matin puisse agir sans vous.

```
[error] commune-allowed: 1 - unexpected values: ['Petite-Riviere']
[error] age_months-missing: 226 - 5.4% missing, limit 5%
ValueError: 2 validation errors in muac-screening-2024-q4.csv
```

Ces deux lignes disent au lecteur quoi faire ensuite. Une `AssertionError` à la ligne 41 ne le dit pas.

7.8 La suite

La suite trouve désormais tout et ne change rien — à dessein, puisque chaque modification jusqu'ici a été un signalement. La dernière leçon porte sur ce qu'il advient de ces signalements : les décisions, qui les a prises, l'effet de chacune sur le chiffre, et le journal qui accompagne le rapport pour que personne n'ait à demander.

CHAPITRE 8

Le journal qui répond à la question avant qu'elle soit posée

Leçon 8 sur 8 · 75 min

Une ligne par règle appliquée — laquelle, combien de lignes touchées, ce qui a changé, qui a validé. Produit par le code plutôt que rédigé après coup, et livré comme l'annexe qui met fin au débat.

8.1 La question qui vous sera posée

Elle arrive sous l'une de trois formes et elle arrive toujours.

— Lors d'une visite de suivi d'un bailleur — *« Votre charge de cas a baissé de 4 % entre le projet et le rapport final. Qu'est-ce qui a changé ? »*

— Par le responsable de programme — *« Le trimestre dernier Gros-Morne était deuxième plus touchée. Elle est maintenant quatrième. Y a-t-il eu une amélioration ? »*

— Par votre successeur, dans un courriel — *« Il y a ici un script qui supprime des lignes. Savez-vous pourquoi ? »*

Chacune trouve réponse en trente secondes avec un journal de nettoyage, et pas du tout sans. Cette leçon est ce journal.

8.2 Un journal est un tableau, pas un récit

Le réflexe est d'écrire un paragraphe dans la section méthodologie. Un paragraphe ne se compte pas, ne se compare pas au trimestre précédent et ne se vérifie pas.

Une ligne par règle appliquée. Chaque ligne porte un compte.

règle	gravité	lignes	action	effet sur l'indicateur	décidé par
muac-range	error	7	exclues	MAG 8,62 % à 8,63 %	R. Cassion
exact-duplicate	error	12	un exemplaire gardé	charge 366 à 360	R. Cassion
re-registration	warning	6	fusionnées après relecture	charge 360 à 354	M. Joseph, super
oedema-yn-coding	warning	25	recodées en false	MAG inchangée	R. Cassion
age-missing	note	226	gardées, signalées	tableau par âge seulement	R. Cassion
district-mapping	warning	75	rattachées à HT07	Nord-Ouest 14,7 % à 15,0 %	A. Pierre, bureau

Lisez la colonne `lignes` de haut en bas et vous avez toute l'histoire de ce fichier. Lisez la colonne `effet sur l'indicateur` et vous avez la réponse à la question du bailleur, par avance.

8.3 Les sept colonnes, et ce qui justifie chacune

— **règle** — l'identifiant de la table de règles de la leçon 5, pour que le journal et le code emploient le même nom. Si le journal dit `muac-range` et le code `implausible_muac`, le lien casse dès qu'un tiers regarde.

— **gravité** — ce que le contrôle signifiait, pour qu'un lecteur puisse survoler jusqu'aux erreurs.

— **lignes** — le compte. Ni « quelques-unes », ni « des valeurs aberrantes ». Un nombre.

— **action** — `exclues`, `corrigées`, `recodées`, `fusionnées`, `gardées et signalées`. Cinq verbes couvrent presque tout, et employer toujours les mêmes cinq rend le journal survolable.

— **effet sur l'indicateur** — l'avant et l'après. C'est la colonne que l'on saute et la seule qu'un bailleur lise.

— **décidé par** — une personne. Pas « l'équipe ». La règle sans nom en face est celle qui se révèle fausse.

— **date** — quand, pour que le journal se lise en regard de la version du résultat qu'il a produite.

8.4 Produisez-le à partir du code

Un journal rédigé après coup, de mémoire, est une fiction, et il est toujours plus court que la vérité. Faites en sorte que chaque étape de nettoyage renvoie sa propre ligne de journal.

```python
LOG = []

def apply_step(df, rule, mask, action, note=""):
    before = indicator(df)
    if action == "exclude":
        out = df[~mask]
    elif action == "flag":
        out = df.assign(**{f"flag_{rule}": mask})
    else:
        raise ValueError(action)
    after = indicator(out)

    LOG.append({
        "rule": rule,
        "rows": int(mask.sum()),
        "action": action,
        "before": before,
        "after": after,
        "effect": round(after - before, 4),
        "note": note,
    })
    return out

muac = apply_step(muac, "muac-range", ~muac["muac_mm"].between(80, 220), "exclude")
muac = apply_step(muac, "age-missing", muac["age_months"].isna(), "flag",
                  "kept; age-disaggregated tables only")

log = pd.DataFrame(LOG)
log.to_csv("outputs/cleaning-log.csv", index=False)
```

```r
LOG <- list()

apply_step <- function(df, rule, mask, action, note = "") {
  before <- indicator(df)
  out <- switch(action,
    exclude = df[!mask, ],
    flag    = dplyr::mutate(df, "flag_{rule}" := mask),
    stop("unknown action: ", action)
```

```
  )
  after <- indicator(out)

  LOG[[length(LOG) + 1]] <<- tibble::tibble(
    rule = rule, rows = sum(mask, na.rm = TRUE), action = action,
    before = before, after = after, effect = round(after - before, 4), note = note
  )
  out
}

muac <- apply_step(muac, "muac-range", !dplyr::between(muac$muac_mm, 80, 220), "exclude")
muac <- apply_step(muac, "age-missing", is.na(muac$age_months), "flag",
                   "kept; age-disaggregated tables only")

readr::write_csv(dplyr::bind_rows(LOG), here::here("outputs", "cleaning-log.csv"))
```

Ce montage garantit deux choses que la seule discipline ne garantit pas.

Une étape ne peut pas avoir lieu sans être consignée, puisque consigner est ce que fait la fonction. Le tentant `df = df[df.muac_mm > 80]` glissé à 18 heures un jour d'échéance est exactement la modification qui n'atteint jamais un journal écrit ; ici, il n'existe pas de manière plus courte de le faire que la manière consignée.

L'effet est mesuré, pas estimé. `before` et `after` proviennent de la même fonction `indicator()`, évaluée sur les mêmes données à un instant d'écart. Personne n'a à se souvenir de ce que valait le chiffre.

8.5 Le tableau avant-après est le livrable

Agrégez le journal en trois lignes et il devient exploitable par un lecteur non technique.

```
print(f"Records received:  {len(raw):>6,}")
print(f"Excluded:          {len(raw) - len(analysis):>6,}")
print(f"Analysed:          {len(analysis):>6,}")
print(f"GAM, raw:          {indicator(raw):>6.1%}")
print(f"GAM, cleaned:      {indicator(analysis):>6.1%}")
```

```
cat(sprintf("Records received: %6d\n", nrow(raw)))
cat(sprintf("Excluded:         %6d\n", nrow(raw) - nrow(analysis)))
cat(sprintf("Analysed:         %6d\n", nrow(analysis)))
cat(sprintf("GAM, raw:         %6.1f%%\n", 100 * indicator(raw)))
cat(sprintf("GAM, cleaned:     %6.1f%%\n", 100 * indicator(analysis)))
```

Si les chiffres brut et nettoyé sont proches, dites-le — c'est la phrase la plus forte dont vous disposiez, car elle signifie que le constat ne dépend pas de votre jugement. S'ils sont éloignés, dites-le aussi, et attendez-vous à la discussion. Un nettoyage qui déplace un indicateur de trois points n'est pas un défaut de l'analyse ; c'est un constat sur la collecte des données, et le dissimuler est la seule version de tout ceci qui relève de la faute.

8.6 Ce qui ne doit pas figurer dans le journal

Le journal est une annexe. Il est envoyé par courriel, joint à des rapports et transféré, ce qui veut dire qu'il voyage plus loin que le jeu de données.

— **Aucun identifiant direct.** Ni noms, ni numéros de téléphone, ni coordonnées GPS. Une ligne disant `fusion de CH03315 dans CH09241` convient, ce sont des pseudonymes ; `fusion`

`de Jean B. (Ti Rivyè) dans...` est un incident de protection.

— **Aucun petit effectif dans une catégorie sensible.** « 3 lignes recodées, catégorie VBG `viol`, Marmelade » identifie des personnes dans une petite commune. Agrégez la règle au niveau que vous publieriez, ou écrivez « 3 lignes, une commune ».

— **Aucun détail en texte libre recopié d'une note de dossier.** La règle est ce que vous avez appliqué, pas ce que l'enregistrement disait.

Cette habitude s'apprend utilement sur des données nutritionnelles, où elle ne coûte rien, afin d'être automatique le jour où l'on vous confie une charge de cas de protection — où les principes ne sont pas des conseils.

8.7 Où vit le journal

À côté du résultat, pas dans un document.

```
outputs/
  profiles/muac-2024-q4.json        arrival profile      (lesson 1)
  validation/muac-2024-q4.json      validation findings  (lesson 7)
  cleaning-log.csv                  decisions            (this lesson)
  review/duplicate-candidates.csv   reviewed and signed  (lesson 4)
  tables/gam_by_commune.csv         the result
  tables/definitions.csv            numerator, denominator, source
```

Ces six fichiers constituent le compte rendu complet d'un trimestre. Quiconque dispose de l'export brut et de ce dossier peut reconstituer tout ce que vous avez fait et vérifier chaque chiffre, et c'est une affirmation bien plus solide que « l'analyse a été soignée ».

Remarquez ce qui **n'y est pas** — le jeu de données nettoyé n'est pas le livrable. Le fichier brut reste en lecture seule et le fichier nettoyé est régénéré en exécutant le script. Un CSV nettoyé qui circule par courriel se détache de son journal en une semaine, et plus personne ne sait quelle version il tient.

8.8 L'annexe dont le rapport a besoin

Trois sections courtes, et elles ont leur place dans toute analyse que ce secteur produit.

Source et complétude des données. D'où vient le fichier, combien d'enregistrements, quelle part des unités de rapportage attendues il couvre, sur quelle période.

Nettoyage et exclusions. Le tableau du journal, ou son résumé avec le tableau complet en pièce jointe. Chaque exclusion avec son compte.

Limites. Ce que le nettoyage n'a pas pu corriger. Le manquant non au hasard que vous soupçonnez sans pouvoir le démontrer. Les six enregistrements fusionnés qu'un superviseur a confirmés et les deux sur lesquels il hésitait. Le district dont le taux repose sur trente-trois ménages.

> L'annexe n'est pas un aveu. C'est la raison pour laquelle le chiffre du résumé exécutif peut être défendu, et la différence professionnelle entre un chiffre et une affirmation.

8.9 Ce que ce cours vous laisse

Vous savez prendre un export brut de programme, décrire honnêtement ce qui est arrivé, y trouver les défauts de quatre manières différentes, décider du traitement de chacun au

regard d'une norme publiée, faire tourner ces décisions à nouveau le trimestre suivant sans vous, et remettre l'ensemble à un auditeur avec la démarche jointe.

Le cours suivant de ce module, *Jointures et restructuration des données de programme*, prend le tableau propre et l'assemble avec les autres — ménage vers membre, registre vers dénominateur de population, registre de formation sanitaire vers agrégat DHIS2 — où le mode de défaillance n'est pas une valeur fausse mais un nombre de lignes qui a doublé pendant que vous le regardiez.

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/data-cleaning-and-validation

Généré le 27 juillet 2026.