CASSION · COURSE

Data Cleaning and Validation

Missing values, duplicates, impossible measurements and inconsistent coding — how to find each one, decide what to do about it, and leave a record a data auditor can follow.

Instructor	Pierre Robentz Cassion
Level	Beginner
Learner hours	16 h
Taught in	Python · R
Updated	July 27, 2026
Sectors	Public Health · Nutrition · WASH
Standards and methodologies	WHO Child Growth Standards · Core Humanitarian Standard (CHS) · Sphere Standards

Read and run the course online at data-analysis.cassion.dev/courses/data-cleaning-and-validation

Course content is licensed CC BY 4.0. You may adapt and redistribute it, including commercially, with attribution to Cassion.

What you will be able to do

- Profile a fresh export before touching it, and keep the profile as the record of what the file looked like on arrival
- Show whether missingness clusters in one site or one week, and state what dropping incomplete rows would do to the ranking you publish
- Prove a table has the key you think it has, and find the duplicate records that share no key at all
- Write plausibility rules from the sector's own thresholds, and separate a recoverable unit error from an unrecoverable one
- Reconcile free-text site and village names against an administrative list without inventing a match
- Ship a cleaning log that states every rule, how many rows it touched, what it changed and who approved it

Prerequisites

None. This course assumes no prior programming experience.

Contents

About this course

Data Analysis Foundations showed you that routine data arrives with defects and gave you five checks for finding them. This course is the next question: **what do you actually do about each one, and how do you prove it afterwards?**

That is a bigger question than it sounds. Almost every cleaning decision moves a published number, most of them move it in a direction that is convenient, and almost none of them are recorded. A district ranking that changes when you drop incomplete rows, a caseload that falls by 2% when you deduplicate, a coverage figure that rises because eleven villages were spelled four ways and three of the spellings were dropped as unmatched — each of those is a defensible decision and an indefensible silence.

So the course is built around one discipline: **every change is a rule, every rule is counted, and the count ships with the number.** By the last unit you will have a validation script that runs on next quarter's export without being edited and a cleaning log an auditor can read without asking you a question.

Every technique is shown in Python and in R. The examples run against platform datasets that carry the defects genuinely — the MUAC screening register from twelve communes in Artibonite, the WASH household survey with a district name written four ways, and the DHIS2-shaped vaccination extract where a facility that did not report and a facility that vaccinated nobody look identical.

You should have done *Data Analysis Foundations*, or be comfortable reading an export into a table and grouping it. You do not need statistics; the one course that needs it says so.

Part I

Before you change anything

CHAPTER 1

The profile you run before you touch it

Lesson 1 of 8 · 90 min

A fresh export is evidence until you edit it. Read it as raw text first, inventory every column, prove or disprove the key, and save the result as the record of what arrived.

1.1 The file is evidence until you edit it

Six months from now someone will ask whether the August figure was always like that. There are two ways that conversation goes. Either you open a file saved on the day the export arrived and read the answer off it, or you reconstruct it from memory and a script that has been edited eleven times since.

So the first thing you do with a new export is not clean it. It is **profile it and write the profile down**, before a single value changes. The profile is cheap — twenty lines — and it is the only artefact that can ever prove what the file looked like on arrival.

This lesson builds that profile. The *Foundations* course taught the five checks that find defects; this one turns them into something you keep.

1.2 Read it as text first

Every reader guesses. `read_csv` looks at the first few thousand rows and decides each column is a number, a date or a string, and every one of those guesses can destroy information you will never get back — a leading zero on a facility code, a sentinel `-99` averaged into a mean, a date read as month-first.

Read once with everything as a string. That read is for looking, not for computing.

```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)
```

Two arguments are doing real work in the Python call. `dtype="string"` stops the type inference; `keep_default_na=False` stops pandas turning the literal strings `NA`, `N/A`, `null` and `nan` into missing values before you have seen them. In this sector that matters more than it sounds — a column where an enumerator typed `NA` for "not applicable" is not the same column as one where the field was skipped, and pandas conflates them by default.

The R read is deliberately `col_character()` for everything, which is also what `problems()` needs to be useful: read strictly later and any value that would not convert is reported rather than silently made `NA`.

1.3 What a profile has to answer

Six questions, and a profile that does not answer all six is not one:

- **How much arrived?** Rows and columns, against what you expected.
- **What is each column really?** Its raw values, not the type a reader guessed.
- **Where are the holes?** Per column, and per site — never one global figure.
- **Is the key a key?** Not "does the column exist" but "is it unique".
- **What is out of range?** Against the sector's limits, not against the data's own.
- **What codes are in use?** Every distinct value of every categorical column.

1.4 Shape, against an expectation you write down

```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)
```

This register has 4,218 rows and eight columns. The useful part is not the count; it is that you wrote down what you expected. A column that quietly disappeared between one export and the next is the most common breaking change a form platform ships, and it is invisible unless something is comparing against a list.

1.5 The column inventory

For every column: how many rows are blank, how many distinct values, and the handful of most frequent ones. This one table replaces most of what people do by scrolling.

```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)
```

Run it on this register and two rows are worth stopping at.

Column	Blank	Distinct	Note
age_months	226	55	5.4% blank — lesson 2 asks where
oedema	44	4	four distinct values in a boolean column

Four distinct values in a boolean column is the finding. The register holds `true`, `false`, an empty string, and `N` — twenty-five rows where an enumerator used `Y`/`N` conventions in a column the form expected `true`/`false` in. Cast that column to a boolean and the `N` rows become missing, silently, and you have lost twenty-five recorded *negatives* by treating them as unrecorded.

That is the whole argument for reading as text first. After the cast there is nothing to find.

1.6 Sentinels are not missing values, yet

`muac_mm` looks complete: no blanks at all. It is not.

```
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")
```

Seventy-two rows carry `-99`, the register's code for "not measured". Read the column as a number without declaring that code and the mean drops by about two millimetres and

the caseload is understated, because you have averaged seventy-two children in at minus ninety-nine.

Sentinel codes are documented in the data dictionary and nowhere else. **Check the dictionary before the first numeric read, every time** — `-99`, `-1`, `999`, `9999` and `88` are all in live use in this sector, and none of them look wrong in a summary.

1.7 Is the key a key?

The column called `child_id` is an identifier. Whether it is *unique* is a separate question, and the answer here is no.

```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))
```

Twenty-four rows across twelve identifiers. Every join you write from here on assumes something about this column, and the assumption is currently false. Unit 2 is about that; the profile's job is only to surface it on day one rather than in the middle of a join that quietly doubles a caseload.

1.8 Ranges, before you trust a summary

A five-number summary hides exactly the values you are looking for, because one implausible row barely moves a quartile. Look at the extremes directly.

```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)
```

The ten smallest values run 13, 14, 14, 15, 16, 17, 18 and then jump to 92. The first seven are centimetres that were never converted — a MUAC of 13.4 cm typed as 13. The mean barely notices them. The tail names them immediately.

Always look at the ten smallest and ten largest values of any measurement column. It costs one line and it is the single highest-yield check in this lesson.

1.9 Save the profile next to the data

The profile is only worth writing if it survives the 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
)
```

JSON rather than a printed table, because the point of the next section is to compare two of them.

1.10 Comparing this export against the last one

Most exports are the same export again, one quarter later. The interesting question is therefore never "what is in this file" but **"what changed"**.

```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)
```

A new value in a categorical column is the single most common way an analysis starts producing wrong answers without failing. A form gets a new response option, a thirteenth commune is added, an antigen is renamed — and every `case_when` written before that day now sends the new value to its `else` branch. Comparing profiles catches it on arrival, which is the only moment it is cheap.

> A profile you did not save is a profile you did not run. The value is entirely in being able to open it later.

1.11 What comes next

You now know this register is missing 5.4% of its ages. That figure on its own is almost meaningless — it matters enormously whether those 226 rows are scattered across twelve communes or concentrated in one. The next lesson breaks missingness down until it either stops looking like an accident or proves it is one, and puts a number on what dropping those rows would do to the ranking you publish.

CHAPTER 2

Missingness that is not an accident

Lesson 2 of 8 · 90 min

A global completeness figure hides the only thing that matters. Break missingness down by site and week, name the mechanism, and put a number on what dropping the incomplete rows does to your ranking.

2.1 5.4% is not a finding

The register is missing `age_months` on 226 of 4,218 rows. Reported as "the dataset is 94.6% complete", that number is worse than useless: it invites the reader to conclude the problem is small, and it is not small, because it is not spread out.

A completeness figure only means something once you know where the holes are. Scattered missingness costs precision. Clustered missingness costs the answer. This lesson is how to tell which one you have, and what to say about it.

2.2 Break it down by group before you do anything else

```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	Missing age	Rows
Gros-Morne	17.7%	361
Anse-Rouge	5.7%	229
Saint-Michel	5.1%	335
Dessalines	4.9%	450
Ennery	3.0%	263

Eleven communes sit between 3% and 6%. One sits at 17.7%. That is not a distribution with a long tail; it is eleven communes with a normal amount of missingness and **one commune with a problem**.

2.3 Then by time, inside the group that stands out

```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()
```

The week beginning 10 June: 54 of 85 screenings, 63.5%. Every other week in that commune is under 10%.

That is no longer a data quality statistic. It is **one team, one week, one tablet form with the age field misconfigured**, and it has a name, a date and probably a person who remembers it. Missingness that resolves to an event is missingness you can ask about, and often fix at source.

2.4 Name the mechanism, in words the report can carry

The statistical literature has three categories. You do not need the notation, but you do need the distinction, because it decides what you are allowed to do.

- **Missing completely at random.** The holes are unrelated to anything, including the value that is missing. Dropping those rows costs precision and nothing else.
- **Missing at random.** The holes depend on something you *observed* — a commune, a week, an enumerator. Dropping them biases the result, but you can adjust for the thing they depend on, because you have it.
- **Missing not at random.** The holes depend on the missing value itself. The sickest children are the ones the queue never reached; the households that refused the water quality test are the ones with the dirtiest water. Nothing in the data can fix this, and the only honest response is to say so.

The age missingness here is **missing at random**: it depends on commune and week, both of which are recorded. The `-99` MUAC codes may be worse than that — if measurement was skipped when a child was distressed, and distress correlates with illness, that is missing not at random, and no amount of code will tell you.

> The category is not a statistical technicality. It is the difference between "we dropped 226 rows" and "we dropped 226 rows, 54 of them from one commune in one week, which moved that commune's rate by 1.1 points".

2.5 Put a number on what dropping them costs

This is the step almost nobody does, and it takes six lines. Compute the indicator twice: once on everything, once on complete cases only.

```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	All rows	Complete cases	Shift
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

Read the first two rows together. On all rows, Gros-Morne is second worst at 14.5% and 1.7 points clear of the commune below it. On complete cases it is 13.4%, and the commune below it is at 13.3%. **The gap between second and third place goes from 1.7 points to 0.1** — which is to say, the ranking you would put in front of a nutrition cluster meeting depends on a form misconfiguration in one commune in June.

Notice also that nothing here crossed a threshold in a way that changed a decision. Say that too. A sensitivity check that finds no effect is a result, and it is the one that lets you stop worrying.

2.6 The three responses, and when each is honest

Drop them, and report the drop. Legitimate when the missingness is scattered and you have shown it is. The reporting is not optional: "n = 3,992 of 4,218; 226 rows excluded for missing age" belongs in the table footnote, not in your head.

Keep them, in a category of their own. Often the right answer for a categorical variable, because "not recorded" is a real finding about the programme. A completeness column beside the indicator column says more than either alone.

Impute — carefully, and rarely. For a variable used to *disaggregate* rather than to compute, filling missing age from the median age of the same commune and month is defensible if you flag every imputed row and report the indicator both ways. For a variable in the numerator, imputation is inventing the finding.

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

```
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()
```

The flag column is the point. An imputed value that is indistinguishable from a measured one has stopped being an estimate and started being a claim.

2.7 Missing rows, not just missing values

The hardest missingness is the kind with no blank cell, because the row is not there at all.

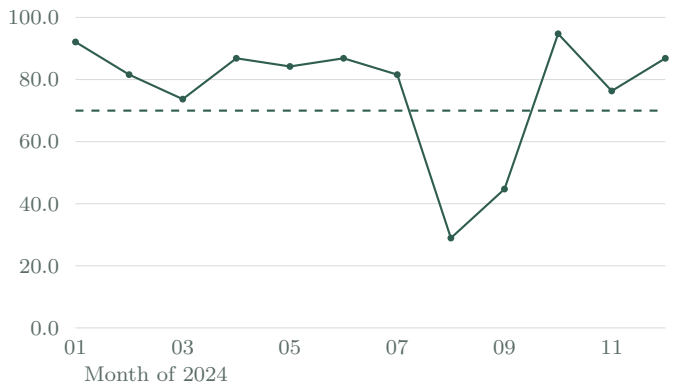

Figure 2.1: Share of health facilities submitting a monthly report through 2024. The collapse across August and September moves together across the district, which is what makes it a reporting event rather than a facility-level failure.

In the routine vaccination extract, a facility that did not report is not absent — it is present with `doses_administered` of zero and `report_submitted` of `false`. Six hundred and forty-two rows are in that state. Compute coverage without separating them and every

non-reporting facility becomes a facility that vaccinated nobody.

```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
  )
```

Two numbers, always published together. **Coverage computed on facilities that reported, and the reporting rate that says how much of the district that covers.** A single coverage figure that quietly includes silent facilities in its denominator is the most common defect in routine health data, and it always points the same way — down.

2.8 What comes next

Missingness is a hole where something should be. The next unit is the opposite problem: something that is there twice. Lesson 3 asks whether the column you have been treating as a key actually is one — and shows what a join does when it is not.

Part II

Identity and duplication

CHAPTER 3

Prove the key before you join

Lesson 3 of 8 · 90 min

Every join makes a claim about uniqueness that nothing checks. Test it, watch two duplicated roster rows fan a 70,245-row table out by 120, and learn to assert the row count instead of inspecting it.

3.1 The assumption nothing checks

You have a student roster and a daily attendance file. You join them on `student_id`, and in doing so you have asserted something: **that `student_id` identifies exactly one row of the roster.**

Nothing in either file says so. No error is raised if it is false. What happens instead is that the join quietly produces more rows than it started with, every downstream count is inflated by an amount nobody can see, and the resulting attendance rate is wrong by an amount too small to look wrong.

This lesson is about testing that claim before making it.

3.2 A key is a claim, and it is testable

The claim has two parts: **unique** and **complete**. A column with duplicates is not a key. A column with missing values is not a key either, because a missing value cannot identify anything.

```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"))
```

On this roster: 1,202 rows, 1,200 distinct `student_id`. **It is not a key**, by two rows.

Two rows out of 1,202 is 0.17% and sounds like it cannot matter. Look at what they are:

```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

Two students transferred school and were never de-registered from the first one. Each now appears **once in a school with a feeding programme and once in a school without** — and the headline analysis on this dataset is whether school feeding improves attendance. Those two students will contribute their entire attendance record to both arms of the comparison.

That is the shape of this defect in general. The duplicate is rarely random; it is usually the record of something that happened — a transfer, a re-registration, a form submitted twice — and it usually lands in exactly the variable the analysis is about.

3.3 Composite keys, and the one you actually have

`student_id` is not a key of the roster. `student_id` plus `school_id` is, because the two rows differ on school. Whether that is the key you *want* is a different question: if the analysis is "one row per student", a composite key that admits two rows per student has documented the problem rather than solved it.

The attendance file is the more usual case, where the natural key is composite from the start:

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

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

70,245 rows, 70,245 distinct pairs. **That is a key.** One row per student per school day, exactly as the file claims to be.

Get in the habit of writing the key down in the code, next to the read:

```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 What a broken key does to a join

The arithmetic is worth doing once by hand, because it explains every fan-out you will ever see.

A join matches every row on the left against every row on the right that shares the key. If a student has 60 attendance rows and 1 roster row, the join produces 60 rows. If that student has 2 roster rows, it produces **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 becomes 70,365. One hundred and twenty extra rows: two students, sixty school days each, counted twice.

A left join that adds rows is a contradiction in terms — "left join" is usually read as "keep the left table and attach columns", and that reading is only true when the right side is unique on the key. It is not a bug in the join. It is the join telling you the truth about your data, in a form nobody looks at.

3.5 Say what you expect, and let it raise

Both languages will check the relationship for you. Use it.

```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")
```

Both **fail loudly** on this data:

```
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`.
```

An error you have to deal with today beats a silently inflated attendance denominator you deal with in a review meeting in November. **Put `validate=` or `relationship=` on every join you write.** It costs one argument and it converts the most expensive class of silent bug into a stack trace.

3.6 Guard the row count when the join is the point

Where the relationship argument does not fit — a many-to-many that genuinely is one — assert the count directly:

```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)
```

The assertion is three words longer than the join. Write it every time.

3.7 The rows on one side and not the other

A key can be unique and still not match. Check both directions before you accept the result of a join:

```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()
```

Here both are zero, which is the answer you want and almost never get. When it is not zero, the two directions mean completely different things:

- **Attendance without a roster row** — a student marked present who is not enrolled. Either the roster is incomplete or someone is being recorded who should not be.
- **Roster rows with no attendance** — enrolled students never marked either way. In an education programme that is not a data defect, it is *the finding*: those are the children who never came.

An inner join makes both groups disappear and reports a clean number. That is why the check goes before the join and not after.

3.8 Columns that were never meant to be keys

Names, phone numbers and household head names get used as keys constantly, because they are the only thing two files have in common. They are not keys, and the failure mode is asymmetric.

- **Duplicates create false merges.** Two households whose head is called Jean Baptiste become one household.
- **Variants create false splits.** `Jean Baptiste`, `JEAN BAPTISTE` and `Jean Baptiste` become three.

If a name is genuinely the only link you have, that is a **record linkage problem**, not a join, and the next lesson is entirely about it. What you must not do is `merge(on="name")` and move on.

3.9 Assert, do not inspect

The pattern this lesson is really teaching is smaller than any of the checks in it. Every one of them can be written as a printout you read once, or as an assertion that runs on every future export. Only the second one survives.

```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)
}
```

> A check you ran once tells you about the file you had. A check that runs on read tells you about the file you have.

Unit 4 turns this pattern into a validation suite. For now, put `assert_key` at the top of every script that joins anything.

3.10 What comes next

`assert_key` finds the students recorded twice under the *same* identifier. It cannot find the child re-registered under a new one, the household interviewed twice by two enumerators, or the beneficiary who appears on three distribution lists with three spellings of their name. Those share no key at all, and the next lesson is how to find them without merging two people who merely resemble each other.

CHAPTER 4

The same person, twice, under two identifiers

Lesson 4 of 8 · 105 min

Record linkage when nothing joins — blocking, normalisation, edit distance and a score. Twelve candidate pairs in the screening register, six of them real, and four manufactured by the missing ages from lesson 2.

4.1 The duplicate with no key

A child screened in the morning, sent home, and brought back in the afternoon by a different carer is registered twice with two identifiers. So is a household visited by two enumerators on the same street. So is a beneficiary on three distribution lists under three spellings of the same name.

None of these share a key. `duplicated()` will never find them. And they matter: in a caseload figure they inflate the numerator, in a coverage figure they inflate the numerator and not the denominator, and in a beneficiary count they are the thing an audit is specifically looking for.

Finding them is **record linkage**, and it is a different discipline from deduplication. Deduplication removes rows that are identical. Record linkage decides whether two rows that are *not* identical describe the same thing — and it can be wrong in both directions, which is why the output of this lesson is a list for a human to review rather than a cleaned table.

4.2 Start with the attributes that should not coincide

The screening register has no name and no household. What it has is a combination that should be near-unique by accident: commune, date, age, sex and measurement.

```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)))
```

Twelve groups come back. Six of them are the same child re-registered under a new identifier. **Six of them are two different children who happen to match.**

4.3 Look at the false ones before the real ones

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

Four of the twelve are from Gros-Morne, in the week beginning 10 June, and every one of them has a **missing age**. That is the same week whose form was misconfigured — the missingness from lesson 2 arriving in a different lesson wearing a different hat.

Here is why it happens. Grouping on a column that is missing makes every missing row match every other missing row on that column. The block collapses: instead of comparing children of the same age, you are comparing all children of unknown age. In a commune screening eighty-five children in a week, two boys with the same MUAC to the millimetre is not surprising at all.

A blocking column with missing values manufactures matches. Either exclude missing rows from the block or block on something else — never let a hole count as an agreement.

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

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

Do that and eight candidates remain: the six real re-registrations and two genuine coincidences in Verrettes and Saint-Marc. **Eight is a list a supervisor can check against the paper register in an afternoon.** Twelve, four of them nonsense, is a list that teaches people to ignore the list.

4.4 Blocking, and why you cannot skip it

Comparing every row against every other row is quadratic. On 4,218 records that is 8.9 million pairs, which is slow but survivable. On a 300,000-record beneficiary registry it is 45 billion, which is not.

Blocking means only comparing records that already agree on something cheap and reliable — the commune, the distribution site, the month, the first letter of the surname. You then do expensive comparisons inside each block only.

```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))
```

The trade is explicit: a block that is too coarse is slow, and a block that is too fine **misses the pairs that disagree on the blocking column**. Two spellings of a village name in different blocks will never be compared. That is why the blocking column should be the one you trust most, and why blocking on a free-text field is usually wrong.

4.5 Normalise before you compare

For anything involving text, most of the work is done before any distance is computed.

```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)
```

Case, accents, double spaces and punctuation account for the large majority of "different" names in this sector's data. Strip them and a surprising share of your fuzzy matching problem becomes exact matching.

Strip accents for the comparison key only, never in the data you keep. `Étienne` is the person's name. `etienne` is an index.

4.6 Edit distance, and choosing a threshold

For what survives normalisation, compare with a string distance. Any of the standard ones will do; what matters is that you pick a threshold deliberately and say what it was.

```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,
)
```

```r
scores <- stringdist::stringsim(
  "jean baptiste pierre",
  households$head_key,
  method = "jw"
```

```
)
which(scores > 0.88)
```

`token_sort_ratio` and Jaro-Winkler are both reasonable defaults here for a reason worth knowing: names in this sector arrive with their parts reordered (`Pierre Jean Baptiste` against `Jean Baptiste Pierre`) and with a typo in the first few characters far less often than in the last few.

The threshold is a **precision-recall dial**, not a setting. At 95 you get few pairs and miss real duplicates. At 80 you get many pairs, most of them wrong, and the reviewer stops reading. Tune it by taking a sample of fifty pairs at your proposed cut-off and checking how many are real. Write the number you chose and the hit rate you measured into the cleaning log.

4.7 Score several fields, do not chain filters

One field is never enough. Compare a handful and combine them, so that a strong agreement on one field can offset a weak one on another.

```
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
}
```

The weights are a judgement, and they should be visible rather than buried in a chain of `and` conditions. A chained filter treats every criterion as absolute: one typo in the community name and the pair is gone. A score lets the evidence add up, which is how a human reviewer reasons about it anyway.

4.8 The output is a review queue, not a clean table

This is the part that separates a linkage that helps from one that causes an 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"))
```

Three empty columns, and they are the point. The file goes to whoever holds the register, comes back with a decision on every row, and **that returned file is what the cleaning script reads** — not the score, and not a threshold applied automatically.

> A duplicate you merged is a record you destroyed. If the merge was wrong, the evidence that it was wrong went with it.

4.9 What you must not automate

In beneficiary, protection and case management data, automatic merging is not a technical shortcut with a small error rate. It is a decision about a person, and the two failure modes are not symmetric.

- **A false merge removes someone.** Two households become one; one of them stops receiving assistance and has no way to find out why. In a protection caseload, two survivors become one file, and one person's history is attached to another person's name.
- **A false split double-counts.** Costly and embarrassing, and it does not deprive anyone of anything.

The asymmetry means the default is to under-merge. Flag, review, and let the person who holds the register decide. Where a merge does happen, keep both original identifiers in the merged record so it can be undone — a linkage table with `kept_id`, `merged_id`, `score`, `reviewer` and `date` beside the data.

None of this is caution for its own sake. The GBV information management principles that govern part of this sector's data make the point directly: the harm from mishandling a record falls on the person in it, not on the analyst.

4.10 What comes next

You now know which rows are the same thing and which columns identify them. The next unit turns to the values themselves — the measurements that cannot be true, the outcome codes that contradict their own measurement, and the sector thresholds that turn "that looks wrong" into a rule a script can apply.

Part III

Values that cannot be true

CHAPTER 5

Rules the sector already gave you

Lesson 5 of 8 · 90 min

Turn "that looks wrong" into a rule table with a source, a severity and a count. Hard bounds against physical limits, soft bounds against the sector's thresholds, and the discipline of flagging rather than deleting.

5.1 "That looks wrong" is not a rule

Everyone doing this work develops an eye. You scroll a column, something snags, you go and look. It works, and it does not scale, does not transfer to the person who takes over from you, and cannot be defended in a review — because the answer to "why did you drop that row" is "it looked wrong".

The fix is to write the rule down: **a condition, a source, a severity and a count.** Once it exists in that form it runs on next quarter's file, it explains itself, and the number of rows it caught becomes a line in the cleaning log.

5.2 The thresholds already exist

You are almost never inventing a limit. This sector is unusually well supplied with published, defensible cut-offs, and using them is what makes a rule a rule rather than an opinion.

Domain	Rule	Source
MUAC, 6-59 months	Below 80 mm or above 220 mm is not a measurement	WHO / CMAM field p
Weight-for-height z	Outside -5 to +5 of the reference	WHO 2006 growth sta
SMART anthropometry	Flag z-scores more than 3 SD from the survey mean	SMART plausibility re
Water quantity	Below 15 litres per person per day is below minimum	Sphere
Water collection	Round trip over 30 minutes is limited, not basic, service	JMP service ladders
Free residual chlorine	0.2 to 2.0 mg/L at the point of consumption	Sphere / WHO
Coverage	Doses administered above the target population needs explaining	UNICEF indicator gui

Cite the source in the rule. A threshold with a citation survives being questioned; the same number without one gets argued about every quarter.

5.3 Rules as data, not as if-statements

Write the rules as a table the code iterates over. Two reasons, and the second is the one that matters.

The first is obvious: adding a rule is adding a row. The second is that a rule table can be **counted, reported and diffed** — you can print how many rows each rule caught, compare against last quarter, and paste the whole thing into the cleaning log without rewriting it in prose.

```
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",
    },
]
```

```
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
    )
)
```

Every test passes on a missing value. That is deliberate and it catches people out constantly: `NA < 80` is not false, it is unknown, and a rule that treats unknown as a violation will report your missingness twice — once as missingness and once as an implausible value. Missingness is lesson 2's problem. Keep the two separate.

5.4 Applying the table

```
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
```

Rule	Severity	Failed	Share
muac-range	error	7	0.17%
muac-unit-error	warning	7	0.17%
age-in-scope	warning	0	0.00%

Seven rows fail both MUAC rules, which is the answer you want: the same seven records are out of range **and** look like centimetres, so they are a unit error rather than seven unrelated mistakes. Two rules agreeing is evidence about the cause; one rule firing alone would only tell you something is wrong.

`age-in-scope` catches nothing, and that is worth keeping too. A rule that never fires costs one line and proves the constraint holds — and the day it starts firing, it is telling you the programme changed.

5.5 Hard bounds and soft bounds

Not every rule means the same thing, and collapsing them is how a cleaning script starts deleting real data.

- **Hard bounds** are physically or logically impossible. A negative household size, a MUAC of 450 mm, a discharge date before an admission date, doses administered to more children than exist. These are always errors.
- **Soft bounds** are implausible but possible. Ten litres per person per day is below the Sphere minimum, and it is also a real thing that happens to real households — that is the *finding*, not a defect.

The WASH survey shows how easy it is to confuse the two. Twelve households report more than 60 litres per person per day, against a median in the teens. Sixty litres per person

is not impossible; it is what a household with a yard connection and a garden actually uses. But in a file where a handful of enumerators recorded the household's **total** consumption in a per-person column, it is also exactly what that error looks like.

```
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)
```

Look at `implied_total`. If a household of seven is recorded at 140 litres per person, the implied total is 980 litres a day carried by hand, which is not happening. **The rule that resolves an ambiguous value is usually a second column, not a tighter threshold.**

The same trap sits in the collection time. Forty-two households report a round trip under six minutes from a source that is not on their plot. Some of those are a tap at the end of the road. Some are an enumerator who wrote hours in a minutes field. Nothing in the column distinguishes them, and the honest rule flags all forty-two for review rather than pretending to know which is which.

5.6 Rules that compare two columns

The highest-value rules are rarely about one column's range. They are about two columns that must agree.

```
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")
)
```

Ten records carry a referral decision with no measurement behind it, and nine carry an outcome their own measurement contradicts. Neither is impossible — a child can be referred on clinical judgement, and a referral can be right for a reason the register does not hold. But both are **worth a phone call**, and neither is findable by looking at either column alone.

5.7 Flag, do not delete

The rule adds a column. It does not remove a row.

```python
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)
```

```r
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)
```

Three things this buys you that a filter does not.

- **The count is still available.** "4,218 screenings, 7 excluded for an implausible measurement" needs both numbers, and a filtered table has thrown one away.
- **The exclusion is reversible.** A reviewer who disagrees with a rule can rerun the analysis without it, in one line.
- **The flags are analysable.** Which brings us to the last section.

Apply the exclusion once, at the point of computation, and say so:

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

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

5.8 The flag rate is itself an indicator

Once flags are columns, group them the way you group anything else — and what comes back is a statement about data collection rather than about children.

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

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

If one commune, one team or one enumerator accounts for most of a flag, the finding is not in the data — it is in the supervision. That is a much more useful thing to put in a monthly report than a corrected number, because it is actionable: the team can be retrained, and next quarter's file will be better rather than merely cleaned.

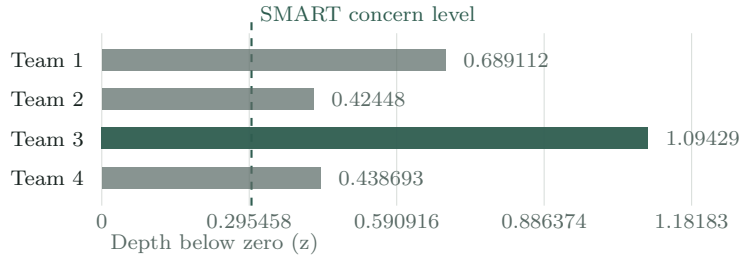

Figure 5.1: Mean weight-for-height z-score by measurement team in the SMART survey, plotted as depth below zero. Clusters were assigned to teams independently of nutrition status, so a spread of this size between teams is a measurement fault rather than a real difference between populations.

> A rule that fires everywhere is a rule about the world. A rule that fires in one place is a rule about a team.

5.9 What comes next

Every rule so far compares a value against a number. The next lesson handles the values that have to be compared against a *list* — the free-text site and village names that arrive in four spellings, where the authority is an administrative list and the hard part is refusing to invent a match.

CHAPTER 6

Six districts, three districts

Lesson 6 of 8 · 90 min

Free-text place names against an administrative list — normalise, match exactly, review what is left, and store the result as a mapping file rather than a chain of replacements. Plus the assertion that stops next round's fifth spelling.

6.1 The table with six rows and three districts

Group the WASH household survey by district and this comes back:

district	Households	Below 15 L/person/day
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 rows. There are three districts. One enumerator team wrote Nord-Ouest four different ways, and because the variants sort apart, that district's 802 households are split into pieces of 727, 33, 22 and 20.

Now read the table the way a coordination meeting would. Sorted worst first, **NORD-OUEST is the priority district at 21.2%** — a figure computed on thirty-three households, which is the smallest cell in the table and the one most easily moved by a handful of interviews. The real Nord-Ouest, all 802 households of it, sits at 15.0% and is second behind Sud-Est.

Nobody will tell you this happened. Nothing errors. The table looks fine, it just answers a different question from the one asked.

6.2 The administrative list is the authority

The first decision is whose names are correct, and the answer is never "the ones in the data". Every country has an official administrative hierarchy — admin 1, admin 2, admin 3 — published with codes. In humanitarian operations these are **p-codes**, distributed through the Humanitarian Data Exchange and used by every cluster, and the whole point of them is that a code does not have spelling variants.

```
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"
)
```

Three consequences follow from treating this list as the authority, and they are the shape of the rest of the lesson.

- A value in the data that does not resolve to the list is **unmatched**, not wrong. It may be a spelling variant, a new administrative unit, or a real place the list does not cover.
- The output of matching is a **code**, and everything downstream joins on the code.
- The list, not the data, decides how many districts there are. A table with six rows fails a check rather than being reported.

6.3 Normalise, then match exactly

Most of the variants are not really different names. Strip what does not carry meaning and they collapse.

```python
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")
```

```r
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))
```

Case, accents and the hyphen-versus-space distinction account for **every one** of the four Nord-Ouest variants here. `NORD-OUEST`, `nord-ouest`, `Nord Ouest` and `Nord-Ouest` all normalise to `nord ouest`, and the merge is exact. Zero rows unmatched.

That is the common case and it is worth internalising: reach for fuzzy matching *after* normalisation, not instead of it. A great deal of what looks like a difficult linkage problem is punctuation.

6.4 What is left over is the actual work

When the exact match does not clear everything — and on village names it never does — the residue is what you work through.

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

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

Work the list by row count, not alphabetically. The unmatched values are almost always a very short head and a long tail: two or three variants covering most of the rows, then singletons. Resolving the head takes ten minutes and recovers most of the data.

For the tail, generate candidates rather than deciding:

```
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)
```

```
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 The matches you must refuse

Fuzzy matching on place names is more dangerous than it looks, because administrative units are frequently named after each other and after the same saints, rivers and colonial officials. Two real, distinct communes can be one character apart.

Three habits that prevent the expensive mistake.

- **Constrain by the parent unit.** A village only needs to be matched against villages in its own commune, and a commune against communes in its own department. This removes most false candidates for free and is why you match the hierarchy top down.

- **Refuse to auto-accept below a high threshold.** A near-match on a two-syllable name is not evidence. Where the top two candidates score within a few points of each other, the answer is "review", not "the first one".
- **Leave unmatched values unmatched.** A row with no district is honest and shows up in the completeness table. A row assigned to the wrong district is invisible and moves two numbers.

> The failure mode is not that the script cannot match. It is that the script matches something, and the household ends up counted in a district it is not in.

6.6 The mapping is a file, not a chain of replacements

The instinct is a `replace` or a `case_when`. Resist it.

```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"))
```

A mapping file beats a chain of replacements on four counts, and the last one is what actually matters.

- It is **reviewable** by someone who does not read code — usually the person who knows the districts.
- It is **reusable** across the notebook, the dashboard and next round's script.
- It is **diffable**, so adding a variant shows up in review as one line.
- It **carries who decided**. `decided_by` and `decided_on` turn a mapping from a technical artefact into a record, and when someone asks in October why `Nord Ouest` was folded into `HT07`, the file answers.

6.7 Match once, join on the code forever

After matching, drop the free-text name from everything downstream and carry the 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)
```

This is the step that makes the problem stay solved. Population denominators, last round's survey, the cluster's 4W matrix and the geodata for the map all join on the code without ever comparing a name again. Two datasets that both carry p-codes join correctly the first time, which is the entire reason the codes exist.

6.8 The assertion that catches next round's fifth spelling

Everything above cleans the file you have. One line stops the same defect arriving unnoticed next quarter.

```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))
)
```

Note what this does when the programme genuinely expands into a fourth district: it fails. That is correct. A new district is a thing someone should tell you about, and a script that silently absorbs it will silently compute a coverage figure against a denominator that does not include it.

6.9 What comes next

You now have rules for values and a mapping for names, and both were applied by hand this quarter. The next unit makes them run by themselves — a validation suite that executes on every read, reports what failed with counts, and stops the pipeline before a bad export reaches a dashboard.

Part IV

The record that survives you

CHAPTER 7

Validation that runs without you

Lesson 7 of 8 · 90 min

Turn the checks into a suite that executes on every read — a schema contract, three severities, a report artefact, and the decision about which failures are allowed to stop the pipeline.

7.1 The checks you ran are not the checks you have

Six lessons of checks, and every one of them was run by a person who decided to run it. Next quarter, one of three things happens: you run them again and it takes an afternoon, someone else runs some of them, or the export goes straight into the dashboard because the deadline was Tuesday.

The third is the normal outcome. **The only checks that survive are the ones that run whether or not anyone remembers them**, and this lesson is how to get there.

7.2 A contract, not a script

Start by writing down what a valid export looks like, separately from the code that reads it. That statement is the contract, and everything else follows from it.

```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)
)
```

Two fields there are easy to skip and are the ones that catch the surprises.

`expected_rows` is a range, not a number. An export with 400 rows where you expect four thousand is a truncated download or a filter someone left on, and it will otherwise be discovered when the caseload looks encouraging.

`max_missing` turns lesson 2's finding into a limit. Ten percent missing age is tolerable and documented; thirty percent means a form is broken, and the difference between those two is not something you want to notice by eye.

7.3 Three severities, and only one of them stops the run

This is the design decision that determines whether the suite gets used or switched off.

Severity	Means	Effect
`error`	The file cannot be analysed as it is	Stop. Nothing downstream runs.
`warning`	Something is wrong with some rows	Flag them, continue, report the count
`note`	Worth knowing, expected to occur	Report only

A missing column is an error: every calculation after it is wrong. Seven implausible MUAC values are a warning: 4,211 rows are still analysable, and stopping the pipeline over 0.17% of them means the pipeline stops every month and somebody removes it.

Errors are about the file. Warnings are about rows. If you find yourself wanting to make a row-level check an error, what you actually want is a threshold on how many rows may fail — which is what `max_missing` is.

7.4 The validator

```python
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)
}
```

Note the **early return** after a missing column. Once the shape is wrong, every subsequent finding is noise, and a validator that prints ninety findings when the real problem is one renamed column has failed at its job even though it worked.

7.5 Wire it into the read, not into a notebook cell

```
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)
}
```

There is now no way to read this file without validating it. That is the whole mechanism. A validation function nobody calls is documentation; a validation function inside the loader is a guarantee.

If you use a library — `pandera` in Python, `pointblank` or `validate` in R — they do the

same thing with less code and better reporting. Use one if you can. The reason this lesson writes it out by hand is that the shape matters more than the tool, and the shape is: contract, severities, findings, loader.

7.6 The findings are an artefact, like the profile

```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
)
```

Saved alongside the arrival profile from lesson 1, these give you something worth more than either alone: **a series**. Quarter on quarter, the same checks against the same file, and the movement in the counts is a data quality trend you can report without collecting anything new.

```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)
```

A warning count that falls after a training visit is evidence the training worked. That is a much better use of this work than a clean table.

7.7 What "fail loudly" costs, and why it is still right

A pipeline that stops has a real cost: someone is waiting for the number, and now they are waiting for you. It is worth being honest that this is a trade rather than a free win.

The trade is favourable for one reason. **A wrong number that shipped is more expensive than a report that is late**, because the wrong number gets quoted, put in a proposal, and compared against next quarter — and the correction, if it ever happens, has to chase it through every document it reached.

So the rule is: stop on anything that makes the output wrong, flag anything that makes some rows wrong, and make the failure message good enough that whoever hits it at 7 a.m. can act on it without you.

```
[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
```

Both of those tell the reader what to do next. `AssertionError` on line 41 does not.

7.8 What comes next

The suite now finds everything and changes nothing — by design, because every change so far has been a flag. The last lesson is what happens to those flags: the decisions, who made them, what each one did to the number, and the log that travels with the report so nobody has to ask.

CHAPTER 8

The log that answers the question before it is asked

Lesson 8 of 8 · 75 min

One row per rule applied — what it was, how many rows it touched, what changed, who approved it. Generated from the code rather than written afterwards, and shipped as the annex that ends the argument.

8.1 The question you will be asked

It arrives in one of three forms and it always arrives.

- A donor's monitoring visit: *"Your caseload fell 4% from the draft to the final report. What changed?"*
- The programme manager: *"Last quarter Gros-Morne was second worst. Now it is fourth. Did something improve?"*
- Your successor, in an email: *"There is a script here that drops rows. Do you know why?"*

Every one of them is answerable in thirty seconds with a cleaning log and not at all without one. This lesson is the log.

8.2 A log is a table, not a narrative

The instinct is to write a paragraph in the methodology section. A paragraph cannot be counted, cannot be diffed against last quarter, and cannot be checked.

One row per rule applied. Every row carries a count.

rule	severity	rows	action	effect on the indicator	decided by
muac-range	error	7	excluded	GAM 8.62% to 8.63%	R. Cassion
exact-duplicate	error	12	one copy kept	caseload 366 to 360	R. Cassion
re-registration	warning	6	merged after review	caseload 360 to 354	M. Joseph, supervisor
oedema-yn-coding	warning	25	recoded to false	GAM unchanged	R. Cassion
age-missing	note	226	kept, flagged	age-disaggregated table only	R. Cassion
district-mapping	warning	75	mapped to HT07	Nord-Ouest 14.7% to 15.0%	A. Pierre, field office

Read down the `rows` column and you have the whole story of what happened to this file. Read down `effect on the indicator` and you have the answer to the donor's question, in advance.

8.3 The seven columns, and why each earns its place

- **rule** — the identifier from the rule table in lesson 5, so the log and the code use the same name. If the log says `muac-range` and the code says `implausible_muac`, the link is broken the first time someone else looks.
- **severity** — what the check meant, so a reader can skim to the errors.

- **rows** — the count. Not "a few", not "some outliers". A number.
- **action** — `excluded`, `corrected`, `recoded`, `merged`, `kept and flagged`. Five verbs cover almost everything, and using the same five makes the log scannable.
- **effect on the indicator** — the before and after. This is the column people skip and the only one a donor reads.
- **decided by** — a person. Not "the team". The rule with no name against it is the rule that turns out to be wrong.
- **date** — when, so the log can be read against the version of the output it produced.

8.4 Generate it from the code

A log written afterwards from memory is fiction, and it is always shorter than the truth. Make each cleaning step return its own log row.

```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"))
```

Two things this arrangement guarantees that discipline alone does not.

A step cannot happen without being logged, because logging is what the function does. The tempting one-line `df = df[df.muac_mm > 80]` inserted at 6 p.m. on a deadline is exactly the change that never reaches a written log; here there is no shorter way to do it than the logged way.

The effect is measured, not estimated. `before` and `after` come from the same `indicator()` function, evaluated on the same data a moment apart. Nobody has to remember what the number was.

8.5 The before-and-after table is the deliverable

Aggregate the log into three lines and it becomes something a non-technical reader can act on:

```
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)))
```

If the raw and cleaned figures are close, say so — it is the strongest sentence available to you, because it means the finding does not depend on your judgement. If they are far apart, say that too, and expect the conversation. A cleaning process that moves an indicator by three points is not a defect in the analysis; it is a finding about data collection, and hiding it is the only version of this that is misconduct.

8.6 What must not be in the log

The log is an annex. It gets emailed, attached to reports and forwarded, which means it travels further than the dataset does.

- **No direct identifiers.** Not names, not phone numbers, not GPS coordinates. A log row reading `merged CH03315 into CH09241` is fine because those are pseudonyms; `merged Jean B. (Ti Rivyè) into...` is a protection incident.

- **No small cells in a sensitive category.** "3 rows recoded, GBV case category `rape`, Marmelade" identifies people in a small commune. Aggregate the rule to the level you would publish, or say `3 rows, one commune`.
- **No free-text detail copied from a case note.** The rule is what you applied, not what the record said.

The habit is worth learning on nutrition data, where it costs nothing, so that it is automatic on the day you are handed a protection caseload — where the principles are not advice.

8.7 Where the log lives

Beside the output, not in a 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
```

Those six files are a complete account of one quarter. Someone with the raw export and this folder can reconstruct everything you did and check every number, and that is a much stronger claim than "the analysis was careful".

Note what is **not** there: the cleaned dataset is not the deliverable. The raw file stays read-only and the cleaned file is regenerated by running the script. A cleaned CSV emailed around detaches from its log within a week and then nobody can tell which version anyone is holding.

8.8 The annex the report needs

Three short sections, and they belong in every analysis this sector produces.

Data source and completeness. Where the file came from, how many records, what share of expected reporting units it covers, over what period.

Cleaning and exclusions. The log table, or a summary of it with the full table attached. Every exclusion with its count.

Limitations. What the cleaning could not fix. The missing-not-at-random you suspect but cannot demonstrate. The six merged records a supervisor confirmed and the two they were unsure about. The district whose rate rests on thirty-three households.

> The annex is not a confession. It is the reason the number in the executive summary can be defended, and the professional difference between a figure and a claim.

8.9 Where this course leaves you

You can take a raw programme export, describe honestly what arrived, find what is wrong with it in four different ways, decide what to do about each fault against a published standard, make those decisions run again next quarter without you, and hand the whole thing to an auditor with the working attached.

The next course in this module, *Joining and Reshaping Programme Data*, takes the clean table and puts it together with the others — household to member, register to population denominator, facility register to DHIS2 aggregate — where the failure mode is not a wrong value but a row count that doubled while you were looking at it.

About this edition

This PDF is generated from the same source as the web version of the course, so the two cannot drift. The web edition carries the runnable code, the downloadable datasets and any corrections issued since this file was produced.

> Every dataset referenced in this course is synthetic or fully de-identified. No record traces to a real person.

This course is published in English and in French. The French edition is a professional adaptation using the terminology UNICEF, WHO, WFP, OCHA and national ministries publish in — not a literal translation.

Read and run the course online at data-analysis.cassion.dev/courses/data-cleaning-and-validation
Course content is licensed CC BY 4.0. You may adapt and redistribute it, including commercially, with attribution to Cassion.
Generated July 27, 2026.