CASSION · COURSE

Data Analysis Foundations for M&E

Load, clean and summarise routine programme data in both Python and R, then turn it into the indicator tables a donor report actually needs.

Instructor	Pierre Robentz Cassion
Level	Beginner
Learner hours	16 h
Taught in	Python · R
Updated	July 26, 2026
Sectors	Public Health · Nutrition · Emergency Response
Standards and methodologies	Logical Framework Approach · Results-Based Management (RBM) · WHO Child Growth Standards

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

What you will be able to do

- Read programme exports from CommCare, KoboToolbox and DHIS2 into a tidy table without corrupting identifiers, dates or missing-value codes
- Diagnose and document the data quality problems routine data always carries, and quantify what each one costs the final number
- Decide what to do about a defect and record that decision in a cleaning log a data auditor can follow
- Produce a disaggregated indicator table that matches a logframe definition, with numerator and denominator stated explicitly
- Reproduce the same analysis in Python and in R, and rerun it on next quarter's export without editing it

Where this sits in the programme

Foundations — Get from a raw programme export to a table you can compute on, in whichever of Python or R your team already uses.

Prerequisites

None. This course assumes no prior programming experience.

Contents

About this course

Routine programme data rarely arrives clean. This course starts from the exports you already have — a CommCare form dump, a KoboToolbox CSV, a DHIS2 pivot — and works through to an indicator table you can defend in a donor review.

Every technique is taught twice, once in Python and once in R, because M&E teams are split across both and you will inherit whichever the last officer used.

Almost every example runs against one dataset: a synthetic MUAC screening register from twelve communes in Artibonite, Haiti, carrying the missing values, unit errors, duplicate registrations and contradictory outcome codes a real campaign register carries. Working the same file from raw export to defended indicator is the point — you see what each decision costs, because you see the number move.

The course assumes no prior programming experience. It does assume you have had to answer for a number in front of someone who did not want to hear it.

CHAPTER 1

Where programme data comes from

Lesson 1 of 8 · 45 min

The shape of exports from CommCare, KoboToolbox and DHIS2, why each one fights you in a different way, and the questions to ask before you write a line of code.

1.1 The three exports you will meet

Most M&E officers inherit data from one of three systems. Each has a house style, and knowing it saves an afternoon.

CommCare exports one row per form submission, with repeat groups flattened into numbered columns (`child_1_muac`, `child_2_muac`, and so on up to whatever the widest form in the dataset needed). The flattening is the first thing you undo. It also means the column count changes between exports: a month where one household had nine children produces nine sets of columns, and next month's file has seven.

KoboToolbox exports close to the XLSForm structure, with repeats in separate sheets joined by `_index` and `_parent_index`. This is more honest than the CommCare shape but it means an analysis that ignores the second sheet silently analyses households instead of people.

DHIS2 exports aggregate data — already summed by period and org unit — so the question is rarely "how do I clean this" and usually "what exactly was counted". You cannot recover an individual from a DHIS2 pivot, and you cannot recompute a denominator that was fixed upstream of you.

1.2 The unit of observation decides everything

Before anything else, answer one question: **what is one row?**

Export	One row is	The trap
CommCare form dump	One form submission	A form may cover several children
KoboToolbox main sheet	One interview	Members live in the repeat sheet
KoboToolbox repeat sheet	One member	No household context until you join
DHIS2 pivot	One org unit × period × data element	Individuals are gone for good

Getting this wrong produces an analysis that is internally consistent and completely wrong. A screening register where one row is a *submission* rather than a *child* will report a caseload that is too low by exactly the number of multi-child households, and nothing in the output will look odd.

> Ask the question out loud before you open the file, and write the answer at the top of your script as a comment. Most of the disputes later in this course trace back to two people holding different answers to it.

1.3 What a real export looks like

This course works against a synthetic MUAC screening register from Artibonite, Haiti. One row is one child screened. It looks like this:

```
child_id,commune,screening_date,age_months,sex,muac_mm,oedema,outcome
CH00854,Terre-Neuve,2024-01-15,22,m,133,false,no-action
CH01439,Verrettes,2024-01-15,16,f,143,false,no-action
```

Eight columns, 4,218 rows, one year of community mass screening across twelve communes. It is small enough to read and messy enough to teach — it carries missing values coded as `-99`, measurements left in centimetres, duplicate registrations, and referral decisions that contradict the measurement recorded next to them.

None of that is decoration. Every defect in this file is one that appears in real campaign registers, put there deliberately so you meet it here rather than the week before a report is due.

1.4 Mid-upper arm circumference, in one paragraph

Mid-upper arm circumference (MUAC) is a screening measurement taken with a colour-coded tape around a child's upper arm. It is used because it needs no scales, no height board and about forty seconds of training. For children from 6 to 59 months the WHO thresholds are:

- Below 115 mm — severe acute malnutrition (SAM)
- 115 mm to under 125 mm — moderate acute malnutrition (MAM)
- 125 mm and above — no acute malnutrition by this measure

Bilateral pitting oedema is SAM regardless of the measurement. That is why the `oedema` column exists and why an analysis that filters on `muac_mm` alone understates the caseload.

Global acute malnutrition (GAM) is SAM plus MAM. You will compute it in lesson 7, and the argument about its denominator is lesson 8.

1.5 The three questions to ask the person who sent you the file

Routine data arrives without a codebook far more often than with one. These three questions recover most of what a codebook would have told you.

1. **What does a blank mean here?** Not measured, measured as zero, refused, or the tablet crashed? These are four different things and they are frequently all stored as an empty cell.
2. **Has anything already been cleaned or dropped?** If someone removed the rows they thought were wrong, your row count is not the caseload and your completeness rate is a fiction.
3. **What period does this cover, and by which date?** Date of screening, date of form submission and date of sync can differ by weeks on a poor connection. A monthly report built on the wrong one moves cases between months.

You will not always get answers. Writing down that you asked, and what you assumed in the absence of an answer, is what separates an analysis that can be defended from one that cannot.

1.6 What comes next

The next lesson sets up a working environment in Python and in R — deliberately one that works offline, because much of this audience is on a connection that cannot be relied on to install a package at the moment it is needed.

CHAPTER 2

A working environment in Python and R

Lesson 2 of 8 · 60 min

Install Python and R so they still work on a bad connection, lay out a project directory that survives a handover, and run the same first script in both languages.

2.1 Why this lesson is not optional

The most common reason an M&E analysis stops being reproducible is not a statistical error. It is that the person who wrote it installed a package one afternoon, never recorded which one, and left. Six months later the script raises an import error on a laptop in a field office with a satellite link, and the quarterly report goes back to being made by hand in a spreadsheet.

Ten minutes of setup now is what prevents that.

2.2 Python: use uv, not the system interpreter

Your operating system ships a Python. Do not analyse data with it — it is there to run the operating system, and changing it can break things you did not know depended on it.

`uv` installs an isolated Python and resolves packages fast enough to be usable on a slow connection. It also writes a lock file, which is what makes the environment reproducible on someone else's machine.

```
# Install uv (macOS and Linux)
curl -LsSf https://astral.sh/uv/install.sh | sh

# Create a project and pin an interpreter
uv init muac-analysis
cd muac-analysis
uv python install 3.13

# Add what this course uses
uv add pandas pyarrow matplotlib
```

`uv add` writes both `pyproject.toml` (what you asked for) and `uv.lock` (exactly what was installed, down to the hash). Commit both. A colleague then reproduces your environment with one command:

```
uv sync
```

If you are behind a proxy or working offline, `uv` can install from a local wheel directory. Downloading the wheels once, onto a USB stick, and installing from it is a normal workflow in this sector and is worth setting up before you need it.

2.3 R: use Projects and renv

R's equivalent problem is the global library — packages installed into one shared location, invisibly shared across every analysis, and silently upgraded under you.

```r
# Once per machine
install.packages("renv")

# Once per analysis, from inside an RStudio Project
renv::init()

# Add what this course uses
install.packages(c("readr", "dplyr", "tidyr", "ggplot2", "janitor"))

# Record exactly what is installed
renv::snapshot()
```

`renv.lock` is the equivalent of `uv.lock`. Commit it. A colleague runs `renv::restore()` and gets your library, not theirs.

Two habits matter as much as the tooling:

- **Work inside a Project.** The working directory is then the project root, for everyone, always. Paths like `data/raw/muac.csv` work on any machine.
- **Never write `setwd()`.** It encodes your username into the analysis and guarantees it fails for the next person.

2.4 A project layout that survives a handover

The same structure works in both languages. It is worth adopting exactly, because the value is in it being predictable rather than in it being clever.

```
muac-analysis/
  data/
    raw/          # exactly as it arrived. Read-only. Never edited.
    interim/      # intermediate output. Disposable.
    processed/    # analysis-ready. Regenerable from raw + code.
  scripts/
    01-read.py
    02-clean.py
    03-indicators.py
  output/
    figures/
    tables/
  README.md
```

One rule carries most of the weight: **`data/raw/` is read-only.** If you open the export in Excel, fix three cells and save it, you have destroyed the only record of what actually arrived, and nobody — including you — can ever again tell what was original and what was a correction.

Everything downstream of `data/raw/` should be reproducible by deleting it and rerunning the scripts. If it is not, something is being done by hand, and that something is the part that will break.

> A useful test: delete `data/processed/` and `output/` entirely, rerun your scripts, and see whether you get the same results. If you cannot bring yourself to try it, you already know the answer.

2.5 The same first script, in both languages

Read the register, look at its shape, and print the first few rows. Nothing more — the point is to confirm the environment works before you depend on it.

```python
import pandas as pd

muac = pd.read_csv("data/raw/muac-screening-artibonite-2024.v1.csv")

print(muac.shape)
print(muac.head())
print(muac.dtypes)
```

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

muac <- read_csv("data/raw/muac-screening-artibonite-2024.v1.csv")

dim(muac)
head(muac)
glimpse(muac)
```

Both should report 4,218 rows and 8 columns. If they do not, stop here — a row count that disagrees with the source is the cheapest error you will ever catch, and it gets much more expensive three scripts later.

2.6 What the two languages disagree about

You will use both, so it helps to know where they differ in ways that matter.

Concern	pandas	dplyr / readr
Missing value	`NaN`, and `None` for objects	`NA`, typed per column
Reading a CSV	`pd.read_csv` guesses types	`read_csv` guesses and reports
Chaining steps	Method chaining or reassignment	The pipe, `|>`
Grouped summary	`groupby().agg()`	`group_by() |> summarise()`
Categories	`category` dtype, opt-in	Factors, and they bite

Neither is better. The one that matters is the one your team already runs, and the reason this course teaches both is that you will change teams.

2.7 What comes next

The next lesson reads the export properly — which is harder than `read_csv(path)`, because the defaults will quietly damage identifiers, dates and the missing-value code before you have looked at anything.

CHAPTER 3

Reading an export without corrupting it

Lesson 3 of 8 · 75 min

Type inference, leading zeros, dates, and the missing-value code that becomes a number. The damage happens at import, before you have looked at anything.

3.1 The damage happens before you look

`read_csv(path)` is one line and it makes half a dozen decisions on your behalf. Most are right. The ones that are wrong are wrong silently, and by the time you notice, the corrupted value has been in three summaries and a chart.

This lesson is about taking those decisions back.

3.2 Missing-value codes become numbers

The MUAC register codes missing measurements as `-99`, not as a blank cell. There is a reason for it — a blank on a paper register is ambiguous between "not measured" and "the enumerator skipped the page", and a sentinel is unambiguous — but a CSV reader has no way to know that `-99` is not a measurement.

```python
import pandas as pd

naive = pd.read_csv("data/raw/muac-screening-artibonite-2024.v1.csv")
print(naive["muac_mm"].mean())
```

That number is several millimetres below the truth. Worse, it is *plausible*: nothing about it looks like an error, and a MUAC mean is not a figure most readers can sanity-check by eye.

Declare the sentinel at read time and it never enters a calculation:

```python
muac = pd.read_csv(
    "data/raw/muac-screening-artibonite-2024.v1.csv",
    na_values={"muac_mm": ["-99"]},
)

print(muac["muac_mm"].mean())
print(muac["muac_mm"].isna().sum())
```

```r
library(readr)

muac <- read_csv(
  "data/raw/muac-screening-artibonite-2024.v1.csv",
  na = c("", "NA", "-99")
)

mean(muac$muac_mm, na.rm = TRUE)
sum(is.na(muac$muac_mm))
```

Note the difference. pandas lets you scope the sentinel to one column, which is what you want: `-99` is a missing MUAC, but if a column ever legitimately held a negative value, blanket

treatment would destroy it. R's `read_csv` applies `na` across the file, so scope it afterwards when that matters.

> Handling a sentinel is not the same as deciding what to do about the missing value. That decision is lesson 6. Here you are only making sure the code stops pretending to be a measurement.

3.3 Identifiers are not numbers

`child_id` in this register looks like `CH00854` and survives import intact. Many real registers use bare numeric identifiers instead — `00854` — and a reader will helpfully turn that into the integer `854`.

The leading zero is gone, the join to the household file fails for exactly the identifiers that had one, and the failure looks like missing households rather than like a type error.

```python
muac = pd.read_csv(
    path,
    dtype={"child_id": "string", "commune": "string"},
    na_values={"muac_mm": ["-99"]},
)
```

```r
muac <- read_csv(
  path,
  col_types = cols(
    child_id = col_character(),
    commune  = col_character()
  ),
  na = c("", "NA", "-99")
)
```

The rule generalises: **if you will never do arithmetic on it, it is not a number.** Facility codes, phone numbers, cluster identifiers, household numbers and administrative codes are all text that happens to be written with digits.

3.4 Dates

`screening_date` arrives as `2024-01-15`. That is ISO 8601, it is unambiguous, and both readers will parse it correctly. Be aware that you are lucky.

Real exports produce `15/01/2024`, `01/15/2024`, `15-Jan-24` and Excel serial numbers like `45306`, sometimes in the same column, because three people entered data on three differently-configured machines. `01/02/2024` is either 1 February or 2 January and the file will not tell you which.

```python
muac["screening_date"] = pd.to_datetime(
    muac["screening_date"], format="%Y-%m-%d", errors="raise"
)
```

```r
muac <- muac |>
  dplyr::mutate(
    screening_date = as.Date(screening_date, format = "%Y-%m-%d")
  )
```

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

Two habits are worth forming:

- **State the format explicitly** rather than letting the parser infer it. An inferred format can change between files as the mix of values changes.
- **Use `errors="raise"`.** The alternative, `errors="coerce"`, converts every unparseable date to missing — which means a file in the wrong date format imports "successfully" with an entirely empty date column.

3.5 Categories with a fixed vocabulary

`outcome` takes four values and `sex` takes two. Declaring them as categorical gives you two things: a smaller object, and — more useful — an error when a value outside the vocabulary appears.

```
outcomes = pd.CategoricalDtype(
    ["no-action", "referred-tsfp", "referred-otp", "referred-sc"], ordered=False
)

muac["outcome"] = muac["outcome"].astype(outcomes)

# Anything outside the vocabulary is now NaN -- count it before moving on.
print(muac["outcome"].isna().sum())
```

```
muac <- muac |>
  dplyr::mutate(
    outcome = factor(
      outcome,
      levels = c("no-action", "referred-tsfp", "referred-otp", "referred-sc")
    )
  )

sum(is.na(muac$outcome))
```

This is a genuine trap in both languages: a value outside the declared levels becomes missing rather than raising. Always count the missing immediately after converting. A jump from zero to nine means nine rows carried a value you did not know about, and you want to see it now rather than discover it as a gap in a table.

The `oedema` column in this register is exactly that case. Ennery and Desdunes recorded it inconsistently in the first quarter, using `Y` and `N` rather than `true` and `false`. Read it naively and those rows land as missing without comment.

3.6 A defensive read, end to end

Putting it together — and asserting what you expect, so the script fails on a bad file instead of producing a bad number.

```
import pandas as pd

PATH = "data/raw/muac-screening-artibonite-2024.v1.csv"

muac = pd.read_csv(
    PATH,
```

```
    dtype={"child_id": "string", "commune": "string", "sex": "string"},
    na_values={"muac_mm": ["-99"]},
    keep_default_na=True,
)

muac["screening_date"] = pd.to_datetime(
    muac["screening_date"], format="%Y-%m-%d", errors="raise"
)

assert len(muac) == 4218, f"expected 4218 rows, got {len(muac)}"
assert muac["child_id"].notna().all(), "every row needs an identifier"

print(muac.dtypes)
print(muac.isna().sum())
```

```
library(readr)
library(dplyr)

PATH <- "data/raw/muac-screening-artibonite-2024.v1.csv"

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

stopifnot(nrow(muac) == 4218)
stopifnot(!any(is.na(muac$child_id)))

glimpse(muac)
colSums(is.na(muac))
```

The assertions are the part people skip and the part that pays. A file that arrives with 4,190 rows instead of 4,218 has lost something between the server and you, and the script should say so rather than quietly report a smaller caseload.

3.7 Reading the other formats

The same principles apply, with different function names.

Source	Python	R
CSV	`pd.read_csv`	`readr::read_csv`
Excel	`pd.read_excel`	`readxl::read_excel`
Stata	`pd.read_stata`	`haven::read_dta`
SPSS	`pd.read_spss`	`haven::read_sav`
Parquet	`pd.read_parquet`	`arrow::read_parquet`

Two notes specific to this sector. Stata and SPSS files carry **value labels** — `1 = Yes`, `2`

`= No` — and `haven` preserves them while pandas mostly does not; if you are handed a `.dta` from a survey firm, read it in R even if you will analyse it in Python. And Excel exports frequently carry a merged title row above the header, which both readers will treat as the header unless you tell them to skip it.

3.8 What comes next

The register is now in memory with its types intact. The next lesson reshapes it — and reshapes the CommCare-style flattened export it could have arrived as — into one row per observation.

CHAPTER 4

Getting to a tidy table

Lesson 4 of 8 · 75 min

Reshape flattened repeat groups into one row per observation, join a member roster to its household, and prove the join did not silently change your caseload.

4.1 The rule

One row per observation, one column per variable, one table per kind of thing. The rule is old and it still does most of the work.

It is worth stating why, because "tidy" sounds like tidiness rather than engineering. In a tidy table, every operation you want — filter, group, join, plot — is the same operation regardless of which variable you are working on. In an untidy one, each variable needs bespoke code, and the bespoke code is where the errors live.

4.2 What untidy looks like in this sector

Three shapes account for nearly everything you will meet.

Flattened repeat groups. CommCare exports one row per form submission, so a household visit that screened three children becomes:

```
form_id,commune,child_1_id,child_1_muac,child_2_id,child_2_muac,child_3_id,child_3_muac
F0012,Gonaives,CH00854,133,CH00855,118,,
F0013,Verrettes,CH01439,143,,,,
```

One row is a form. You want one row per child. The column count also changes between exports, which means any code that names the columns explicitly breaks next month.

Values in column headers. A DHIS2 pivot puts periods across the top:

```
org_unit,Jan-2024,Feb-2024,Mar-2024
Gonaives,412,388,401
```

The month is data — it is a value of a variable called `period` — but it is living in a header.

Multiple things in one table. A survey export with household characteristics repeated on every member row. Nothing is wrong until someone computes a mean household size and gets a number weighted by household size.

4.3 Long and wide

Reshaping between the two is the single most useful mechanical skill in this course.

```
import pandas as pd

wide = pd.read_csv("data/raw/commcare-screening-export.csv")

long = wide.melt(
    id_vars=["form_id", "commune"],
    var_name="field",
```

```
    value_name="value",
)

# child_1_muac -> child 1, field muac
parts = long["field"].str.extract(r"^child_(?P<slot>\d+)_(?P<attribute>.+)$")
long = pd.concat([long, parts], axis=1).dropna(subset=["slot"])

children = (
    long.pivot(index=["form_id", "commune", "slot"], columns="attribute", values="value")
    .reset_index()
    .dropna(subset=["id"])
)

print(children.head())
```

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

wide <- read_csv("data/raw/commcare-screening-export.csv")

children <- wide |>
  pivot_longer(
    cols = starts_with("child_"),
    names_to = c("slot", "attribute"),
    names_pattern = "child_(\\d+)_(.+)",
    values_to = "value",
    values_transform = as.character
  ) |>
  pivot_wider(names_from = attribute, values_from = value) |>
  filter(!is.na(id))

children
```

Two details carry the weight:

- **Match the slot number with a pattern**, never by listing columns. The list changes between exports; the pattern does not.
- **Drop the empty slots.** A form that screened one child still produces columns for slots two and three, filled with nothing. Keeping them inflates your caseload with rows that are not children.

Check the arithmetic afterwards. If the export had 1,600 forms and the widest had three children, the melt produces 4,800 candidate rows, of which only the real ones survive the drop. That surviving count is your caseload, and it should match what the programme thinks it screened.

4.4 Joining a roster to its parent

KoboToolbox splits repeats into their own sheet, joined on `_parent_index`.

```
households = pd.read_excel("data/raw/kobo-export.xlsx", sheet_name="household")
members = pd.read_excel("data/raw/kobo-export.xlsx", sheet_name="member")

merged = members.merge(
    households[["_index", "commune", "interview_date"]],
    left_on="_parent_index",
```

```
    right_on="_index",
    how="left",
    validate="many_to_one",
    indicator=True,
)

print(merged["_merge"].value_counts())
```

```
library(readxl)

households <- read_excel("data/raw/kobo-export.xlsx", sheet = "household")
members     <- read_excel("data/raw/kobo-export.xlsx", sheet = "member")

merged <- members |>
  left_join(
    households |> select(`_index`, commune, interview_date),
    by = join_by(`_parent_index` == `_index`),
    relationship = "many-to-one",
    unmatched = "error"
  )

nrow(merged) == nrow(members)
```

`validate="many_to_one"` in pandas and `relationship = "many-to-one"` in dplyr are the important arguments, and almost nobody uses them. They raise if the relationship you asserted is not the one in the data — which is exactly the failure that otherwise multiplies your row count and is discovered three steps later as a caseload that is 14% too high.

4.5 Prove the join did what you said

Three checks, every time. They take a minute and they have caught more errors than any technique in this course.

```
before = len(members)
after = len(merged)

assert after == before, f"join changed the row count: {before} -> {after}"

unmatched = (merged["_merge"] == "left_only").sum()
assert unmatched == 0, f"{unmatched} members have no household"

assert merged["child_id"].is_unique, "duplicate identifiers after join"
```

```
stopifnot(nrow(merged) == nrow(members))
stopifnot(!any(is.na(merged$commune)))
stopifnot(!any(duplicated(merged$child_id)))
```

The row count check is the one to internalise. A left join can only leave the row count unchanged or increase it. If it increased, the right-hand table had duplicate keys, and every duplicated key has quietly multiplied its rows. In a caseload table, that is an overcount you will report to a donor.

> A join that changes your row count is not a data problem you fix downstream. It is a signal that you misunderstood one of the two tables, and the fix is upstream of the join.

4.6 The register is already tidy

The MUAC screening register this course uses is one row per child, which is why it can be worked directly:

```
muac.head()
```

```
head(muac)
```

That is deliberate. Reshaping is a real skill and you will need it, but it is not where the interesting decisions are. The interesting decisions start in the next lesson, when you look at what is actually in those 4,218 rows.

4.7 What comes next

The table is the right shape. The next lesson finds out what is wrong with it — systematically, and with the missingness broken down by site and week, because missingness that clusters is a completely different problem from missingness that does not.

CHAPTER 5

Finding what is wrong with the data

Lesson 5 of 8 · 90 min

Profile a fresh export systematically — missingness by site and week, implausible measurements, duplicates with no clean key, and codes that contradict each other.

5.1 Profile before you analyse

You have a tidy table with the right types. You do not yet know what is in it.

The instinct is to compute the indicator and look at the number. Resist it: the number will be plausible whatever is wrong, and once you have seen it you will find reasons to believe it. Profile first, decide second, compute third.

This lesson is the profiling. Five checks, in order of how much damage they do when missed.

5.2 Check 1: completeness, by group and not overall

A single completeness figure is nearly useless. What matters is whether the missingness clusters, because clustered missingness biases and scattered missingness mostly just costs precision.

```python
overall = muac.isna().mean().sort_values(ascending=False)
print(overall)

by_commune = (
    muac.assign(missing_age=muac["age_months"].isna())
    .groupby("commune")["missing_age"]
    .agg(["mean", "size"])
    .sort_values("mean", ascending=False)
)
print(by_commune)
```

```r
muac |>
  summarise(across(everything(), ~ mean(is.na(.x)))) |>
  tidyr::pivot_longer(everything(), names_to = "column", values_to = "missing") |>
  arrange(desc(missing))

muac |>
  group_by(commune) |>
  summarise(
    missing_age = mean(is.na(age_months)),
    n = n()
  ) |>
  arrange(desc(missing_age))
```

Run it on this register and the overall missingness of `age_months` is about 5%, which sounds tolerable. Break it down and Gros-Morne is far worse than the others. Break it down by week as well and it collapses onto a single campaign week, 10 to 14 June:

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

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")) |>
  group_by(week) |>
  summarise(missing_age = mean(is.na(age_months)), n = n()) |>
  arrange(desc(missing_age))
```

That is not a data quality nuisance. It is one team, one week, one tablet form that had the age field misconfigured — and it means that dropping incomplete rows removes Gros-Morne far more than any other commune. If you then rank communes by malnutrition rate, you have ranked them partly on which team's form was broken.

> Missingness that clusters in one site or one week is the difference between losing precision and losing the answer. Always break it down before you decide what to do with it.

5.3 Check 2: implausible values

Every sector has physical limits. Use them.

For MUAC in children 6 to 59 months, values below about 80 mm or above about 220 mm are not measurements — they are recording errors. The register carries seven records where the measurement was left in centimetres and never converted, which appear as values below 40.

```python
implausible = muac[(muac["muac_mm"] < 80) | (muac["muac_mm"] > 220)]
print(implausible[["child_id", "commune", "age_months", "muac_mm"]])

print(muac["muac_mm"].describe())
```

```r
muac |>
  filter(muac_mm < 80 | muac_mm > 220) |>
  select(child_id, commune, age_months, muac_mm)

summary(muac$muac_mm)
```

A value of 13.3 where you expected 133 is a unit error, and it is recoverable — you know what it was meant to be. A value of 450 is not recoverable and has to be treated as missing. Distinguishing the two is a judgement call, which is why it belongs in the cleaning log rather than in a one-line filter.

Check the age range too. A MUAC screening programme targets 6 to 59 months, so a row recording 72 months is either out of scope or a data entry error, and which one it is changes your denominator.

```python
print(muac["age_months"].describe())
print(muac[(muac["age_months"] < 6) | (muac["age_months"] > 59)].shape[0])
```

```r
summary(muac$age_months)
sum(muac$age_months < 6 | muac$age_months > 59, na.rm = TRUE)
```

5.4 Check 3: duplicates, including the ones without a clean key

Exact duplicates are easy:

```python
exact = muac[muac.duplicated(keep=False)]
print(f"{len(exact)} rows in exact-duplicate groups")

repeated_ids = muac[muac.duplicated(subset=["child_id"], keep=False)]
print(f"{len(repeated_ids)} rows sharing a child_id")
```

```r
sum(duplicated(muac))

muac |>
  group_by(child_id) |>
  filter(n() > 1) |>
  arrange(child_id)
```

This register has twelve exact duplicates, from forms submitted twice on a poor connection. Those are unambiguous — the same submission arrived twice, and one copy should go.

The harder case is a child re-registered under a new identifier, which shares no key at all. Six of those are in this file, findable only by matching on the attributes that should not coincide by chance:

```python
suspects = (
    muac.groupby(["commune", "screening_date", "age_months", "sex", "muac_mm"])
    .filter(lambda g: len(g) > 1)
    .sort_values(["commune", "screening_date", "muac_mm"])
)

print(suspects[["child_id", "commune", "screening_date", "age_months", "sex", "muac_mm"]])
```

```r
muac |>
  group_by(commune, screening_date, age_months, sex, muac_mm) |>
  filter(n() > 1) |>
  arrange(commune, screening_date, muac_mm) |>
  select(child_id, commune, screening_date, age_months, sex, muac_mm)
```

Be careful here: two different children of the same age and sex, screened in the same commune on the same day, with the same MUAC to the millimetre, is possible. In a large campaign it is even likely. This check produces *suspects*, not duplicates, and the resolution is a human looking at the register — not a `drop_duplicates()` call.

5.5 Check 4: internal contradictions

Columns that should agree, and do not. This register has two such faults.

Seventy-three records carry a referral outcome but no MUAC value — the decision column and the measurement column were filled by different people:

```
contradiction_a = muac[
    muac["outcome"].isin(["referred-tsfp", "referred-otp", "referred-sc"])
    & muac["muac_mm"].isna()
]
print(len(contradiction_a))
```

```
muac |>
  filter(outcome %in% c("referred-tsfp", "referred-otp", "referred-sc"),
         is.na(muac_mm)) |>
  nrow()
```

And nine records carry an outcome that contradicts their own measurement — a child measured at 140 mm marked as referred to outpatient therapeutic care, or a child at 110 mm marked no-action. This is what a mis-tapped tablet screen produces.

```
def expected_outcome(row):
    if pd.isna(row["muac_mm"]):
        return None
    if row["oedema"] is True or row["muac_mm"] < 115:
        return "referred-otp"
    if row["muac_mm"] < 125:
        return "referred-tsfp"
    return "no-action"

check = muac.assign(expected=muac.apply(expected_outcome, axis=1))
mismatched = check[
    check["expected"].notna()
    & (check["expected"] == "no-action")
    & (check["outcome"] != "no-action")
]
print(len(mismatched))
```

```
muac |>
  mutate(
    expected = case_when(
      is.na(muac_mm)             ~ NA_character_,
      oedema | muac_mm < 115     ~ "referred-otp",
      muac_mm < 125              ~ "referred-tsfp",
      TRUE                       ~ "no-action"
    )
  ) |>
  filter(!is.na(expected), expected == "no-action", outcome != "no-action") |>
  nrow()
```

Note that this check encodes a clinical rule. It is only as good as the rule, and referral to a stabilisation centre depends on complications the register does not record — so treat a mismatch as a flag for review, not as proof the outcome is wrong.

5.6 Check 5: inconsistent coding of the same thing

The `oedema` column is recorded as `true`/`false` for most of the file, but Ennery and Desdunes used `Y`/`N` in the first quarter, and some rows are blank.

```
raw = pd.read_csv(PATH, dtype={"oedema": "string"})
print(raw.groupby("commune")["oedema"].value_counts(dropna=False))
```

```
read_csv(PATH, col_types = cols(.default = col_character())) |>
  count(commune, oedema)
```

Always look at the raw string values of a categorical column before converting. Converting first and counting the missing afterwards tells you *how many* rows had an unexpected value; looking at the raw values tells you *what* they were, which is what you need to decide whether they are recoverable.

5.7 A profile function worth keeping

Wrap the whole thing so it runs on every new export without being retyped.

```
def profile(df, group=None):
    report = {
        "rows": len(df),
        "columns": df.shape[1],
        "duplicate_rows": int(df.duplicated().sum()),
        "missing": df.isna().mean().round(4).to_dict(),
    }
    if group:
        report["missing_by_group"] = (
            df.isna().groupby(df[group]).mean().round(4).to_dict("index")
        )
    return report

import json
print(json.dumps(profile(muac, group="commune"), indent=2, default=str))
```

```
profile <- function(df, group = NULL) {
  out <- list(
    rows = nrow(df),
    columns = ncol(df),
    duplicate_rows = sum(duplicated(df)),
    missing = sapply(df, function(x) round(mean(is.na(x)), 4))
  )
  if (!is.null(group)) {
    out$missing_by_group <- df |>
      group_by(.data[[group]]) |>
      summarise(across(everything(), ~ round(mean(is.na(.x)), 4)))
  }
  out
}

str(profile(muac, group = "commune"))
```

Run this on every export, save the output next to the data, and you have a record of what the file looked like on arrival. That record is what lets you answer "was this always like that?" six months later.

5.8 What comes next

You now know what is wrong. The next lesson is about deciding what to do about each fault — and, more importantly, about writing those decisions down in a form that survives the person who made them.

CHAPTER 6

Cleaning decisions and the cleaning log

Lesson 6 of 8 · 75 min

Decide what to do about each defect, quantify what the decision costs the final number, and record it so an auditor — or you in six months — can follow the reasoning.

6.1 Cleaning is a set of decisions, not a script

The profiling in the last lesson produced a list of faults. None of them has an objectively correct treatment. Each has a defensible treatment and a rationale, and the rationale is the part that matters — a reviewer will not object to you dropping seven rows, they will object to not being able to find out why.

So: for each fault, three things. What you did, why, and how many rows it touched.

6.2 The four options

Every defect gets one of these.

Option	When it applies	What it costs
Correct	You know what the value was meant to be	Nothing, if you are right
Set to missing	The value is wrong and unrecoverable	Precision, and possibly bias
Drop the row	The row is not a real observation	Caseload, if you are wrong
Keep and flag	The value may be right, and you cannot tell	Nothing, but the reader must handle it

"Keep and flag" is under-used. It is the honest option when a value is suspicious but not impossible, and it pushes the judgement to the person reading the analysis instead of hiding it inside it.

6.3 Working through this register

Unit errors — seven records in centimetres. Recoverable. A MUAC of 13.3 was measured as 133 mm; the decimal point is the whole story. Correct them, and record that you did.

```python
import numpy as np

unit_error = muac["muac_mm"].notna() & (muac["muac_mm"] < 40)
n_unit_error = int(unit_error.sum())

muac.loc[unit_error, "muac_mm"] = muac.loc[unit_error, "muac_mm"] * 10
```

```r
unit_error <- !is.na(muac$muac_mm) & muac$muac_mm < 40
n_unit_error <- sum(unit_error)

muac <- muac |>
  mutate(muac_mm = if_else(!is.na(muac_mm) & muac_mm < 40, muac_mm * 10L, muac_mm))
```

Multiplying by ten is safe here only because the plausible ranges do not overlap: nothing genuinely measured in millimetres is below 40, and nothing genuinely measured in centimetres is above 22. Check that the two ranges are disjoint before applying a rule like this. If they overlap, you cannot tell the cases apart and the values are not recoverable.

Implausible values that are not unit errors. Not recoverable. Set to missing, do not drop the row — the child was screened, and the rest of the record is still evidence.

```python
implausible = muac["muac_mm"].notna() & (
    (muac["muac_mm"] < 80) | (muac["muac_mm"] > 220)
)
n_implausible = int(implausible.sum())

muac.loc[implausible, "muac_mm"] = np.nan
```

```r
implausible <- !is.na(muac$muac_mm) & (muac$muac_mm < 80 | muac$muac_mm > 220)
n_implausible <- sum(implausible)

muac <- muac |>
  mutate(muac_mm = if_else(implausible, NA_integer_, muac_mm))
```

Exact duplicates — twelve rows. Drop. A form submitted twice on a poor connection is one screening, not two.

```python
before = len(muac)
muac = muac.drop_duplicates()
n_exact_dupes = before - len(muac)
```

```r
before <- nrow(muac)
muac <- distinct(muac)
n_exact_dupes <- before - nrow(muac)
```

Re-registrations under a new identifier — six suspected. Keep and flag. You cannot distinguish these from genuine coincidences without going back to the register, so mark them and say so in the report.

```python
key = ["commune", "screening_date", "age_months", "sex", "muac_mm"]
muac["suspected_duplicate"] = muac.duplicated(subset=key, keep=False) & muac[
    "muac_mm"
].notna()

n_suspected = int(muac["suspected_duplicate"].sum())
```

```r
muac <- muac |>
  group_by(commune, screening_date, age_months, sex, muac_mm) |>
  mutate(suspected_duplicate = n() > 1 & !is.na(muac_mm)) |>
  ungroup()

n_suspected <- sum(muac$suspected_duplicate)
```

Inconsistent oedema coding. Recoverable. `Y` and `N` mean what they look like they mean; blanks do not, and become missing.

```python
oedema_map = {
    "true": True, "TRUE": True, "Y": True, "y": True, "yes": True,
```

```
    "false": False, "FALSE": False, "N": False, "n": False, "no": False,
}
muac["oedema"] = muac["oedema"].astype("string").str.strip().map(oedema_map)
n_oedema_missing = int(muac["oedema"].isna().sum())
```

```
muac <- muac |>
  mutate(
    oedema = case_when(
      tolower(trimws(oedema)) %in% c("true", "y", "yes") ~ TRUE,
      tolower(trimws(oedema)) %in% c("false", "n", "no") ~ FALSE,
      TRUE ~ NA
    )
  )

n_oedema_missing <- sum(is.na(muac$oedema))
```

Missing age clustered in Gros-Morne. This is the one that needs a decision rather than a rule, and it is the subject of the next section.

6.4 Quantify what the decision costs

The reason to take missing age seriously is that the obvious treatment — drop rows with missing age — changes the answer. Measure it rather than assuming.

```
gam_threshold = 125

complete = muac.dropna(subset=["age_months", "muac_mm"])
all_measured = muac.dropna(subset=["muac_mm"])

def gam_rate(df):
    return (df["muac_mm"] < gam_threshold).mean()

comparison = pd.DataFrame(
    {
        "dropping_missing_age": complete.groupby("commune").apply(gam_rate),
        "keeping_missing_age": all_measured.groupby("commune").apply(gam_rate),
    }
)
comparison["difference"] = (
    comparison["dropping_missing_age"] - comparison["keeping_missing_age"]
)
print(comparison.sort_values("difference", ascending=False).round(4))
```

```
gam_rate <- function(df) mean(df$muac_mm < 125, na.rm = TRUE)

complete <- muac |> filter(!is.na(age_months), !is.na(muac_mm))
all_measured <- muac |> filter(!is.na(muac_mm))

comparison <- full_join(
  complete |> group_by(commune) |> summarise(dropping = gam_rate(pick(everything()))),
  all_measured |> group_by(commune) |> summarise(keeping = gam_rate(pick(everything()))),
  by = "commune"
) |>
  mutate(difference = dropping - keeping) |>
  arrange(desc(abs(difference)))

comparison
```

Gros-Morne moves and the other communes barely do. That is the bias, made visible. Since the indicator here does not require age — MUAC thresholds for 6 to 59 months are a single band, unlike weight-for-height z-scores — the right decision is to keep the rows, use them for the MUAC indicator, and exclude them only from any analysis that genuinely needs age.

That is a better decision than dropping, and it is only available because you looked.

> The general principle: drop rows for a specific analysis, never from the dataset. A row missing age is still evidence about MUAC. Filtering at the point of use keeps each analysis on the largest sample that supports it.

6.5 The cleaning log

Everything above becomes rows in a table that ships with the analysis.

```
cleaning_log = pd.DataFrame(
    [
        {
            "rule": "MUAC unit error corrected (cm to mm)",
            "column": "muac_mm",
            "action": "corrected",
            "rows": n_unit_error,
            "rationale": "Values below 40 are centimetres left unconverted. "
            "Plausible mm and cm ranges are disjoint, so the correction is unambiguous.",
        },
        {
            "rule": "Implausible MUAC set to missing",
            "column": "muac_mm",
            "action": "set-missing",
            "rows": n_implausible,
            "rationale": "Outside 80-220 mm is not a measurement for 6-59 months. "
            "Row retained; the screening happened.",
        },
        {
            "rule": "Exact duplicate submissions removed",
            "column": "all",
            "action": "dropped",
            "rows": n_exact_dupes,
            "rationale": "Identical rows are one form submitted twice on a poor connection.
    ",
        },
        {
            "rule": "Suspected re-registration flagged",
            "column": "suspected_duplicate",
            "action": "flagged",
            "rows": n_suspected,
            "rationale": "Same commune, date, age, sex and MUAC under different ids. "
            "Cannot be distinguished from coincidence without the paper register.",
        },
        {
            "rule": "Oedema coding harmonised",
            "column": "oedema",
            "action": "corrected",
            "rows": n_oedema_missing,
            "rationale": "Ennery and Desdunes used Y/N in Q1. Blanks remain missing.",
        },
        {
```

```
            "rule": "Missing age retained",
            "column": "age_months",
            "action": "kept",
            "rows": int(muac["age_months"].isna().sum()),
            "rationale": "60% missing in the Gros-Morne week of 10-14 June. Dropping "
            "shifts the commune ranking; MUAC thresholds do not require age.",
        },
    ]
)

cleaning_log.to_csv("output/tables/cleaning-log.csv", index=False)
print(cleaning_log[["rule", "action", "rows"]])
```

```
cleaning_log <- tibble::tribble(
  ~rule,                                  ~column,               ~action,       ~rows,
  "MUAC unit error corrected (cm to mm)",  "muac_mm",             "corrected",   n_unit_
    error,
  "Implausible MUAC set to missing",       "muac_mm",             "set-missing", n_
    implausible,
  "Exact duplicate submissions removed",   "all",                 "dropped",     n_exact_
    dupes,
  "Suspected re-registration flagged",     "suspected_duplicate", "flagged",     n_
    suspected,
  "Oedema coding harmonised",              "oedema",              "corrected",   n_oedema
    _missing,
  "Missing age retained",                  "age_months",          "kept",        sum(is.
    na(muac$age_months))
)

readr::write_csv(cleaning_log, "output/tables/cleaning-log.csv")
cleaning_log
```

Four properties make this log useful rather than ceremonial:

- **The row counts come from the code**, not from memory. If the rule changes, the count changes with it.
- **It ships with the analysis** — as an annex to the report, not as a file on someone's laptop.
- **The rationale is a sentence, not a category.** "Data quality" is not a rationale.
- **It is written as the cleaning happens**, not reconstructed afterwards. Reconstructed logs are wrong in exactly the places that matter.

6.6 The one thing never to do

Do not edit `data/raw/`. Not one cell, not to fix an obvious typo, not "just this once" the night before a deadline.

Every correction above is code. That means it is visible, reversible, applied identically to next quarter's export, and re-runnable by someone who doubts you. A cell edited by hand in a spreadsheet is none of those things, and it is invisible six months later when the number is queried.

6.7 What comes next

The data is clean and the decisions are recorded. The next lesson turns it into an indicator — which means confronting the fact that a rate is a numerator over a denominator, and that almost nobody agrees on the denominator.

CHAPTER 7

From tidy data to an indicator table

Lesson 7 of 8 · 90 min

Turn a clean dataset into the disaggregated indicator table a logframe asks for, with numerator, denominator and confidence interval stated explicitly.

7.1 An indicator is a numerator over a denominator, disaggregated

That sentence is the whole lesson. Most reporting disputes are really disagreements about the denominator, and most of the rest are disagreements about which rows belong in the numerator.

So before any code: write the definition down.

7.2 The indicator reference sheet

For every indicator you report, fill this in. It takes five minutes and it ends the argument before it starts.

Field	For our example
Indicator	Global acute malnutrition (GAM) prevalence by MUAC
Numerator	Children 6-59 months screened with MUAC below 125 mm, or with bilateral pitting oedema
Denominator	Children 6-59 months with a valid MUAC measurement or a recorded oedema assessment
Disaggregation	Commune, sex, age band (6-23, 24-59 months)
Frequency	Quarterly, and annually for the donor report
Source	Community mass screening register, CommCare
Decision it informs	Which communes receive an additional CMAM site next quarter
Reference value	GAM at or above 15% is the emergency threshold

Note what the denominator is *not*. It is not all children in the commune — that would be coverage-adjusted prevalence and requires a population figure this register does not have. It is not all rows in the file — that would include rows with no measurement at all. Being explicit about this is what stops the number being quietly redefined between one report and the next.

> If you cannot write the denominator in one sentence, you do not yet have an indicator. You have a column you are about to average.

7.3 Build the numerator as a column

Do not filter. Create a flag, keep every row, and let the aggregation do the work — that way the denominator is visible in the same table as the numerator.

```
import numpy as np

SAM_MM = 115
```

```
GAM_MM = 125

muac["has_assessment"] = muac["muac_mm"].notna() | muac["oedema"].notna()

muac["sam"] = np.where(
    ~muac["has_assessment"],
    np.nan,
    ((muac["muac_mm"] < SAM_MM) | (muac["oedema"] == True)).astype(float),
)

muac["gam"] = np.where(
    ~muac["has_assessment"],
    np.nan,
    ((muac["muac_mm"] < GAM_MM) | (muac["oedema"] == True)).astype(float),
)
```

```
SAM_MM <- 115
GAM_MM <- 125

muac <- muac |>
  mutate(
    has_assessment = !is.na(muac_mm) | !is.na(oedema),
    sam = if_else(has_assessment, (muac_mm < SAM_MM) | oedema %in% TRUE, NA),
    gam = if_else(has_assessment, (muac_mm < GAM_MM) | oedema %in% TRUE, NA)
  )
```

Two details that are easy to get wrong and expensive to miss:

- **Oedema is SAM regardless of measurement.** A child at 130 mm with bilateral pitting oedema is severely acutely malnourished. Filtering on `muac_mm` alone understates the caseload, and it understates it precisely among the most severe cases.
- **`oedema == True` rather than a truthiness test.** Missing oedema is not false. In R, `oedema %in% TRUE` treats `NA` as not-oedema for the flag while keeping it distinguishable elsewhere; in Python the explicit comparison does the same.

7.4 The indicator table

```
def indicator_table(df, by):
    grouped = df.groupby(by, dropna=False)
    table = grouped.agg(
        screened=("child_id", "size"),
        denominator=("has_assessment", "sum"),
        sam_cases=("sam", "sum"),
        gam_cases=("gam", "sum"),
    )
    table["gam_rate"] = table["gam_cases"] / table["denominator"]
    table["sam_rate"] = table["sam_cases"] / table["denominator"]
    return table.reset_index()

by_commune = indicator_table(muac, "commune").sort_values(
    "gam_rate", ascending=False
)
print(by_commune.round(4))
```

```
indicator_table <- function(df, by) {
```

```
  df |>
    group_by(across(all_of(by))) |>
    summarise(
      screened    = n(),
      denominator = sum(has_assessment),
      sam_cases   = sum(sam, na.rm = TRUE),
      gam_cases   = sum(gam, na.rm = TRUE),
      .groups = "drop"
    ) |>
    mutate(
      gam_rate = gam_cases / denominator,
      sam_rate = sam_cases / denominator
    )
}

by_commune <- indicator_table(muac, "commune") |> arrange(desc(gam_rate))
by_commune
```

The table carries `screened` and `denominator` as separate columns on purpose. They differ by the rows with no assessment at all, and a reader who can see both can tell how much of the register the rate is based on. A table that reports only the rate hides that entirely.

Run it and the overall GAM rate lands near 8.6% with SAM near 2.2%, ranging from roughly 5% to 15% across communes. One commune crosses the 15% emergency threshold; the best does not come close. Those figures are stated in the dataset's quality notes precisely so you can check your working against them — if your GAM comes out at 30%, you have a bug, not a famine.

7.5 Disaggregation

The logframe asked for commune, sex and age band. Age band needs constructing, and the boundaries are a choice you should state.

```
muac["age_band"] = pd.cut(
    muac["age_months"],
    bins=[6, 24, 60],
    labels=["6-23 months", "24-59 months"],
    right=False,
)

by_sex = indicator_table(muac, ["commune", "sex"])
by_age = indicator_table(muac, "age_band")

print(by_age.round(4))
```

```
muac <- muac |>
  mutate(
    age_band = cut(
      age_months,
      breaks = c(6, 24, 60),
      labels = c("6-23 months", "24-59 months"),
      right = FALSE
    )
  )

by_sex <- indicator_table(muac, c("commune", "sex"))
by_age <- indicator_table(muac, "age_band")
```

```
by_age
```

`right = FALSE` makes the bands left-closed: 6 to 23 completed months, then 24 to 59. Getting this backwards puts 24-month-olds in the younger band, which shifts every rate slightly and is invisible in the output. State the convention in the reference sheet.

Remember the missing-age problem from lesson 5. The age disaggregation necessarily excludes rows with no age, and Gros-Morne is over-represented among those. Report the age table with that caveat attached, or report it excluding Gros-Morne and say so.

7.6 Say how uncertain you are

A rate from 340 children is not the same claim as a rate from 12. Attach an interval.

```
from scipy.stats import beta

def wilson_interval(successes, n, confidence=0.95):
    if n == 0:
        return (np.nan, np.nan)
    lower = beta.ppf((1 - confidence) / 2, successes, n - successes + 1) if successes > 0
    else 0.0
    upper = beta.ppf(1 - (1 - confidence) / 2, successes + 1, n - successes) if successes
    < n else 1.0
    return (lower, upper)

bounds = by_commune.apply(
    lambda r: wilson_interval(r["gam_cases"], r["denominator"]), axis=1
)
by_commune["gam_low"] = [b[0] for b in bounds]
by_commune["gam_high"] = [b[1] for b in bounds]

print(by_commune[["commune", "denominator", "gam_rate", "gam_low", "gam_high"]].round(4))
```

```
ci <- function(cases, n) {
  if (n == 0) return(c(NA_real_, NA_real_))
  test <- binom.test(cases, n)
  test$conf.int
}

by_commune <- by_commune |>
  rowwise() |>
  mutate(
    gam_low  = ci(gam_cases, denominator)[1],
    gam_high = ci(gam_cases, denominator)[2]
  ) |>
  ungroup()

by_commune |> select(commune, denominator, gam_rate, gam_low, gam_high)
```

Now look at the ranking again. Several communes have overlapping intervals, which means the ordering between them is not supported by the data. If the decision this table informs is "which commune gets the additional CMAM site", that matters enormously — and a table without intervals would have let you rank them with false confidence.

This is also a census of those screened rather than a probability sample, so the interval describes sampling variation only. It says nothing about whether the children who came to be screened resemble the children who did not, which is usually the larger source of error and belongs in the limitations section.

7.7 Format it for the report

```python
report = by_commune.assign(
    gam=lambda d: (d["gam_rate"] * 100).round(1).astype(str) + "%",
    ci=lambda d: "("
    + (d["gam_low"] * 100).round(1).astype(str)
    + " – "
    + (d["gam_high"] * 100).round(1).astype(str)
    + ")",
)[["commune", "screened", "denominator", "gam_cases", "gam", "ci"]]

report.columns = [
    "Commune", "Screened", "Assessed", "GAM cases", "GAM rate", "95% CI",
]
report.to_csv("output/tables/gam-by-commune.csv", index=False)
print(report.to_string(index=False))
```

```r
report <- by_commune |>
  transmute(
    Commune     = commune,
    Screened    = screened,
    Assessed    = denominator,
    `GAM cases` = gam_cases,
    `GAM rate`  = sprintf("%.1f%%", gam_rate * 100),
    `95% CI`    = sprintf("(%.1f – %.1f)", gam_low * 100, gam_high * 100)
  )

readr::write_csv(report, "output/tables/gam-by-commune.csv")
report
```

Every column in that table is defensible: you can say where each number came from, what it counted, what it did not, and how uncertain it is.

7.8 What comes next

The last lesson makes this run again — on next quarter's export, on someone else's laptop, without editing anything.

CHAPTER 8

Making it run again next quarter

Lesson 8 of 8 · 60 min

Turn a working analysis into one that reruns on a new export without editing, fails loudly when the input changes, and can be handed to someone who has none of your context.

8.1 The test

Next quarter's export arrives. How many things do you have to edit before the report regenerates?

If the answer is more than one — the input path — the analysis is not reproducible, it is a record of one afternoon. This lesson gets the answer down to one.

8.2 Split the analysis into stages that hand off files

Three scripts, each with one job, each writing a file the next one reads.

```
scripts/
  01-read.py        raw export      -> data/interim/muac-typed.parquet
  02-clean.py       typed           -> data/processed/muac-clean.parquet
                                       output/tables/cleaning-log.csv
  03-indicators.py  clean           -> output/tables/gam-by-commune.csv
```

The value is not tidiness. It is that when the indicator table looks wrong you can rerun stage three in two seconds instead of rerunning a fifteen-minute notebook, and that a colleague debugging stage two does not have to understand stage three.

Use Parquet rather than CSV between stages. It preserves types — which means the work you did in lesson 3 to get `child_id` read as text is not undone the moment you write it out and read it back.

```python
muac.to_parquet("data/interim/muac-typed.parquet", index=False)
```

```r
arrow::write_parquet(muac, "data/interim/muac-typed.parquet")
```

8.3 Put the settings in one place

Every threshold, path and cut-off goes at the top of the file or in a config, never buried in the middle of an expression.

```python
# config.py
from pathlib import Path

RAW = Path("data/raw/muac-screening-artibonite-2024.v1.csv")
INTERIM = Path("data/interim/muac-typed.parquet")
PROCESSED = Path("data/processed/muac-clean.parquet")
TABLES = Path("output/tables")
```

```
SAM_MM = 115
GAM_MM = 125
PLAUSIBLE_MUAC = (80, 220)
AGE_RANGE_MONTHS = (6, 59)
MISSING_CODE = "-99"
```

```
# config.R
RAW       <- "data/raw/muac-screening-artibonite-2024.v1.csv"
INTERIM   <- "data/interim/muac-typed.parquet"
PROCESSED <- "data/processed/muac-clean.parquet"
TABLES    <- "output/tables"

SAM_MM           <- 115
GAM_MM           <- 125
PLAUSIBLE_MUAC   <- c(80, 220)
AGE_RANGE_MONTHS <- c(6, 59)
MISSING_CODE     <- "-99"
```

A reviewer who wants to know what thresholds you used reads one file. A successor adapting this to a different country changes one file. And when the WHO revises a cut-off, you change one number rather than searching for `125` across four scripts and missing the one written as `12.5 * 10`.

8.4 Make the pipeline fail loudly

An analysis that silently produces a wrong number is worse than one that crashes. Assert what must be true, at every stage boundary.

```
def check_input(df):
    expected = {
        "child_id", "commune", "screening_date", "age_months",
        "sex", "muac_mm", "oedema", "outcome",
    }
    missing = expected - set(df.columns)
    if missing:
        raise ValueError(f"export is missing columns: {sorted(missing)}")

    if df["child_id"].isna().any():
        raise ValueError("rows without an identifier")

    measured = df["muac_mm"].dropna()
    if len(measured) == 0:
        raise ValueError("no MUAC measurements at all -- wrong file?")

    share_missing = df["muac_mm"].isna().mean()
    if share_missing > 0.20:
        raise ValueError(
            f"{share_missing:.1%} of MUAC missing -- above the 20% tolerance. "
            "Investigate before reporting."
        )
```

```
check_input <- function(df) {
  expected <- c("child_id", "commune", "screening_date", "age_months",
                "sex", "muac_mm", "oedema", "outcome")
  missing <- setdiff(expected, names(df))
  if (length(missing) > 0) {
    stop("export is missing columns: ", paste(missing, collapse = ", "))
```

```
  }

  if (any(is.na(df$child_id))) stop("rows without an identifier")

  share_missing <- mean(is.na(df$muac_mm))
  if (share_missing > 0.20) {
    stop(sprintf(
      "%.1f%% of MUAC missing - above the 20%% tolerance. Investigate before reporting.",
      share_missing * 100
    ))
  }
}
```

The threshold check is the interesting one. It encodes an editorial judgement — that above 20% missing, the rate is not worth reporting — and it enforces it automatically, on every future export, including the ones processed by someone who never read this lesson.

> Write the assertion for the failure you would be embarrassed to miss, not for the failure you think is likely. The likely ones you will notice.

8.5 Parameterise instead of copying

Twelve communes, twelve commune reports. Do not copy the script twelve times.

```
import sys
from pathlib import Path

def build_report(commune: str | None = None):
    df = pd.read_parquet(PROCESSED)
    if commune:
        df = df[df["commune"] == commune]
        if df.empty:
            raise ValueError(f"no rows for commune {commune!r}")

    table = indicator_table(df, "age_band")
    name = commune.lower().replace(" ", "-") if commune else "all"
    table.to_csv(TABLES / f"gam-{name}.csv", index=False)
    return table

if __name__ == "__main__":
    target = sys.argv[1] if len(sys.argv) > 1 else None
    build_report(target)
```

```
build_report <- function(commune = NULL) {
  df <- arrow::read_parquet(PROCESSED)
  if (!is.null(commune)) {
    df <- filter(df, commune == !!commune)
    if (nrow(df) == 0) stop("no rows for commune ", commune)
  }

  table <- indicator_table(df, "age_band")
  name <- if (is.null(commune)) "all" else tolower(gsub(" ", "-", commune))
  readr::write_csv(table, file.path(TABLES, paste0("gam-", name, ".csv")))
  table
}

args <- commandArgs(trailingOnly = TRUE)
```

```
build_report(if (length(args) > 0) args[1] else NULL)
```

The same pattern extends to the full narrative report. Quarto renders one template with a parameter, twelve times, producing twelve documents that cannot drift from one another because there is only one document.

8.6 What never goes in the repository

This matters more in this sector than in most. A repository holding programme data can hold beneficiary names, GPS points that identify a household, or a protection case list.

- **Never commit raw beneficiary data.** Not identified, not pseudonymised. Git keeps every version forever, and deleting a file does not remove it from the history.
- **Never commit credentials.** DHIS2 tokens, KoboToolbox API keys, database passwords. Read them from the environment.
- **Add `data/raw/` to `.gitignore`** and document where the real file lives — a shared drive, an encrypted volume — in the README.

```
data/raw/
data/interim/
data/processed/
.env
*.Rhistory
__pycache__/
.Rproj.user/
```

The analysis code is what belongs in version control. The data belongs wherever your organisation's data protection policy says it belongs, and the README should say which.

8.7 The handover README

Written for someone with your job title and none of your context. Five headings.

```
# MUAC screening analysis, Artibonite

## What this produces
Quarterly GAM and SAM rates by commune, sex and age band, with 95% intervals.
Feeds the CMAM site allocation decision and the quarterly donor report.

## Where the data comes from
CommCare export, project `artibonite-nutrition`, form `Community screening`.
Exported monthly by the M&E assistant to the shared drive at <path>.
Not in this repository.

## How to run it
    uv sync
    uv run python scripts/01-read.py
    uv run python scripts/02-clean.py
    uv run python scripts/03-indicators.py

## Decisions you should know about
See output/tables/cleaning-log.csv. The one that matters: rows with missing
age are kept, because 60% of the Gros-Morne week of 10-14 June is missing age
and dropping them shifts the commune ranking.
```

```
## Who to ask
Indicator definitions: <name>, M&E Manager.
Data collection issues: <name>, Field Coordinator.
```

That README is the difference between a successor continuing your work and a successor rebuilding it in Excel because they could not tell what your scripts assumed.

8.8 Where to go from here

You can now take a routine programme export from arrival to a defended indicator table, in either language, with the reasoning recorded. That is the foundation the rest of the programme builds on:

- **Data Cleaning and Validation** goes deeper into the profiling and correction work of lessons 5 and 6, including fuzzy matching and plausibility rule sets.
- **Indicator Design and the LogFrame** starts where lesson 7's reference sheet ends, and works back to the theory of change the indicator is supposed to evidence.
- **Survey Analysis, Sampling and Weighting** covers what this course did not: what changes when the data is a probability sample rather than a census of those screened.

Practise on a different dataset before moving on. The WASH household survey and the routine vaccination coverage register on this platform both carry a different mix of defects, and working one of them end to end without the lesson in front of you is the real test of whether this stuck.

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-analysis-foundations

Generated July 26, 2026.