CASSION · COURSE

Joining and Reshaping Programme Data

Household and member rosters, repeat groups, admissions and discharges, and monthly aggregates that have to line up with a population denominator.

Instructor	Pierre Robentz Cassion
Level	Intermediate
Learner hours	14 h
Taught in	Python · R
Updated	July 27, 2026
Sectors	Public Health · Emergency Response
Standards and methodologies	UNICEF indicator definitions · Results-Based Management (RBM)

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

What you will be able to do

- Choose a join type from what you want to be true of the result, and prove afterwards that it was
- State the grain of every table you touch, and aggregate to the grain before joining rather than after
- Move a table between long and wide deliberately, and undo the flattened repeat group a form platform exports
- Attach a population denominator from an administrative frame, and defend the denominator you chose
- Build a complete period grid so a month that was never reported is visible rather than absent
- Reconcile a line-level register against the aggregate that was reported upward, and publish the gap

Prerequisites

None. This course assumes no prior programming experience.

Contents

About this course

Almost no programme question is answerable from one file. The children are in the screening register and the population is in the census projection. The households are in one export and their members in another. The daily register is in the tablet and the monthly figure is in DHIS2, and the two disagree.

This course is about putting those together. It is shorter than the cleaning course and harder, because the failure mode is different: a cleaning defect makes a value wrong, and you can usually see it. **A join defect makes a row count wrong, and there is nothing to see** — no error, no warning, no implausible value. A left join that adds 120 rows and a many-to-many that turns 70,245 rows into 3.5 million both produce a table that looks exactly like a table.

So the discipline here is stated up front and repeated in every lesson: **say what the join should do before you run it, and prove afterwards that it did.** Both languages will check the relationship for you if you ask, and asking costs one argument.

Every technique is shown in Python and in R, against platform datasets where the problem is genuine — a student roster that is not unique on its own identifier, a DHIS2-shaped extract whose complete twelve-month grid hides which facilities were silent, and a WASH survey where the analysis is per person and the data is per household.

You should have done *Data Cleaning and Validation*, or at least be in the habit of asserting a key before you use it. That course ends where this one starts.

Part I

Joins you can prove

CHAPTER 1

Four joins, and the question each one answers

Lesson 1 of 8 · 90 min

Inner, left, full and anti — chosen from what you need to be true of the result rather than from habit. Plus the row-count arithmetic that explains every fan-out you will ever see.

1.1 Pick the join from the sentence you are going to write

Most people pick a join by habit — left join, because it is the one that usually works. That is backwards. The join is a claim about which rows belong in the answer, and the claim comes from the sentence you intend to publish.

The sentence you will write	The join
"Attendance for every enrolled student"	left, register on the left
"Attendance for students we have both a record and a roster row for"	inner
"Everything from both sides, so nothing is lost"	full
"Students marked present who are not on the roster"	anti

Read those the other way and each join has a failure it invites. An inner join **silently discards** the rows that did not match, and those rows are frequently the finding. A left join **silently multiplies** when the right side is not unique. A full join produces a table where a missing value can mean two different things. And an anti-join produces nothing when everything matched, which people read as "the check passed" without checking that it ran.

1.2 The arithmetic, once, by hand

Every surprise in this course comes out of one rule. A join emits **one row for every pair of rows that share the key.**

If a key value appears `m` times on the left and `n` times on the right, it contributes `m × n` rows.

Left occurrences	Right occurrences	Rows emitted
1	1	1
60	1	60
60	2	120
50	3000	150,000

Three consequences fall straight out of that table:

- **Row count preserved** only when the right side is unique on the key. This is the sentence the whole lesson rests on.
- **Row count multiplied** when it is not — quietly, with no error, by a factor nobody chose.
- **Row count reduced** only by an inner join, or by a key that fails to match. A left join never removes a row, so a shrinking left join means the key changed underneath you.

The third line of the table is a real defect from the previous course: two students on the roster twice, sixty school days each, and a left join that adds 120 rows to a 70,245-row table. The fourth is what happens when you join on the wrong column, and we will do it deliberately in a moment.

1.3 The setup

Two files. A roster of 1,202 rows covering 1,200 students across 24 schools, and 70,245 daily attendance marks.

```python
import pandas as pd

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

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

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

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

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

1,202 rows and 1,200 students. Deal with that before joining — the previous course showed why, and the rest of this lesson assumes you have.

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

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

Keeping the first row is a decision, not a default. For these two students it is the wrong one — one row records the school they left and the other the school they joined, and "first" picks whichever the export happened to sort first. In real work this goes to whoever holds the register. Here it is a placeholder so the joins below have something clean to work on, and it belongs in the cleaning log.

1.4 Left join: keep the left table, attach columns

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

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

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

70,245 to 70,245. That is what a left join is supposed to do, and the `validate` /

`relationship` argument is what makes it a guarantee rather than a hope. Without it, the same call on the undeduplicated roster returns 70,365 and tells you nothing.

1.5 Inner join: and the rows it takes with it

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

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

Here nothing is dropped, because every attendance row has a roster row. That is unusual and worth saying out loud when it happens.

When it is not zero, the number matters more than the join does. An inner join that drops 4% of a distribution list is dropping four percent of somebody's beneficiaries, and the report will say "12,000 people reached" with no footnote, because the rows that would have raised the question are the ones that left.

> Use an inner join when you have already looked at what it removes. Using it *to* remove them is how the awkward rows get disposed of without a decision.

1.6 Full join: nothing is lost, and now you have two kinds of blank

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

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

pandas' `indicator=True` adds a `_merge` column reading `both`, `left_only` or `right_only`, and it is the single most useful argument in this lesson. dplyr has no equivalent, so add one:

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

The reason to bother: after a full join, a blank `grade` means either "this student has no roster row" or "this student has a roster row with no grade recorded". Those are completely different findings and the join has made them look identical. The side column keeps them apart.

1.7 Anti-join: the check, not the join

An anti-join returns the rows on one side with no partner on the other. It is rarely the analysis and almost always the check.

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

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

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

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

Both directions, every time, and they mean opposite things:

- **Attendance with no roster row** — someone is being recorded who is not enrolled. A data problem, or an enrolment list that is out of date.
- **Roster rows with no attendance** — enrolled children never marked either way. In an education programme that is not a defect, it is the finding: those are the children who never came.

Both are zero here. Write the check anyway, because next term it will not be.

1.8 Joining on the wrong column

Do this once, deliberately, so you recognise the shape of it when it happens by accident. Both files carry `school_id` after the first join, and joining on it instead of `student_id` is a single-character mistake:

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

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

70,245 becomes **3,567,105**. Each attendance row matched every student in the same school — about fifty of them — and the attendance rate computed from that table is still around 88%, because multiplying every row by fifty leaves the proportion alone.

That is the point. **The rate survives; the counts do not.** Anything reported as a number of children rises fiftyfold, and a rate that looks right is exactly the reason nobody checks the count.

1.9 What comes next

You have four joins and the arithmetic that governs them. The next lesson turns that into a habit that runs without you thinking about it — a reconciliation table produced by every join, with matched rows, both anti-join counts and an assertion that fails when the row count moves.

CHAPTER 2

Proving the join did what you said

Lesson 2 of 8 · 90 min

A reconciliation table from every join — matched rows, both anti-join counts, the row count before and after — plus the suffix collision that overwrites a column without telling you.

2.1 The check has to be cheaper than the bug

Nobody skips join checks because they think joins are safe. They skip them because checking takes four commands, the output has to be read, and the join "obviously" worked.

So the check has to become one call that produces one small table. Write it once, put it in the file every script imports, and the cost of checking drops below the cost of wondering.

2.2 Declare the relationship, always

The first line of defence is an argument you were already able to pass.

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

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

Four values, and choosing between them forces you to say what you believe:

You expect	pandas `validate=`	dplyr `relationship=`
One row each side	`one_to_one`	`one-to-one`
Many left, one right	`many_to_one`	`many-to-one`
One left, many right	`one_to_many`	`one-to-many`
Genuinely many-to-many	omit	`many-to-many`

dplyr warns on an unexpected many-to-many even without the argument; pandas does not. That difference has decided a number of quietly wrong figures, and it is the reason the Python examples in this course always pass `validate=`.

Note the last row. Passing `many-to-many` explicitly is not a way of silencing the check — it is a claim that the fan-out is intended, and it belongs beside a comment saying what the resulting grain is.

2.3 The reconciliation table

The relationship argument tells you the shape was right. It does not tell you what failed to match, and that is usually the more interesting question.

```
def reconcile(left, right, on, left_name="left", right_name="right"):
    left_keys = left[on].drop_duplicates()
    right_keys = right[on].drop_duplicates()
```

```
    merged = left_keys.merge(right_keys, on=on, how="outer", indicator=True)
    counts = merged["_merge"].value_counts()

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

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

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

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

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

Note it reconciles the **distinct keys**, not the rows. Comparing row counts across a one-to-many join tells you nothing; comparing the key sets tells you exactly who is on one side and not the other.

measure	value
attendance rows	70,245
roster rows	1,200
keys matched	1,200
attendance only	0
roster only	0

That is the table to paste into a notebook cell above every join. Four seconds to read, and it makes the two silent failure modes impossible to miss.

2.4 Guard the row count when you meant to preserve it

```
def join_preserving(left, right, on, how="left", **kwargs):
    before = len(left)
    out = left.merge(right, on=on, how=how, **kwargs)
    if len(out) != before:
        raise ValueError(f"join changed row count: {before} -> {len(out)}")
    return out
```

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

This is strictly stronger than `validate=`, because it also catches the case where the key is unique on both sides but the *left* table lost rows — which happens the moment someone changes `how="left"` to `how="inner"` while debugging something else and does not change it back.

2.5 The unmatched rate is an indicator, not an error

A join that fails to match 6% of a beneficiary list is telling you something about registration, not about your code. Report it.

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

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

Track the number across rounds and it becomes a data quality trend that belongs in a monthly report — the same move as the validation-finding series in the cleaning course. A match rate that falls from 98% to 91% between two rounds is a registration process that changed, and it is far more useful to know that in March than to discover it in an audit.

2.6 Normalise the key before you blame the join

A large share of "the join did not work" is a key that does not compare equal:

- **Type.** A facility code read as an integer on one side and a string on the other. `1042` never equals `"1042"`, and `"01042"` read as a number loses the leading zero permanently.
- **Whitespace and case.** `"FAC001 "` and `"fac001"` are three different keys.
- **Dates against date-times.** `2024-03-01` and `2024-03-01 00:00:00` compare equal in some libraries and not others; a period column is safer as a string.

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

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

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

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

Do this to both sides in the same function, so they cannot drift. Two separate cleaning expressions is the same defect as two separate indicator calculations.

2.7 The suffix collision

Both tables have a column called `updated_at`, or `district`, or `notes`. The join does not fail; it renames.

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

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

Both libraries default to something unhelpful — `_x` and `_y` in pandas, `.x` and `.y` in dplyr — and three joins later nobody can say whether `district_x` came from the register or the aggregate. **Name the suffixes after the source, every time.** It costs one argument and it is the difference between a traceable table and a guess.

The worse version is a shared column you did not know about, silently carried through the join and then used. Check before joining:

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

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

2.8 Keys with different names

Say so explicitly rather than renaming a column to make the join work — the rename outlives the join and confuses the next reader.

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

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

2.9 What comes next

Every check so far has assumed you know what one row of each table is. That assumption is where the remaining join failures live: a table of households joined to a table of people, or a monthly aggregate joined to a daily register. The next lesson is about naming the grain — and aggregating to it before the join, rather than discovering afterwards that you multiplied.

Part II

One row per what?

CHAPTER 3

One row per what?

Lesson 3 of 8 · 90 min

The grain of a table decides what every number computed from it means. Households against people, episodes against patients, and the many-to-many that turns 2,403 rows into 13,004 without anyone deciding it should.

3.1 Say the grain out loud

The **grain** of a table is what one row is. One household interview. One child screened on one day. One facility-month-antigen. One admission episode.

It sounds like a formality. It is the single most useful sentence you can write above a table, because almost every join failure and almost every denominator argument is really two people holding different grains in their heads.

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

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

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

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

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

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

Three comments, and they will prevent more damage than any assertion in this course. A join is only safe when you can name the grain of both sides and the grain of the result.

3.2 The grain decides what the number means

The WASH survey is 2,403 households covering 13,004 people, a mean household of 5.52. Two perfectly correct sentences come out of that file:

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

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

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

```
print(f"people in those households: {by_person:.1%}")
```

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

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

13.3% of households, 13.5% of people. And on open defecation, 13.4% of households against 13.9% of people.

The gaps are small here, and that is worth saying plainly: **on this file the choice barely moves the number.** It moves what the number *is*. Sphere states its water quantity standard per person, so the person-level figure is the one that answers the standard; a household-level figure answers "how many households are affected", which is what a distribution plan needs.

Where the gap is not small is where it matters most. Large households are systematically more likely to be short of water per person, so in a survey with more variation in household size the two figures separate — and then reporting the household figure against a per-person standard understates the problem.

Report the grain in the label. `share_of_households_below_15l` and `share_of_people_below_15l` cannot be confused; `water_below_minimum` can.

3.3 Expanding households to people

Sometimes you genuinely need one row per person — to join to an age-sex distribution, to feed a per-person weight, or to compute a population denominator from the survey itself.

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

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

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

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

2,403 rows become 13,004. Three things to hold about that operation.

- **It is a claim, not a transformation.** Every person in a household is now assumed to have the household's water access, sanitation and hygiene. For water quantity that is reasonable; for "who fetches the water" it is false.
- **Nothing about the individuals is real.** `person_index` is a position, not a person. Do not disaggregate by it, and do not let it look like an identifier.
- **The forty-six households with no recorded size vanish.** State that, or the expanded population is quietly 46 households short of the survey it came from.

3.4 Weighting is usually better than expanding

Expansion is memory-hungry and easy to misuse. Where you only need population-weighted statistics, weight instead:

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

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

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

Same answer, one table, and the weight is visible as a column — which means a reviewer can see what you weighted by. An expanded frame hides the weighting inside the row count.

3.5 Aggregate to the grain before you join

This is the operational rule the whole lesson exists for.

You have a daily attendance file and you want a per-student summary joined to the roster. The wrong order is to join first and aggregate second — it works, but it carries 70,245 rows through the join and makes the relationship many-to-one when it did not have to be.

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

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

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

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

Two gains, and the second is the real one. The join is now `one_to_one`, so the strongest possible check applies. And the aggregation is written where you can see its denominator: `days_marked` counts the days a mark exists, which is not the same as the days the school was open — the next lesson but one is entirely about that distinction.

3.6 The many-to-many nobody chose

A many-to-many is almost never intended. It arrives when the key is not the key you thought.

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

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

Every household in a community now matches every person in that community. Four hundred households of six people each, in one community, produce 400 × 2,400 rows — nine hundred and sixty thousand — from a file whose honest grain is 2,400 people.

The tell is always the same: a row count that is a product rather than a sum, and a total that is suspiciously round in orders of magnitude. `validate="one_to_many"` catches it before you have to notice.

3.7 Episodes, not people

One more grain worth naming, because it is where treatment programme figures go wrong. In a CMAM register a row is usually an **admission episode**, not a child. A child readmitted after relapse has two rows.

That means:

- **Caseload** is a count of episodes. Two admissions of the same child are two admissions, and reporting them as one understates the work done.
- **Children reached** is a count of distinct children, which is smaller.
- **Cured rate** has episodes in both numerator and denominator, and mixing an episode numerator with a child denominator produces a rate above 100% that someone will spend an afternoon explaining.

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

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

Publishing both numbers, with their labels, ends the argument before it starts.

3.8 Write the grain into the file name

A last small habit that pays. When you save an intermediate table, put the grain in the name:

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

Six months later, `summary.csv` tells you nothing and one of these tells you everything. It also makes a wrong join obvious at the point where someone reads two file names side by side.

3.9 What comes next

Grain answers what a row is. The next lesson answers what a *column* is — the flattened repeat group a form platform exports as `symptom_1`, `symptom_2`, `symptom_3`, and the pivot that turns it back into rows you can count.

CHAPTER 4

Long, wide, and the repeat group in between

Lesson 4 of 8 · 90 min

Pivot because the analysis needs the shape, not because the export arrived in it. The flattened repeat group, the DHIS2 pivot table, and the fill value that turns "not reported" into zero.

4.1 Two shapes, and what each is for

The same data, twice:

Long — one row per observation, one column per variable. 2,736 rows of facility, period, antigen, doses.

Wide — one row per unit, one column per measurement. 456 rows of facility and period, with six antigen columns.

Neither is correct in general. The rule that actually works:

> **Long is for computing. Wide is for reading.**

Group, filter, join and summarise on long data, because every one of those verbs takes a column name and long data has one column per concept. Pivot to wide as the last step before a human looks at it, because a person reading a table wants the six antigens side by side.

Most trouble in this lesson comes from doing the reverse — receiving a wide export and analysing it wide, which turns "compute the coverage for every antigen" from one line into six.

4.2 The flattened repeat group

A form platform asks a repeating question — every child in the household, every symptom observed, every water source used — and exports it as numbered columns:

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

This is one of the most common shapes in this sector and it is unusable as it stands. You cannot count members, cannot filter to children under five, and cannot join to anything, because the thing you want to work with is spread across columns instead of down rows.

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

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

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

Two pivots, not one. The first turns every numbered column into rows; the second turns the attribute names back into columns. Trying to do it in a single step is where people get stuck, because the column name carries two facts — which member, and which attribute — and they have to be separated before either can be used.

`names_pattern` in R and the regex extract in Python are doing that separation. Get the pattern right against the real column names before you write anything else; a silent `NA` here becomes a member who disappears.

4.3 The empty slots, and the ones that mean something

A form with room for eight members and a household of three exports five empty slots. Most of them are nothing. One of them is not.

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

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

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

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

The distinction: a slot where **every** attribute is missing is padding and should go. A slot where the age is missing but the sex is recorded is a real member with a missing age, and dropping it removes a person from the household. Filter on "the whole row is empty", never on one column.

Check the result against something the file already knows:

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

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

The reported household size and the number of member rows should agree. Where they do not, the unflattening dropped someone or the interview did.

4.4 Long to wide: the DHIS2 pivot

The vaccination extract arrives long — one row per facility, period and antigen — and that is the right shape to compute on. To put it in front of a district health officer, pivot:

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

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

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

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

2,736 rows become 456 — 38 facilities across 12 months — with six antigen columns. The arithmetic is worth checking every time: 2,736 divided by 6 antigens is 456, so nothing was aggregated away. When the two do not divide cleanly, the long table had duplicates on the pivot key, and `pivot_table` will have silently summed them.

pandas `pivot` raises on duplicates; `pivot_table` aggregates them. Reach for `pivot` first, precisely because you want the error. tidyr's `pivot_wider` warns and produces list-columns, which is uglier and equally informative.

4.5 The fill value that invents data

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

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

`fill_value=0` fills combinations that did not exist in the long data. For doses administered that is often right — a facility with no penta3 row genuinely gave no penta3 doses.

For a **rate** it is always wrong. A missing coverage cell means "not computed", and filling it with zero publishes a facility at 0% coverage that simply did not report. The cleaning course made this point about reporting; here it arrives through a different door, as a pivot argument nobody thinks of as a decision.

The safe default is to leave it missing and handle it explicitly:

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

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

4.6 The wide table you cannot filter

Once wide, a question like "which facility-months had fewer than ten doses of any antigen" needs six comparisons joined by `or`, and adding a seventh antigen means editing every one of them. The same question on long data is one line:

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

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

That is the test for whether you pivoted too early. **If your next operation names more than one column that holds the same kind of value, go back to long.**

4.7 Round-trip, and check it

Any reshape worth trusting is reversible. Assert it once while you are writing the code:

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

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

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

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

Row count and total. Two lines, and they catch a silently dropped antigen, a duplicate that got summed, and a fill value that invented rows.

4.8 What comes next

You can now put a table into whatever shape the question needs. The next unit is about the tables you have to bring in from outside — the population frame that supplies a coverage denominator, and the calendar that says which periods should have existed at all.

Part III

Making the pieces line up

CHAPTER 5

Attaching a population denominator

Lesson 5 of 8 · 90 min

Where a denominator comes from, how to join it without matching on a name, and why catchment populations do not add up to a district. Plus the coverage figure above 100% and what it is really telling you.

5.1 The numerator is the easy half

Counting what your programme did is bookkeeping. Deciding what to divide it by is the analysis, and it is where a coverage figure is won or lost.

The vaccination extract makes this concrete: it ships `target_population` in the file, one figure per facility, constant across the twelve months. That is convenient and it is also somebody's estimate, made at some point, by some method, and the whole of this lesson is about not treating it as a fact just because it arrived in a column.

5.2 Where a denominator actually comes from

Four sources, in roughly descending order of how often they are argued about.

- **A census projection.** The national statistics office publishes a census and a growth rate; every district population you use is a projection forward from the census year. In much of this sector that census is a decade old or older.
- **An administrative population frame.** The health ministry's own denominators by facility catchment, which is what `target_population` here is. Usually derived from the projection with a coefficient — 3.5% of the population for children under one, 20% for children under five, 4% for pregnant women.
- **A programme target.** How many people the programme planned to reach. A legitimate denominator for "did we do what we said", never for "what share of the population is covered".
- **A survey.** The most defensible and the least available, because a survey gives you a proportion with a confidence interval rather than a count.

Name the source in the column. `target_population_moh_2024` and `target_population_census_projecti` are different numbers, and the moment two of them are in the same folder without labels somebody will average them.

5.3 Join on the code, never on the name

The cleaning course made this argument about place names. It is worth repeating here because a population frame is where a name join does the most damage: the denominator is the one column where a failed match does not look like a failed match, it looks like a facility with no coverage.

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

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

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

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

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

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

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

The assertion is the important line. A left join that fails to match leaves a missing denominator; a missing denominator makes a coverage figure missing; and a missing coverage figure gets excluded from a mean, so the district average silently becomes the average of the places that matched.

5.4 The projection, and the year nobody states

A census projection is a base population and a growth rate. Compute it in the open:

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

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

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

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

Nine years at 2.4% is a factor of 1.24. **A quarter of your denominator is an assumption**, and it compounds: two districts projected from the same census with the same national rate will preserve their relative sizes exactly, which is precisely what does not happen where there has been displacement.

So when a coverage figure is challenged, the projection is usually the honest answer. Three habits that keep it defensible:

- Write the census year, the growth rate and its source next to the number.
- Use the **same** projection for every indicator in a report. Two indicators on two projections cannot be compared, and nobody will notice.
- Where displacement has happened, say that the projection does not account for it, and

give the direction of the error rather than pretending to correct it.

5.5 Catchment populations do not add up

A facility's catchment is the population it is *supposed* to serve. Sum the catchments of every facility in a district and you will not get the district.

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

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

Here that sum is 11,774 for the month. It is a usable district denominator only if the catchments partition the district — no overlap, no gaps — and they almost never do. Two facilities on either side of a town both count the town. A population between two catchments is counted twice or not at all.

The consequence is specific and common: **facility-level coverage is unreliable and district-level coverage is fine.** People cross catchment boundaries to be vaccinated, so a facility's numerator includes children from its neighbour's denominator. Aggregate to the level where the boundary crossing happens inside the unit, and the noise cancels.

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

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

5.6 Coverage above 100%

It happens constantly and it is never a programme exceeding its population. Four causes, in the order worth checking:

1. **The denominator is too small.** An out-of-date projection, or a catchment coefficient that assumes fewer under-ones than there are.
2. **The numerator includes people from outside.** Boundary crossing, or a campaign that drew people in from a neighbouring district.
3. **The grain is wrong.** Doses counted where children were meant — a child receiving three doses of a three-dose antigen is one child.
4. **A join multiplied something.** Everything in lesson 1.

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

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

Report it rather than capping it. A coverage figure clipped at 100% has thrown away the evidence that the denominator is wrong, and the denominator being wrong is the finding.

5.7 Rates per thousand, and the annualisation trap

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

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

Two traps live in the same place. First, a **monthly** coverage figure is not an annual one, and multiplying by twelve assumes every month reported — which the cleaning course showed is false in August and September here. Second, a rate per thousand needs its period stated in the label: `doses_per_1000_under1_per_month` is unambiguous, `rate` is not.

5.8 Write the denominator down

Every indicator you publish should carry a reference row:

Field	Value
Indicator	Penta3 coverage, district
Numerator	Penta3 doses administered, facilities reporting
Denominator	Surviving infants, MoH catchment figures summed to district
Source	MoH population estimates 2024, projected from 2015 census at 2.4%
Disaggregation	Month, facility type
Caveat	Reporting rate 29% in August; coverage computed on reporting facilities only

That table takes two minutes and settles the question permanently. It is the same move as the definitions file the foundations course writes beside a results table, and it is what the *Indicator Design and the LogFrame* course later builds into a full reference sheet.

5.9 What comes next

The denominator answers "out of how many people". The next lesson answers "out of how many periods" — building a complete calendar so that a facility which never reported in August appears as a gap rather than not appearing at all.

CHAPTER 6

The rows that were never written

Lesson 6 of 8 · 90 min

A missing row has no value to be missing. Build the grid the data should have filled, find where 1,755 attendance rows went, and learn why a complete grid still does not prove everyone reported.

6.1 A missing row has no value to be missing

Every check in the cleaning course looked at values: blanks, sentinels, implausible numbers. All of them share an assumption — that the row is there.

The failure this lesson deals with is different. A school that was closed has no attendance rows. A facility that did not report has no row in the extract. A month with no distribution has nothing at all. There is no blank cell to count, no `NA` to detect, and every summary you compute is over the periods that happened to be recorded.

The fix is always the same shape: **construct the grid of periods that should exist, join the data onto it, and let the holes appear as rows.**

6.2 Build the grid

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

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

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

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

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

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

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

1,200 students by 60 school days is a grid of 72,000. The file holds 70,245. **1,755 rows were never written.**

dplyr has a shorter route when the frame is already the data:

```r
complete <- attendance |>
  tidyr::complete(student_id, attendance_date)
```

`complete()` fills the cross-product of the values it finds, which is right when every combination should exist and wrong when a student joined mid-term — it will not invent days for a student the file never mentions before March. Build the grid explicitly when the universe is defined outside the data, which is the usual case.

6.3 Where all 1,755 went

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

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

school_id	missing rows	days
SCH07	915	15
SCH18	840	15

Two schools, fifteen days each, and $61 \times 15 + 56 \times 15 = 1{,}755$. **The grid accounts for every single missing row**, and it does so in a form that names the cause: two schools were shut for the same fifteen days in March.

That is the payoff. Before the grid, the missing rows were invisible. After it, they are two schools and a date range — a strike, a flood, an exam period, something a colleague can confirm in one phone call.

6.4 Absent is not absent

Now the decision, and it is the whole reason the lesson exists.

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

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

Filling the gaps with "absent" makes SCH07 and SCH18 look like a dropout emergency, because a quarter of their term is now recorded as every child missing every day. The correct handling is the opposite: those days were **not school days for those schools**, and they belong in neither numerator nor denominator.

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

grid = (
```

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

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

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

Build the grid per school, not per district. The universe of periods is a property of the reporting unit, and assuming a single shared calendar is how a closure becomes an absence.

> Deciding whether a gap is a zero, a missing value or a period that should not exist is not a technical question. It is the analysis, and it belongs in the log with its reason.

6.5 A complete grid does not prove everyone reported

The vaccination extract is the counter-example, and it is important because it looks like the good case.

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

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

$38 \times 12 \times 6 = 2{,}736$, and the file has 2,736 rows. The grid is perfect. And 642 of those rows carry `report_submitted` of `false` and zero doses.

So the grid check passes and the data is still full of holes — they are just holes with a row around them. **Two separate checks, and you need both:** is every expected row present, and does every present row contain a report.

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

```
vax |>
  summarise(
    reporting_rate = mean(report_submitted),
    doses  = sum(doses_administered[report_submitted]),
```

```
    target = sum(target_population[report_submitted]),
    .by = c(period, antigen)
  )
```

6.6 Period keys that sort

A small mechanical point that causes disproportionate trouble.

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

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

Use ISO period keys — `2024-08`, `2024-Q3`, `2024-W32` — everywhere a period is a key. `Aug`, `August` and `08/2024` all sort wrongly, and `08/2024` is ambiguous between two conventions that are both in daily use in this sector.

The month must carry its year. A twelve-month grid keyed on `month` alone silently merges August 2023 and August 2024 the first time two years of data are in the same folder.

6.7 Aligning a daily register to a monthly aggregate

The last shape in this lesson, and the one the next lesson builds on. To compare a line-level register with a monthly return, aggregate the register to the aggregate's grain — never the other way.

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

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

Two things to notice, because both are choices. The month a record belongs to comes from the **event date**, not from when the form was submitted — a late submission still belongs to the month it describes. And the last month in the file is very often partial, so a trend chart that includes it always appears to be falling.

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

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

Drop it or mark it, but never plot it silently next to complete months.

6.8 What comes next

You now have a register aggregated to the same grain as the monthly return. The next lesson puts the two side by side, works out why they disagree, and turns the difference into a table you can publish rather than an argument you have to win.

Part IV

The table you analyse

CHAPTER 7

When the register and the report disagree

Lesson 7 of 8 · 90 min

Recompute the indicator from the line-level register, put it beside what was reported upward, and decompose the difference by reporting unit instead of averaging it away.

7.1 Two numbers for the same thing

Somewhere there is a line-level register — daily attendance marks, a screening book, a patient register. Somewhere else there is a monthly figure that was reported upward from it. The two disagree, and the gap is the most informative number in this course.

It is informative because **it is bounded**. Unlike most data quality questions, this one has a right answer sitting in a drawer: the register is the source, the report is a claim about the register, and the difference between them is a measurable quantity rather than an opinion.

Donor audits call the ratio between them the **verification factor** — recount the source documents, divide the recount by what was reported. A verification factor of 1.0 means the reporting is accurate. Anything else is a number with a cause.

7.2 Recompute from the register

Aggregate the register to the reporting grain, exactly as the previous lesson did:

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

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

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

Note what `marked` counts. A mark exists where the value is `true` or `false`. Values of `Y`, `N` and blank are **not** marks under this definition — and that definition is a decision, which is the whole of the next section.

7.3 The coding decision that moves a denominator

One school recorded attendance with `Y` and `N` instead of `true` and `false`. Recompute both ways:

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

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

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

SCH09	Strict	Y/N mapped
Marks in denominator	2,015	2,865
Attendance rate	86.05%	85.72%

Read those two columns together, because the lesson is in the contrast. **The denominator grows by 42%. The rate moves by a third of a point.**

That is the most common shape of this problem and the reason it survives review so easily. Anyone checking the *rate* concludes the coding did not matter. Anyone who then reports "number of attendance days recorded" is out by 850. A defect that barely touches a ratio can be enormous in a count, and programme reporting is full of counts.

Across the whole file the same operation moves the district rate from 88.47% to 88.42% and the denominator from 69,101 to 69,974.

7.4 Put the two sources in one table

Never compare two numbers by looking at two printouts. Join them, keyed on the reporting unit and period, and compute the difference as a column.

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

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

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

A **full** join, deliberately. A unit that appears in the register and not in the report has stopped reporting; a unit in the report with nothing in the register is reporting figures nobody can trace. Both are findings and an inner join deletes both.

7.5 Decompose, do not average

The single most common mistake here is computing one district-wide verification factor. A district factor of 1.02 can be twenty schools at 1.00 and one at 1.40, and the average has hidden the only thing worth acting on.

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

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

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

Report the distribution and the outliers, never the mean alone. A tolerance band — anything outside 0.95 to 1.05 gets looked at — turns this from a number into a work list, which is what a supervisor can actually use.

7.6 Why two sources disagree

Four causes, and they call for completely different responses. Work through them in this order.

- **Different definitions.** The register counts marks; the report counts enrolled children. The most common cause by a distance, the least often suspected, and it is not a data quality problem at all — it is two indicators with one name.
- **Different periods.** The report covers a calendar month, the register was extracted on the 28th. Check the boundaries before anything else.
- **Different coverage.** The report includes a school the register extract excluded, or vice versa. The anti-join from lesson 1 answers this in one line.
- **Transcription.** Somebody added the column up by hand. This is the cause everyone assumes and the least frequent of the four.

Only the last is an error in the ordinary sense. The first three are reconciliation problems, and reporting them as "data quality issues" gets a supervisor blamed for something that was decided in a form design meeting.

7.7 Report the gap; do not reconcile it away

The temptation is to adjust the register figure until it matches the report, or to publish only the one that looks better. Publish both, with the difference:

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

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

Two rows and a sentence naming the largest contributor to the gap. This is what a data quality assessment is built from, and the *Data Quality Assessment* course — the next and last in this module — turns it into a routine with a sampling strategy and a corrective action plan.

> A gap you report is a finding. A gap you close quietly is a figure nobody can reproduce, including you, in six months.

7.8 What comes next

Every piece is now in place: joins you can prove, a stated grain, the right shape, a denominator, a complete calendar and a reconciled source. The last lesson assembles them into one analysis table — one row per unit of analysis, every column traceable to where it came from, built by a script that runs from raw files to final table in one command.

CHAPTER 8

One table, assembled on purpose

Lesson 8 of 8 · 75 min

Start from a spine of the units you must report on, attach one measure at a time, assert at every seam, and carry provenance columns so a reviewer can trace any figure back to the file it came from.

8.1 The table everything else is computed from

By the end of an assembly you want exactly one table: **one row per unit of analysis, one column per measure, and nothing else.** Every chart, every summary and every figure in the report comes off that table.

The discipline matters because the alternative is what usually happens — a notebook of eleven data frames named `df`, `df2`, `merged`, `merged_final` and `merged_final_v2`, where the table a figure was computed from is whichever one was in memory at the time. Nobody can reproduce that, including the person who wrote it.

8.2 Decide the unit of analysis before you write a line

The unit of analysis is the thing your report makes statements about. Write it down first, because it determines every join that follows.

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


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


If you cannot state it in one line, the analysis is not ready to be assembled. And if two figures in your report have different units — one per school, one per child — they belong to two tables, not one.

8.3 Build the spine first

The spine is the complete set of units you must report on, built from the authority, not from the data. Every measure is then attached to it.

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

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

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

Spine first is the single most useful habit in this lesson. A school-month with no data now exists as a row with missing measures, which is visible, instead of not existing, which is not. It is the same argument as the period grid, applied to the whole assembly.

8.4 Attach one measure at a time, and assert at every seam

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

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

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

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

Two properties worth having. The row count cannot change, so a fan-out is impossible by construction. And the **match rate is printed for every attachment** — an enrolment table that matched 100% of the spine and an attendance table that matched 92% tell you immediately where the holes are, before any figure is computed.

`attach()` looks like ceremony until the first time it fires. It will.

8.5 Carry provenance

Add columns that say where things came from and what they rest on.

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

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

It looks redundant when every row has the same value. It stops being redundant the moment two extracts are concatenated, a second district arrives, or someone opens the CSV a year later with no idea which version of the register it came from. **Constant columns are cheap; unlabelled numbers are not.**

8.6 Assert what must be true of the finished table

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

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

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

Four assertions, and each corresponds to a lesson in this course: the grain is unique, the rate is a rate, the numerator cannot exceed its denominator, and the spine is complete. Run them at the end of assembly, every time.

8.7 What does not belong in the table

- **Intermediate columns.** `present_x`, `_merge`, `key_clean` and the six columns you needed for one calculation. Drop them; they are how a table becomes unreadable.
- **Rows below your reporting threshold.** If you would suppress a cell of four cases in a published table, do not carry it into a file that gets emailed.
- **Anything person-level.** An analysis table at school-month grain has no business carrying a student identifier, and the day it does is the day it cannot be shared.

8.8 Save it with its grain in the name

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

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

And save the assembly script beside it, because the table is the output and the script is the method. If regenerating the table takes more than one command, regenerating it in three months will not happen.

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

Four scripts, run in order, from raw files to final table. That is the layout *Reproducible Analysis Workflows* builds out later in the programme; adopting it now costs nothing and saves the reconstruction.

8.9 Where this course leaves you

You can take several files that describe the same programme and produce one table you would defend column by column — with the joins proved, the grain stated, the denominator sourced, the calendar complete, and the difference from what was reported upward written down rather than argued about.

The last course in this module, *Data Quality Assessment*, takes the last of those and makes it a routine: verification factors across a sample of facilities, the five quality dimensions, and a report that names the corrective action, the owner and the deadline — which is what turns a gap you found into a gap that closes.

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/joining-and-reshaping

Generated July 27, 2026.