CASSION · COURSE

R and the Tidyverse for Programme Data

The same ground as the Python course, in R — projects and renv, readr and haven, the dplyr verbs, forcats and tidyr — taught on survey and registry exports, with the defaults that differ from pandas named rather than glossed.

Instructor	Pierre Robentz Cassion
Level	Beginner
Learner hours	20 h
Taught in	R
Updated	July 27, 2026
Sectors	Public Health · Nutrition
Standards and methodologies	Results-Based Management (RBM) · WHO Child Growth Standards

Read and run the course online at data-analysis.cassion.dev/courses/r-for-programme-data

What you will be able to do

- Set up an R project that reopens a year later on a different machine, with its package versions recorded and no absolute path in the code
- Read CSV, Excel, Stata and SPSS exports without losing a leading zero, a date or a value label, and say what each reader does with a missing-value code
- Use the dplyr verbs deliberately, and explain what group_by() and summarise() do to missing keys, unused levels and the grouping that survives the call
- Order categories with forcats so a table and a chart read the way the report needs rather than alphabetically
- Reshape a flattened repeat group with pivot_longer(), and wrap the whole cleaning in a function that runs unchanged on next quarter's export

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

This is an R course, not an analysis course. It assumes you already know what an indicator is and why a denominator matters — *Data Analysis Foundations* teaches that — and spends its time on the language: what the tidyverse verbs actually do, which of R's defaults will surprise you, and how to write something that still runs when you are not there.

R only. Its sibling, *Python for Programme Data*, covers the same ground in Python. Elsewhere on this platform every technique is shown in both languages; here the split is the point, because a team usually inherits one codebase rather than choosing between two.

If you have read the Python course, the most useful thing this one does is name the places where R's default is the **opposite** of pandas'. A missing value makes `mean()` return `NA` instead of being skipped. A missing group key becomes its own group instead of vanishing. An unused factor level survives instead of being dropped. None of those are better or worse; each is a different bet about what silence should mean, and knowing which bet you are inside is most of the job.

Everything runs against real platform datasets — the MUAC screening register from twelve communes in Artibonite, a DHIS2-shaped vaccination extract, a WASH household survey — with the leading zeros, sentinel codes and inconsistent coding those files actually carry.

Part I

A project that reopens

CHAPTER 1

An analysis that reopens a year later

Lesson 1 of 8 · 80 min

RStudio projects, here(), renv and the two habits — setwd() and a saved workspace — that make an R analysis run on exactly one machine on exactly one day.

1.1 The failure this lesson prevents

Someone asks you to explain last quarter's figure. You open the analysis, run it, and get a different number — or it does not run at all, because the package that produced it has moved on two versions.

That is not a bug in the analysis. It is the environment, and in R there are three specific habits that cause it.

1.2 The three habits to drop

1.2.1 `setwd()`

```
setwd("/Users/marie/Desktop/nutrition")
```

This runs on exactly one computer. It is the single most common reason a colleague's script fails on your machine, and the fix is not to edit the path — it is to stop having one.

1.2.2 `rm(list = ls())` at the top

It looks like starting clean and is not. It removes the objects in your workspace and leaves everything else: attached packages, options, the working directory, anything loaded from a `.Rprofile`. A script that needs it is a script that is not reproducible, and running it in a fresh session is the only real test.

Restart R instead — Ctrl/Cmd + Shift + F10 in RStudio. That is the clean start `rm(list = ls())` pretends to be.

1.2.3 A saved workspace

RStudio offers to save `.RData` when you quit and to reload it when you start. Turn both off:

```
Tools -> Global Options -> General
  Restore .RData into workspace at startup    [ ]
  Save workspace to .RData on exit            Never
```

A reloaded workspace means your session contains objects whose code you cannot see. The analysis appears to work because something from last week is still in memory, and it stops working the moment someone else runs it.

1.3 The project

An RStudio project is a directory with a `.Rproj` file in it. Opening the project sets the working directory to that directory, and that is the whole mechanism — but it is enough to make every path in your code relative to the project instead of to your home folder.

```
muac-analysis/
  muac-analysis.Rproj
  data/
    raw/            the export exactly as it arrived, never edited
    interim/
  outputs/
    tables/
    figures/
  R/
    read_register.R
    indicator_table.R
  renv.lock
  README.md
  .gitignore
```

`data/raw` is read-only. The export as it arrived is the only thing you cannot reproduce; everything else is regenerated by running the code. So it is never edited, never sorted in Excel "just to look", and a correction lands beside the original as a new versioned filename rather than replacing it.

1.4 `here()` rather than relative paths

A relative path like `"data/raw/muac.csv"` works from the project root and breaks in a notebook, in an R Markdown document knitted from a subdirectory, or when someone runs one line at a time from a different location.

```
library(here)

muac <- readr::read_csv(here("data", "raw", "muac-screening-artibonite-2024.v1.csv"))
```

`here()` finds the project root — the directory containing the `.Rproj` file, or a `.here` file, or a `.git` directory — and builds the path from there. The result is the same wherever the code is run from.

```
here()
#> [1] "/home/marie/muac-analysis"
```

Call `here()` once at the top of a script and use it everywhere. A path built by `paste0()` with a `/` in it will not work on Windows; `here()` handles the separator.

1.5 renv: the package versions, recorded

`here()` fixes paths. It does nothing about the packages, and the packages are where the number changes.

```
install.packages("renv")

renv::init()       # once per project
```

`renv::init()` gives the project its own library and writes `renv.lock`, a record of every package and its exact version. From then on:

```
renv::snapshot()   # after installing or upgrading anything
renv::restore()    # on another machine, or a year later
```

Commit `renv.lock`. It is the difference between "install the tidyverse" and "install the versions that produced this table".

1.5.1 Installing where there is no internet

The part most tutorials skip, and the part that matters on a deployment.

On a connected machine, download the sources once:

```
renv::init()
renv::snapshot()

# Fill a local cache with everything the lockfile names.
renv::install()
renv::isolate()
```

Then copy the project directory — including `renv/library` — to the field laptop. `renv::restore()` will use what is already there rather than reaching for a repository it cannot see.

For a machine that will need packages you have not yet installed, the general mechanism is a local repository:

```
# On the connected machine
dir.create("pkgs")
download.packages(c("dplyr", "readr", "haven"), destdir = "pkgs", type = "source")

# On the offline machine
install.packages(
  c("dplyr", "readr", "haven"),
  repos = NULL,
  type = "source",
  contriburl = paste0("file://", normalizePath("pkgs"))
)
```

Source packages need a compiler for anything with C or C++ in it, which is common. If the field machines are Windows, download the binaries instead (`type = "win.binary"`) on a Windows machine of the same R version.

1.6 Loading packages

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

Two things not to do:

Do not use `require()` in a script. It returns `FALSE` and continues when the package is missing, so the script fails later with a confusing error about an object that does not exist. `library()` stops there and tells you which package.

Do not call `install.packages()` from a script. A script that installs software as a side effect of being run is a script nobody can safely run twice. Installation is `renv::restore()`, once, deliberately.

Where two packages export the same name — `dplyr::filter()` and `stats::filter()`, `dplyr::lag()` and `stats::lag()` — say which you mean:

```
muac |> dplyr::filter(muac_mm < 125)
```

The `::` form is worth the characters in a script that will outlive your memory of what was attached.

1.7 Verifying the environment before trusting it

```
sessionInfo()
```

For an analysis whose numbers matter, assert rather than inspect:

```
stopifnot(getRversion() >= "4.2.0")
stopifnot(requireNamespace("dplyr", quietly = TRUE))
```

`|>`, the native pipe used throughout this course, needs R 4.1 or later. If your team is on an older R, `%>%` from magrittr does the same job and the code in these lessons works unchanged with it.

1.8 What to hand to a colleague

Commit	Do not commit
`R/`, `.Rproj`	`renv/library/`
`renv.lock`	`data/raw/` — see below
`README.md`	`outputs/`
`.gitignore`	`.Rhistory`, `.RData`

```
.Rproj.user/
.Rhistory
.RData
renv/library/
data/raw/
outputs/
```

Never commit raw beneficiary data, identified or pseudonymised. Git keeps every version of every file forever, and a repository that was internal on Monday can be shared on Friday. Credentials — a DHIS2 token, a KoboToolbox key — never appear in a script:

```
token <- Sys.getenv("DHIS2_TOKEN")
stopifnot(nzchar(token))
```

Keep the value in a `.Renviron` file that `.gitignore` covers, and document the variable's name — not its value — in the README.

1.9 What comes next

The project reopens and the packages are pinned. The next lesson reads an export into it: `readr` for CSV, `readxl` for Excel, and the column specification that stops a facility code from becoming a number.

CHAPTER 2

Reading an export with readr and readxl

Lesson 2 of 8 · 85 min

Column specifications instead of guesses, problems() as a habit rather than a rescue, and the Excel traps — multiple sheets, a header that is not row one, merged cells and serial dates.

2.1 readr guesses, and tells you what it guessed

```
library(readr)
library(here)

PATH <- here("data", "raw", "muac-screening-artibonite-2024.v1.csv")
muac <- read_csv(PATH)
```

`read_csv()` prints the specification it inferred. That message is not noise to suppress — it is the only moment the reader tells you what it decided, and on this register it decides eight things:

```
child_id       character
commune        character
screening_date date
age_months     numeric
sex            character
muac_mm        numeric
oedema         character
outcome        character
```

Most of that is right. Two parts are worth taking control of.

2.2 The guess uses the first 1,000 rows

`guess_max` defaults to 1,000. A column that is empty for the first thousand rows and numeric afterwards is guessed as logical — because `NA` is logical — and every later value becomes `NA`.

This is the readr equivalent of pandas' mixed-dtype warning, and it fails more quietly: you get a column full of missing values and a `problems()` table you did not look at.

```
# Deliberate: read everything as text, look, then convert.
peek <- read_csv(PATH, col_types = cols(.default = col_character()), n_max = 200)
print(peek)
```

Two hundred rows is enough to see the shape and cheap on a large file.

2.3 Declare the columns

```
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")
)
```

Three things that buys you.

Leading zeros survive. `col_character()` on an identifier is the whole lesson: a facility code `007` read as a number becomes `7`, the join to the facility list fails for exactly the codes that had a leading zero, and the failure looks like missing facilities rather than a type error.

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 text that happens to be written with digits.

The date format is stated. An inferred format can change between files as the mix of values changes. `01/02/2024` is either 1 February or 2 January and the file will not tell you which.

The sentinel is declared. This register codes an unmeasured MUAC as `-99`. Read without `na`, that is a number:

```
mean(muac$muac_mm)
#> [1] NA
```

Which brings us to the most important difference from pandas.

2.4 `mean()` returns NA, and that is a feature

pandas skips missing values by default; R does not.

```
mean(muac$muac_mm)
#> [1] NA

mean(muac$muac_mm, na.rm = TRUE)
#> [1] 139.92
```

R is loud where pandas is silent. The `NA` is a question: *do you know there are missing values here, and have you decided what they mean?* Answering it with `na.rm = TRUE` is fine — answering it without noticing you were asked is how a denominator quietly changes.

`na.rm = TRUE` drops the missing from the numerator **and** from the denominator. That is usually right for a mean and usually wrong for a coverage rate, where the unmeasured child is still a child who was screened. Count them:

```
sum(is.na(muac$muac_mm))
#> [1] 72
```

2.5 `na` is file-wide, not per column

```
muac <- read_csv(PATH, na = c("", "NA", "-99"))
```

readr applies `na` across every column. pandas can scope a sentinel to one column with `na_values = {"muac_mm": ["-99"]}`; readr cannot, so if another column legitimately holds `-99`, handle it afterwards:

```
muac <- read_csv(PATH, na = c("", "NA")) |>
  dplyr::mutate(muac_mm = dplyr::na_if(muac_mm, -99L))
```

`na_if()` is the scoped form and is worth preferring whenever more than one column is numeric.

2.6 `problems()` is a habit, not a rescue

When a value does not fit its declared type, readr does not stop. It records the row, the column, what it expected and what it found.

```
issues <- problems(muac)
nrow(issues)
#> [1] 0
```

Zero on this file, and that is the point of checking: **the number is only meaningful if you look at it every time.** A run that quietly parsed 200 values as `NA` looks identical to a clean one until you ask.

```
if (nrow(problems(muac)) > 0) {
  print(problems(muac))
  stop("Input did not match its column specification.")
}
```

In a script that produces a reported figure, that `stop()` is correct. Refusing to run beats producing a table with a silently empty column.

2.7 Verify the read before doing anything with it

Four checks, one minute, and they catch nearly everything:

```
library(dplyr)

check <- function(df, expected_rows = NULL) {
  cat("rows/cols ", nrow(df), ncol(df), "\n")
  print(dplyr::glimpse(df))
  print(colSums(is.na(df)))
  if (!is.null(expected_rows)) {
    stopifnot(nrow(df) == expected_rows)
  }
}

check(muac, expected_rows = 4218)
```

The row-count assertion is the one people skip. 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.

2.8 Separators, encodings and thousands

A file produced on a French or Spanish Windows machine is frequently semicolon-separated, because the comma is the decimal mark there.

```
epi <- read_csv2(PATH)                      # ; separator, , decimal
epi <- read_delim(PATH, delim = "\t")       # tabs
```

`read_csv2()` is the European variant — semicolon and comma — and it exists precisely because this is common enough to deserve its own function.

Encoding shows up in commune names: Gonaïves, L'Estère, Anse-Rouge.

```
muac <- read_csv(PATH, locale = locale(encoding = "UTF-8"))
muac <- read_csv(PATH, locale = locale(encoding = "latin1"))   # what Excel often writes
```

`latin1` never errors, because every byte is a valid latin1 character. That makes it a poor diagnostic and a decent fallback: if a file read as `latin1` shows `GonaÃ¯ves`, it was UTF-8 all along and you told the reader otherwise.

Thousands separators turn a number into text:

```
epi <- read_csv(PATH, locale = locale(grouping_mark = ","))
```

Without it, `target_population` arrives as `"1,480"` — a character column on which every arithmetic operation fails or concatenates.

2.9 Excel

```
library(readxl)

excel_sheets(path)
#> [1] "Cover" "Data" "Codebook" "Sheet3"

data <- read_excel(path, sheet = "Data")
```

`read_excel()` without `sheet` returns the **first** sheet, which in a programme workbook is usually a cover page. Always list the sheets first.

2.9.1 The header is not row one

```
data <- read_excel(path, sheet = "Data", skip = 3)
```

Programme workbooks routinely carry a title, a logo row and a blank line above the real header. The symptom is column names like `...1`, `...2`.

2.9.2 Merged cells

A merged cell holds its value in the top-left position and nothing in the rest. When a district name is merged across its facilities, only the first row of each district has one:

```
data <- data |> tidyr::fill(district, .direction = "down")
```

`fill()` is correct here and dangerous in general: it is only right because the blanks came from merging, not from non-response. Never apply it to a column where a blank might mean "not answered".

2.9.3 Serial dates

Excel stores dates as a number of days since 1899-12-30. `read_excel()` usually converts them, but a column typed as text in the workbook comes through as the raw serial.

```
data$screening_date <- as.Date(as.numeric(data$screening_date), origin = "1899-12-30")
```

If a date column arrives as five-digit numbers, this is why.

2.9.4 Types in Excel

```
data <- read_excel(
  path,
  sheet = "Data",
  col_types = c("text", "text", "date", "numeric", "text", "numeric", "text", "text")
)
```

`readxl` takes a positional vector rather than a named specification, so a column added to the export shifts everything after it. Check `ncol()` against the length of your vector, or read as text and convert with dplyr.

2.10 The read function, assembled

Everything above belongs in one function the scripts share:

```
# R/read_register.R
read_register <- function(path) {
  muac <- readr::read_csv(
    path,
    col_types = readr::cols(
      child_id       = readr::col_character(),
      commune        = readr::col_character(),
      screening_date = readr::col_date(format = "%Y-%m-%d"),
      age_months     = readr::col_integer(),
      sex            = readr::col_character(),
      muac_mm        = readr::col_integer(),
      oedema         = readr::col_character(),
      outcome        = readr::col_character()
    ),
    na = c("", "NA", "-99")
  )

  if (nrow(readr::problems(muac)) > 0) {
    print(readr::problems(muac))
    stop("Input did not match its column specification.")
  }
  stopifnot(!any(is.na(muac$child_id)))

  muac
}
```

One function, one place to fix when the next export changes.

2.11 What comes next

The CSV is in and its types are right. The next lesson handles the files that carry more than values — Stata and SPSS exports, where `1 = Yes` travels with the data and `haven` is the reason to read them in R even when you will analyse them somewhere else.

Part II

Data that keeps its meaning

CHAPTER 3

Stata and SPSS files keep their labels

Lesson 3 of 8 · 75 min

haven, the labelled class, and why a survey firm's .dta should be read in R even when the analysis happens elsewhere — plus the two ways to lose a codebook without noticing.

3.1 What a CSV throws away

A survey firm delivers a `.dta` or a `.sav`. Inside it, a column is not just values — it carries a **variable label** ("Household head consented to interview") and **value labels** (`1 = Yes`, `2 = No`, `9 = Don't know`).

Export it to CSV and both are gone. You get a column of `1`, `2` and `9`, a separate PDF codebook somebody has to keep, and a mean of 1.7 that means nothing at all.

```
library(haven)

survey <- read_dta(here("data", "raw", "household-survey.dta"))
```

`haven` reads the labels along with the values. That is the reason this lesson exists, and the reason to read a survey file in R even when the analysis will happen in Python: **R preserves more of the file than pandas does**, and a CSV plus a codebook you wrote yourself is a worse artefact than the original.

3.2 The `labelled` class

`read_dta()` gives you columns of class `haven_labelled`: the underlying values, with their labels attached.

```
class(survey$consent)
#> [1] "haven_labelled" "vctrs_vctr" "double"

attr(survey$consent, "labels")
#>       Yes        No Don't know
#>         1         2         9

attr(survey$consent, "label")
#> [1] "Household head consented to interview"
```

The values are still there — arithmetic works — which is exactly the trap. A labelled column is a number as far as `mean()` is concerned:

```
mean(survey$consent, na.rm = TRUE)
#> [1] 1.7
```

That figure is the average of `1`, `2` and `9`. It is meaningless and it does not warn.

3.3 Converting deliberately

`haven` gives you three moves, and which one is right depends on what the column is.

A categorical variable becomes a factor.

```
library(dplyr)

survey <- survey |>
  mutate(consent = as_factor(consent))

levels(survey$consent)
#> [1] "Yes" "No" "Don't know"
```

`as_factor()` — haven's, not base R's `as.factor()` — uses the value labels as levels, in the order the file declares them rather than alphabetically. That ordering is a gift: the codebook's order is usually the order a report wants.

A genuine number loses its labels.

```
survey <- survey |> mutate(hh_size = zap_labels(hh_size))
```

`zap_labels()` strips the labelling and leaves the values. Use it where the column really is a quantity that happened to carry a label.

Everything at once, when the file is uniformly categorical:

```
survey <- read_dta(path) |> as_factor()
```

Applying `as_factor()` to the whole tibble converts every labelled column. Fast, and wrong for any file mixing categories with quantities — check with `glimpse()` before reaching for it.

3.4 The two ways to lose a codebook

3.4.1 Reading with the labels already applied

```
survey <- read_dta(path, .name_repair = "unique")
survey <- as_factor(survey)          # values are gone; only labels remain
```

Now the underlying codes are unavailable. That matters more than it sounds: the codebook, the questionnaire, the tabulation plan and every previous analysis all refer to `1` and `2`. A colleague asking "how many were coded 9?" cannot be answered from a factor.

Keep both:

```
survey <- read_dta(path)

codes <- survey |> mutate(across(where(is.labelled), zap_labels))
labels <- survey |> as_factor()
```

3.4.2 Writing the CSV and moving on

```
readr::write_csv(as_factor(survey), "survey.csv")
```

The CSV now holds `"Yes"` and `"No"`, which is readable and lossy in a different direction — the codes are gone, and so is the distinction between "Don't know" and a genuine missing value.

If you must hand a CSV to a Python analysis, hand a codebook with it, generated from the file rather than typed:

```
codebook <- tibble::tibble(
  variable = names(survey),
  label    = vapply(survey, function(x) attr(x, "label") %||% NA_character_, character(1))
)

readr::write_csv(codebook, "survey-codebook.csv")
```

The value labels need one row per level, which `labelled::look_for()` produces directly if you have that package. Either way, the codebook is **derived**, not transcribed — a transcribed codebook drifts from the file it describes.

3.5 User-defined missing values

Stata and SPSS distinguish kinds of missing: refused, not applicable, not asked. `haven` preserves them as **tagged NA**s rather than collapsing them.

```
survey$income
#> <labelled<double>[3]>
#> [1]  4500 NA(a) NA(b)
```

`NA(a)` and `NA(b)` are still `NA` to `is.na()`, so nothing silently becomes a number. But the distinction survives, and it is often the finding: a question refused by 40% of respondents is a different problem from one that did not apply to them.

```
survey <- survey |> mutate(income = zap_missing(income))
```

`zap_missing()` flattens them to plain `NA` once you have counted them. Count first.

3.6 SPSS and the other formats

```
read_sav(path)     # SPSS .sav
read_por(path)     # SPSS portable
read_dta(path)     # Stata
read_xpt(path)     # SAS transport
```

All return the same labelled structure, so everything above applies unchanged.

Writing back is symmetrical, and occasionally necessary when a ministry expects a `.dta`:

```
write_dta(survey, "cleaned.dta")
```

Stata has variable-name rules R does not — no dots, 32 characters — and `write_dta()` will error rather than silently mangle. That is the behaviour you want.

3.7 Where this fits with the Python course

The *Python for Programme Data* course says, in its lesson on codes and sentinels, that this is the one place where the answer is "use the other language". This is the lesson it points at.

pandas reads a `.dta` and gives you either the codes or the labels, not both, and does not preserve tagged missing values at all. If the labels matter — and on a survey file they always do — read in R, convert deliberately, and export both the codes and a derived codebook.

3.8 What comes next

The labels are attached and converted into factors. The next lesson is about what a factor actually is in R, and about the ordering problem `as_factor()` solved for you here and will not solve for a column you build yourself.

CHAPTER 4

Factors, and the order a report needs

Lesson 4 of 8 · 75 min

What a factor really is, the as.numeric() trap that silently returns level positions, and the forcats verbs that put a JMP ladder in ladder order instead of alphabetical order.

4.1 A factor is an integer with a lookup table

```
service <- factor(c("basic", "limited", "unimproved", "basic"))

levels(service)
#> [1] "basic"      "limited"    "unimproved"

as.integer(service)
#> [1] 1 2 3 1
```

Underneath, R stores `1 2 3 1` and a character vector of levels. Everything that follows — the ordering problem, the `as.numeric()` trap, unused levels — comes from that one fact.

Notice the levels are **alphabetical**. Nothing about `basic`, `limited`, `unimproved` says that is the order, and it is exactly backwards for a JMP service ladder, where unimproved is worst and basic is better than limited. Left alone, every table and every chart built from that column reads in an order that means nothing.

4.2 The `as.numeric()` trap

This is the single most expensive factor mistake, and it does not warn.

```
age <- factor(c("10", "9", "8"))

as.numeric(age)
#> [1] 1 3 2
```

Those are **level positions**, not values. The levels are alphabetical — `"10"`, `"8"`, `"9"` — so `"10"` is position 1 and `"9"` is position 3.

```
as.numeric(as.character(age))
#> [1] 10 9 8
```

`as.character()` first, always. It arises whenever a numeric column was read as text — an Excel column with one stray `"n/a"` in it, a CSV read with `col_character()` — and converted to a factor along the way.

Watch for a mean that is suspiciously close to the number of categories.

4.3 Setting the order you want

```
library(forcats)
```

```
ladder <- fct_relevel(service, "unimproved", "limited", "basic")
levels(ladder)
#> [1] "unimproved" "limited"    "basic"
```

`fct_relevel()` names the order explicitly. Declare it once, near where the column is created, and every table, chart and model downstream inherits it.

Two forcats verbs earn their place immediately:

```
fct_infreq(source)      # most common level first
fct_reorder(name, value)  # order one factor by another column
```

`fct_reorder()` is what turns an unreadable bar chart into a readable one — the communes sorted by their GAM rate rather than by the alphabet — and it does it without a manual `levels =` you have to update when the data changes.

```
library(dplyr)

by_commune |>
  mutate(commune = fct_reorder(commune, gam_rate)) |>
  ggplot2::ggplot(ggplot2::aes(gam_rate, commune)) +
  ggplot2::geom_col()
```

4.4 Ordered factors

`fct_relevel()` sets the order of the levels. It does **not** make the factor comparable:

```
service[1] >= service[2]
#> Warning: '>=' not meaningful for factors
#> [1] NA
```

For a ladder, where "at least basic" is a real question, declare it ordered:

```
ladder <- factor(
  c("basic", "limited"),
  levels = c("unimproved", "limited", "basic"),
  ordered = TRUE
)

ladder[1] >= ladder[2]
#> [1] TRUE
```

Now `>= "basic"` computes the JMP coverage indicator directly. Without `ordered = TRUE` it returns `NA` with a warning, which — if the warning is not read — becomes a coverage figure of `NA` or, worse, a filter that selects nothing.

Use ordered factors sparingly. They change how models treat the variable (polynomial contrasts rather than dummies), which is rarely what you want in a regression. For a service ladder, a severity band or an IPC phase, the ordering is worth it. For a commune, it is not.

4.5 Unused levels, and the two defaults that disagree

A level with no rows is still a level. Whether it appears in your output depends on which function you ask, and R is not internally consistent about it.

```
d <- tibble::tibble(k = factor(c("a", "b", "a"), levels = c("a", "b", "c")))
```

```
nrow(dplyr::count(d, k))
#> [1] 2

length(table(d$k))
#> [1] 3
```

`count()` drops the empty level; `table()` keeps it. Both are right for different questions, and the question is the same one the *Python for Programme Data* course raises about `observed=`: reporting on facilities that submitted data wants the empty one gone, and reporting coverage against a list of facilities that *should* have submitted wants it there, because a facility with zero rows is the finding.

Say which you mean:

```
dplyr::count(d, k, .drop = FALSE)   # keep the empty level
droplevels(d$k)                     # remove levels with no rows
forcats::fct_drop(d$k)              # the forcats spelling of the same thing
```

4.6 Collapsing a long tail

Ten water sources is too many rows for a report table and too many colours for a chart.

```
source <- factor(wash$water_source)
length(levels(source))
#> [1] 10

levels(fct_lump_n(source, 4))
#> [1] "borehole" "piped-into-dwelling" "piped-into-yard" "public-tap" "Other"
```

`fct_lump_n()` keeps the four most common and folds the rest into `Other`.

Do not lump before computing an indicator. `Other` here mixes protected springs, which are improved, with surface water, which is not — so a coverage rate computed after lumping is wrong. Lump for presentation, on a copy, after the numbers are settled.

```
fct_recode(source,
  improved = "borehole",
  improved = "protected-well",
  unimproved = "surface-water"
)
```

`fct_recode()` is the explicit alternative and is what you want for a classification: it names every mapping, so a source that appears in the next export and is not in the list stays itself rather than silently joining a group.

4.7 Factors and missing values

`NA` is not a level, so it survives every reordering and appears in `table()` only if you ask:

```
table(muac$outcome, useNA = "ifany")
```

To make missingness an explicit category — which is often right in a report, where "not recorded" is a finding rather than a gap:

```
muac <- muac |> mutate(outcome = fct_na_value_to_level(outcome, level = "not recorded"))
```

Once it is a level it will be counted, plotted and summed like any other. That is the point, and it is also the risk: a rate computed over a factor that includes "not recorded" has a different denominator from one that does not.

4.8 Where factors bite in a join

```
left_join(muac, sites, by = "commune")
```

If `commune` is a factor in one frame and a character in the other, dplyr coerces and warns. If it is a factor in both **with different levels**, the join still works — dplyr compares the labels, not the integers — but the result's levels are the union, which can quietly reintroduce empty categories.

The simplest habit: **join on character columns, convert to factor afterwards.** Factors are a presentation and modelling device, not a key type.

4.9 What comes next

Columns now hold their meaning and their order. The next unit works them: the dplyr verbs, and then the grouping call where most indicator bugs are born.

Part III

The verbs

CHAPTER 5

The dplyr verbs, used deliberately

Lesson 5 of 8 · 85 min

filter, select, mutate and arrange — what each does with a missing value, why if_else() refuses what ifelse() silently accepts, and the across() that applies one rule to many columns.

5.1 Five verbs and a pipe

```
library(dplyr)

muac |>
  filter(!is.na(muac_mm)) |>
  mutate(gam = muac_mm < 125) |>
  group_by(commune) |>
  summarise(assessed = n(), cases = sum(gam)) |>
  arrange(desc(cases / assessed))
```

Each verb takes a data frame and returns one. The pipe passes the result along. That is the whole grammar, and the rest of this lesson is about what each verb does when the data is not clean — which is always.

`|>` is R's native pipe, available since 4.1. `%>%` from magrittr predates it and behaves near-identically for this kind of code; if your team's scripts use it, everything here works unchanged.

5.2 `filter()` drops missing values, silently

```
nrow(muac)
#> [1] 4218

nrow(filter(muac, muac_mm < 125))
#> [1] 319

sum(is.na(muac$muac_mm))
#> [1] 72
```

A comparison against `NA` yields `NA`, and `filter()` keeps only rows that are `TRUE`. So those 72 unmeasured children are gone, and nothing said so.

This is the same behaviour as pandas, and it is worth stating in both languages because it is the fastest way to change a denominator by accident. Be explicit about which you meant:

```
# Children measured below 125 mm.
filter(muac, muac_mm < 125)

# Children not known to be at or above 125 mm -- the unmeasured included.
filter(muac, is.na(muac_mm) | muac_mm < 125)
#> 391 rows
```

Seventy-two rows separate those two questions. Which one you want depends on whether an unmeasured child is a non-case or an unknown, and that is a decision for a cleaning log, not a default.

5.3 `select()` and the helpers

```
muac |> select(child_id, commune, muac_mm)
muac |> select(-oedema)
muac |> select(starts_with("muac"), ends_with("_date"))
muac |> select(where(is.numeric))
```

`select()` renames as it selects, which is the tidy way to fix an export's column names once at the top:

```
epi |> select(facility = facility_id, month = period, doses = doses_administered)
```

Two habits worth forming. **Select before you join**, so the result does not carry forty columns you did not want. And **never select by position** — `select(3:5)` breaks the day the export gains a column, and it breaks silently because the code still runs.

5.4 `mutate()` and the two conditional functions

```
muac <- muac |>
  mutate(
    gam = muac_mm < 125,
    sam = muac_mm < 115,
    band = case_when(
      muac_mm < 115 ~ "severe",
      muac_mm < 125 ~ "moderate",
      .default = "normal"
    )
  )
```

`case_when()` evaluates in order and the first match wins, so `< 115` must come first. Its `.default` catches **everything else including `NA`**, which is almost never what you want:

```
case_when(c(110, 130, NA) < 125 ~ "case", .default = "not")
#> [1] "case" "not"  "not"
```

The missing measurement became `"not"`. Handle it explicitly:

```
band = case_when(
  is.na(muac_mm) ~ NA_character_,
  muac_mm < 115  ~ "severe",
  muac_mm < 125  ~ "moderate",
  .default = "normal"
)
```

Putting the `is.na()` branch first is the habit. This is the same trap as `np.select`'s `default` in the Python course, and it is worth recognising in both places.

5.4.1 `if_else()` rather than `ifelse()`

```
ifelse(c(TRUE, FALSE), 1L, "x")
#> [1] "1" "x"
```

Base R's `ifelse()` silently coerced an integer and a string into a character vector. A column you believed was numeric is now text, and you find out three steps later when a sum fails.

```
if_else(c(TRUE, FALSE), 1L, "x")
#> Error: Can't combine `true` <integer> and `false` <character>.
```

dplyr's `if_else()` refuses. **Prefer it everywhere**, for the reason this course keeps returning to: an error where the mistake is made beats a wrong answer three steps later.

`if_else()` also takes a `missing =` argument, which is how you say what an `NA` condition should produce rather than letting it propagate.

5.5 `across()`: one rule, many columns

```
muac |>
  summarise(across(c(muac_mm, age_months), ~ mean(.x, na.rm = TRUE)))
#> # A tibble: 1 × 2
#>   muac_mm age_months
#> 1    140.       30.9
```

`across()` is what stops a cleaning script being fifteen near-identical lines.

```
survey |> mutate(across(where(is.character), stringr::str_squish))

survey |> mutate(across(starts_with("fcs_"), ~ if_else(.x > 7, NA_real_, .x)))
```

That second line is the food-security rule from the *Python for Programme Data* course, in one statement: a consumption value above seven days is impossible and must become missing rather than being clipped, across all eight food groups at once.

The `~ .x` form is a shorthand for a function of one argument. `\(x) ...` is the newer base R spelling and works identically:

```
survey |> mutate(across(starts_with("fcs_"), \(x) if_else(x > 7, NA_real_, x)))
```

5.6 `arrange()`

```
muac |> arrange(commune, desc(screening_date))
```

`NA` sorts **last** regardless of direction, which is usually what you want and occasionally hides a problem — a column that is entirely missing looks sorted.

Sorting arranges rows. It says nothing about whether the differences between adjacent rows are real, a point the *Nutrition programme dashboard* project makes at length where nine of twelve communes have overlapping confidence intervals.

5.7 `distinct()` and counting

```
n_distinct(muac$commune)
#> [1] 12

muac |> distinct(child_id, .keep_all = TRUE)
```

`distinct()` without `.keep_all = TRUE` returns only the named columns, which surprises people expecting deduplicated rows. With it, the **first** row of each group is kept — so sort first if which one survives matters.

```
muac |> count(commune, sort = TRUE)
```

`count()` is `group_by() |> summarise(n = n()) |> ungroup()`, and it is the right tool whenever that is all you want.

5.8 Joins, and the check that belongs beside them

```
daily <- attendance |>
  left_join(roster, by = "student_id")
```

The failure to guard against is a join that multiplies rows because the right side has duplicate keys. dplyr warns, but a warning in a long script scrolls past:

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

That assertion is the R equivalent of pandas' `validate="many_to_one"`, and it caught two students registered twice in the *School attendance* project — an unregistered transfer that left both registrations live.

`anti_join()` is the fastest way to see what did not match:

```
attendance |> anti_join(roster, by = "student_id") |> distinct(student_id)
```

An empty result means every attendance row found a student. A non-empty one is a list to take back to whoever maintains the roster.

5.9 What comes next

The verbs are in hand. The next lesson is the one they lead to and the one where most indicator bugs are born: `group_by()` and `summarise()`, and the three things R does with missing keys, unused levels and leftover grouping that pandas does the other way round.

CHAPTER 6

Where most indicator bugs are born

Lesson 6 of 8 · 90 min

group_by() and summarise() — the grouping that survives the call, the missing key that becomes its own group, and the three defaults that are the exact opposite of pandas'.

6.1 An indicator is two numbers

Almost every figure this sector reports is a numerator over a denominator, and almost every argument about a figure is an argument about the denominator. The practical consequence: **compute both in the same call.**

```
library(dplyr)

by_commune <- muac |>
  mutate(
    assessed = !is.na(muac_mm) | !is.na(oedema),
    gam = assessed & (muac_mm < 125 | oedema == "true")
  ) |>
  group_by(commune) |>
  summarise(
    screened  = n(),
    assessed  = sum(assessed, na.rm = TRUE),
    gam_cases = sum(gam, na.rm = TRUE),
    .groups = "drop"
  ) |>
  mutate(gam_rate = round(gam_cases / assessed, 3))
```

Three columns and every one is load-bearing. `screened` is how many children came; `assessed` is how many produced a usable measurement and is the denominator; `gam_cases` is the numerator. Two separate calculations drift — one gets a filter the other does not — and the ratio becomes a number nobody can reconstruct.

6.2 `n()` and the two ways to count

```
summarise(muac, rows = n(), measured = sum(!is.na(muac_mm)))
```

`n()` counts rows in the group, including those with missing values. Summing a logical counts the `TRUE`s. That distinction is the difference between the two denominators above, and it is worth saying out loud every time you use one.

`sum()` on a logical with `NA` in it returns `NA`, which is R being loud again:

```
sum(c(TRUE, NA))
#> [1] NA

sum(c(TRUE, NA), na.rm = TRUE)
#> [1] 1
```

Reach for `na.rm = TRUE` deliberately, and know that it removes the row from the count rather than counting it as `FALSE`. Those are different claims.

6.3 The grouping survives, and it is the commonest bug

This is the one to internalise. `summarise()` removes the **last** grouping variable and leaves the rest:

```
out <- muac |>
  mutate(month = format(screening_date, "%Y-%m")) |>
  group_by(commune, month) |>
  summarise(n = n())
```

```
`summarise()` has regrouped the output.
i Summaries were computed grouped by commune and month.
i Output is grouped by commune.
i Use `summarise(.groups = "drop_last")` to silence this message.
```

```
class(out)[1]
#> [1] "grouped_df"

dplyr::group_vars(out)
#> [1] "commune"
```

The result is **still grouped by commune**. Every verb after this silently operates per commune:

```
out |> mutate(share = n / sum(n))     # share within the commune, not the district
out |> arrange(desc(n))               # sorts within each commune
out |> slice_max(n, n = 5)            # top 5 per commune, not overall
```

Each of those produces a plausible number that answers a different question from the one you asked. Nothing errors.

Say what you want, every time:

```
summarise(..., .groups = "drop")        # ungrouped result -- the usual answer
summarise(..., .groups = "drop_last")   # keep all but the last (the default)
summarise(..., .groups = "keep")        # keep all groupings
```

Or avoid the state entirely with per-operation grouping, which dplyr gained in 1.1:

```
muac |>
  summarise(n = n(), .by = c(commune, month))
```

`.by` groups for that call only and returns an ungrouped result. **Prefer it** in new code: there is no leftover state, so there is no bug of this shape.

6.4 A missing key becomes its own group

Here R and pandas take opposite bets.

```
d <- tibble::tibble(g = c("a", NA, "a"), v = c(1, 2, 3))

d |> group_by(g) |> summarise(s = sum(v), .groups = "drop")
```

```
# A tibble: 2 × 2
  g         s
  <chr> <dbl>
1 a         4
2 <NA>      2
```

The `NA` group is **kept**. pandas drops it by default, so the same operation in the two languages gives you a different set of rows and a different total.

R's default is the safer one, and in this sector the missing key is usually the interesting one: a facility with no district recorded, a household with no site code, a child with no commune. But it means a table can arrive at a report with an `<NA>` row nobody meant to publish.

```
d |> filter(!is.na(g)) |> group_by(g) |> summarise(s = sum(v), .groups = "drop")
```

Excluding it is fine. Excluding it **without noticing** is how the parts stop summing to the whole. Assert instead, where the key should be complete:

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

6.5 Unused factor levels: `count()` drops, `table()` keeps

The second reversal, and R is not consistent with itself here — which is why the lesson on factors covers it and this one repeats it.

```
d <- tibble::tibble(k = factor(c("a", "b", "a"), levels = c("a", "b", "c")))

nrow(count(d, k))                # 2   -- the empty level is gone
nrow(count(d, k, .drop = FALSE)) # 3   -- kept
length(table(d$k))               # 3   -- kept
```

Both are right for different questions, and it is the same question the Python course raises about `observed=`. Reporting on facilities that submitted data wants the empty level gone; reporting coverage against a list of facilities that *should* have submitted wants it there, because a facility with zero rows is the finding.

Pass `.drop` explicitly in anything that produces a reported table.

6.6 Grouped `mutate()`: a group statistic on every row

`summarise()` collapses; `mutate()` on a grouped frame returns a value per original row. This is how you compare a row against its own group without a join:

```
muac |>
  group_by(commune) |>
  mutate(
    commune_mean = mean(muac_mm, na.rm = TRUE),
    vs_commune   = muac_mm - commune_mean
  ) |>
  ungroup()
```

That is the check the *SMART survey analysis* project uses to find a team measuring half a kilogram light.

Note the `ungroup()`. A grouped frame that escapes into the rest of a script is the same bug as the one above, arriving from a different direction.

6.7 `count()` for when that is all you want

```
muac |> count(commune, outcome, sort = TRUE)
muac |> count(commune, wt = muac_mm)          # sum a column instead of counting rows
```

`count()` groups, counts and ungroups in one call, so it cannot leave state behind. Where a count is all you need, it is the better verb.

6.8 Two keys, and the table a report wants

```
by_commune_month <- muac |>
  mutate(month = format(screening_date, "%Y-%m")) |>
  summarise(cases = sum(gam, na.rm = TRUE), .by = c(commune, month))

wide <- by_commune_month |>
  tidyr::pivot_wider(names_from = month, values_from = cases, values_fill = 0)
```

`values_fill = 0` is safe here because a commune-month with no cases genuinely had zero. It would be wrong for a rate, where the absence means "not computed", not "zero percent" — a distinction worth checking every time you reach for it.

6.9 Writing the result out

```
readr::write_csv(by_commune, here::here("outputs", "tables", "gam_by_commune.csv"))
```

For a table someone will read rather than compute on, write the definition with it:

```
definitions <- tibble::tibble(
  indicator   = "GAM",
  numerator   = "MUAC < 125 mm or bilateral pitting oedema",
  denominator = "children with a MUAC measurement or a recorded oedema assessment"
)

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

A number and its definition travel together, or the monthly argument about the denominator starts again.

6.10 The three reversals, in one place

	pandas	**R / dplyr**
Missing value in `mean()`	skipped silently	returns `NA` until you say `na.rm`
Missing group key	dropped	kept as its own group
Unused category	dropped by `observed=True`	kept by `table()`, dropped by `count()`

None is better. Each is a different bet about what silence should mean, and knowing which bet you are inside is most of the job.

6.11 What comes next

The table aggregates correctly and both halves of every rate come from one call. The next unit reshapes: `pivot_longer()` on the flattened repeat group a CommCare or Kobo export arrives as, and then the function that makes the whole cleaning run again next quarter.

Part IV

Reshaping and repeating

CHAPTER 7

Undoing a flattened repeat group

Lesson 7 of 8 · 80 min

pivot_longer() with .value, the one call that turns a CommCare or Kobo export back into one row per person — plus the pivot_wider() warning that means your key is not a key.

7.1 What a form platform does to a repeat group

A household questionnaire asks about each member. In CommCare, KoboToolbox or ODK the answers come back **flattened** — one row per household, with the repeat group spread across numbered columns:

```
hh    member_1_age  member_1_sex  member_2_age  member_2_sex
H1              34  f                        8  m
```

Nothing about that shape is analysable. You cannot compute a mean age, filter to children, or join to anything, because a person is not a row.

7.2 `pivot_longer()` with `.value`

```
library(tidyr)

long <- wide |>
  pivot_longer(
    starts_with("member_"),
    names_to = c("member", ".value"),
    names_pattern = "member_(\\d+)_(.*)"
  )
```

```
# A tibble: 2 × 4
  hh    member   age sex
  <chr> <chr>  <dbl> <chr>
1 H1    1         34 f
2 H1    2          8 m
```

One row per person, and `age` and `sex` are separate columns with their own types. That is the shape everything downstream expects.

The `.value` sentinel is the piece worth understanding. In `names_to`, an ordinary name — `"member"` — becomes a new column holding that part of the old column name. `".value"` is different: it says *this part of the name is the name of the output column.* So `member_1_age` splits into `member = "1"` and a value belonging to a column called `age`.

Without `.value` you would get a single `value` column mixing numbers and text, and a `name` column you then have to split and pivot again.

7.2.1 The pattern

`names_pattern` is a regular expression with one capture group per entry in `names_to`, in order. `"member_(\\d+)_(.*)"` captures the number and then everything after the second underscore.

Two alternatives when the names are simpler:

```
# Fixed separator, no regex needed.
pivot_longer(wide, starts_with("member_"),
             names_to = c("member", ".value"), names_sep = "_(?=[a-z]+$)")

# One value column, name kept whole.
pivot_longer(wide, starts_with("crop_"),
             names_to = "slot", names_prefix = "crop_", values_to = "crop",
             values_drop_na = TRUE)
```

`values_drop_na = TRUE` on that last one matters: a household with three crop slots and two crops has an empty third, and a `NA` row per unused slot is noise that changes every count.

7.3 Check the row count against what you expect

A pivot that produces the wrong number of rows is the easiest error to make and the hardest to see, because the result looks correct.

```
stopifnot(nrow(long) == sum(!is.na(wide$member_1_age)) + sum(!is.na(wide$member_2_age)))
```

More practically, count people per household and compare against the household size the form recorded:

```
long |>
  count(hh, name = "members_found") |>
  left_join(households |> select(hh, hh_size), by = "hh") |>
  filter(members_found != hh_size)
```

An empty result is the check passing. A non-empty one is a list of households where the roster and the reported size disagree — which is a data-quality finding before it is a pivot problem.

7.4 `pivot_wider()` and the warning that means something

The reverse operation builds the wide table a report wants:

```
by_commune_month |>
  pivot_wider(names_from = month, values_from = cases, values_fill = 0)
```

If more than one row shares the same key, tidyr does not error:

```
Warning: Values from `v` are not uniquely identified; output will contain
list-cols.
```

That warning means your key is not a key. The cell now holds a list of the several values that collided, and every downstream operation on it either fails strangely or silently operates on a list.

Do not add `values_fn = sum` to make the warning go away without first finding out why there were duplicates. It is usually one of three things: a re-submitted form, a facility

reporting twice in a month, or a grouping variable you forgot to include. Only the first is safe to sum.

```
by_commune_month |> count(commune, month) |> filter(n > 1)
```

That is the diagnostic, and it should come before the pivot rather than after the warning.

7.4.1 `values_fill`

```
pivot_wider(..., values_fill = 0)
```

Safe for a count, where an absent combination genuinely had zero. **Wrong for a rate**, where the absence means "not computed" rather than "zero percent" — a commune with no screening in March has an undefined GAM rate, not a GAM rate of 0%, and filling it with zero drags every average down.

7.5 Separating and uniting columns

```
tibble(x = c("2024-01", "2024-02")) |>
  separate_wider_delim(x, delim = "-", names = c("year", "month"))
```

```
# A tibble: 2 × 2
  year  month
  <chr> <chr>
1 2024  01
2 2024  02
```

`separate_wider_delim()` replaced the older `separate()`, and the improvement is that it is strict: a row with the wrong number of pieces is an error rather than a silent truncation. Where ragged input is expected, say so:

```
separate_wider_delim(x, "-", names = c("year", "month"), too_few = "align_start")
```

The union is `unite()`, useful for building a composite key before a join:

```
unite(df, "key", district, community, sep = "-", remove = FALSE)
```

`remove = FALSE` keeps the source columns, which you almost always still want.

7.6 Nesting, briefly

A repeat group can also stay nested rather than being flattened:

```
by_commune <- muac |>
  tidyr::nest(.by = commune)

by_commune$data[[1]]      # the full tibble for the first commune
```

That is the shape for running the same model or the same summary per commune without a loop. It is worth knowing it exists; for most programme reporting, `.by` in `summarise()` gets you there with less machinery.

7.7 Where this fits

The Python course covers the same reshaping under `melt` and `pivot`. The vocabulary differs, the failure modes do not: a repeat group that must become rows, a key that turns out not to be unique, and a fill value that is right for a count and wrong for a rate.

7.8 What comes next

The data has the right shape. The last lesson turns everything so far into a function another officer can run on next quarter's export without editing anything but its arguments.

CHAPTER 8

A function another officer can run

Lesson 8 of 8 · 90 min

Turning a script into functions with arguments and stopifnot(), the tidy evaluation you need to pass a column name, and the command-line script that runs on next quarter's export unedited.

8.1 The handover test

An analysis is finished when someone else can run it on next quarter's export without editing the code. Not "without much editing" — without editing.

That standard rules out the three habits every R script accumulates: a path written for one machine, a threshold changed by hand between runs, and a step that only works if you already ran the lines above it in the right order.

8.2 From script to function

The cleaning from the earlier lessons, as one function:

```
# R/clean_register.R

clean_register <- function(muac, gam_mm = 125, sam_mm = 115) {
  stopifnot(is.data.frame(muac))
  required <- c("child_id", "commune", "muac_mm", "oedema")
  missing <- setdiff(required, names(muac))
  if (length(missing) > 0) {
    stop("Input is missing columns: ", paste(missing, collapse = ", "))
  }

  muac |>
    dplyr::mutate(
      # A MUAC below 40 is centimetres in a millimetre field. Applied only
      # below a threshold no plausible measurement reaches.
      muac_mm = dplyr::if_else(muac_mm < 40, muac_mm * 10L, muac_mm),
      muac_mm = dplyr::if_else(dplyr::between(muac_mm, 80L, 220L), muac_mm, NA_integer_),
      oedema  = dplyr::recode(
        tolower(trimws(oedema)),
        "true" = TRUE, "y" = TRUE, "yes" = TRUE,
        "false" = FALSE, "n" = FALSE, "no" = FALSE,
        .default = NA
      ),
      assessed = !is.na(muac_mm) | !is.na(oedema),
      gam = assessed & (muac_mm < gam_mm | oedema),
      sam = assessed & (muac_mm < sam_mm | oedema)
    )
}
```

Four things about that shape are deliberate.

It takes a data frame and returns one. No file reading, no writing, no «-. That is what makes it testable and what lets a notebook and a script share it.

The thresholds are arguments with defaults. A threshold that is only ever changed by editing a line will eventually be changed and not changed back. One that is an argument is visible in the call that produced the output.

It validates its input. `stop()` with a message naming the missing columns beats a `NULL` propagating into a mutate and surfacing as an unrelated error two functions later.

Nothing is grouped on the way out. A grouped frame escaping a function is the bug from the grouping lesson, arriving from a new direction.

8.3 `stopifnot()` for what cannot happen

```
stopifnot(
  "every row needs an identifier" = !any(is.na(muac$child_id)),
  "MUAC must be plausible after cleaning" =
    all(is.na(muac$muac_mm) | dplyr::between(muac$muac_mm, 80, 220))
)
```

Named expressions became the error message, which is the difference between a useful failure and `Error:  ...  is not TRUE`.

Use `stop()` for "this input is not what I was promised" and `stopifnot()` for "this cannot happen unless the code is wrong". The distinction matters to whoever reads the failure at seven in the morning.

8.4 Passing a column name: tidy evaluation

This is the one piece of R that surprises people coming from Python, and it has exactly one thing to learn.

```
# Does not work.
rate_by <- function(df, group_col) {
  df |> dplyr::summarise(n = dplyr::n(), .by = group_col)
}

rate_by(muac, commune)
#> Error: object 'commune' not found
```

dplyr verbs evaluate their arguments **inside the data frame**. Your function's `group_col` is a variable in the function, not a column, so dplyr looks for a column called `group_col` and fails.

The fix is one operator:

```
rate_by <- function(df, group_col) {
  df |> dplyr::summarise(n = dplyr::n(), .by = {{ group_col }})
}

rate_by(muac, commune)
```

`{{ }}` — "embrace" — means *take what the caller wrote and evaluate it in the data.* It is the answer for almost every function you will write around dplyr.

For a column name arriving as a **string**, which is what a command-line argument gives you:

```
rate_by_name <- function(df, group_col) {
  df |> dplyr::summarise(n = dplyr::n(), .by = dplyr::all_of(group_col))
```

```
}

rate_by_name(muac, "commune")
```

`all_of()` errors if the column does not exist; `any_of()` silently skips it. For a script whose output is reported, `all_of()` is the one you want.

To name an output column from an argument:

```
count_as <- function(df, group_col, out_name) {
  df |> dplyr::summarise("{out_name}" := dplyr::n(), .by = {{ group_col }})
}
```

The `"{...}" :=` form is glue syntax inside a tidyverse verb. It is worth recognising; you will need it rarely.

8.5 The indicator function

```
indicator_table <- function(muac, gam_mm = 125, sam_mm = 115) {
  clean_register(muac, gam_mm, sam_mm) |>
    dplyr::summarise(
      screened  = dplyr::n(),
      assessed  = sum(assessed, na.rm = TRUE),
      gam_cases = sum(gam, na.rm = TRUE),
      sam_cases = sum(sam, na.rm = TRUE),
      .by = commune
    ) |>
    dplyr::mutate(
      assessment_rate = round(assessed / screened, 3),
      gam_rate = round(gam_cases / assessed, 3),
      sam_rate = round(sam_cases / assessed, 3)
    ) |>
    dplyr::arrange(dplyr::desc(gam_rate))
}
```

Note what it does **not** do: it does not drop communes with a low assessment rate. Dropping one would change the district denominator and nothing in the output would say so. Flagging is the caller's job, and the caller can see the rate.

8.6 Testing the part that computes

Once the computation is a function taking a frame, a test is three lines:

```
# tests/testthat/test-indicator-table.R
test_that("the denominator excludes unmeasured children", {
  muac <- tibble::tibble(
    child_id = c("a", "b", "c"),
    commune  = "X",
    muac_mm  = c(110L, 130L, NA_integer_),
    oedema   = c("false", "false", NA_character_)
  )

  out <- indicator_table(muac)

  expect_equal(out$screened, 3)
  expect_equal(out$assessed, 2)     # the unmeasured child is not a denominator
  expect_equal(out$gam_rate, 0.5)
```

```
})
```

That test encodes the decision from the grouping lesson — which children are in the denominator — in a form that fails if someone changes it. Worth writing for every indicator whose definition you had to argue about.

`usethis::use_testthat()` sets the directory up; `devtools::test()` runs it.

8.7 The command-line script

```
#!/usr/bin/env Rscript
# scripts/indicator_table.R

suppressPackageStartupMessages({
  library(optparse)
  library(readr)
})

source(here::here("R", "clean_register.R"))
source(here::here("R", "indicator_table.R"))

option_list <- list(
  make_option("--input", type = "character", help = "Screening register CSV."),
  make_option("--output", type = "character", help = "Directory for the tables."),
  make_option("--gam-threshold", type = "integer", default = 125L,
              help = "MUAC cut-off in mm for GAM [default %default].")
)

opt <- parse_args(OptionParser(option_list = option_list))
if (is.null(opt$input) || is.null(opt$output)) {
  stop("--input and --output are required.")
}

muac <- read_register(opt$input)
message("Read ", nrow(muac), " rows from ", opt$input)

table <- indicator_table(muac, gam_mm = opt$`gam-threshold`)

thin <- table$commune[table$assessment_rate < 0.8]
if (length(thin) > 0) {
  warning("Assessment rate below floor: ", paste(thin, collapse = ", "))
}

dir.create(opt$output, recursive = TRUE, showWarnings = FALSE)
destination <- file.path(opt$output, "gam_by_commune.csv")
write_csv(table, destination)
message("Wrote ", destination, " (", nrow(table), " communes)")
```

```
Rscript scripts/indicator_table.R \
  --input data/raw/muac-screening-artibonite-2024.v1.csv \
  --output outputs/tables
```

`message()` rather than `print()`: messages go to standard error, so the script's narration does not end up mixed into piped output. `warning()` rather than `message()` for the thin communes, because it is a condition a caller might want to escalate with `options(warn = 2)`.

8.8 Recording what produced the output

```
run_metadata <- function(opt) {
  list(
    generated_at  = format(Sys.time(), "%Y-%m-%dT%H:%M:%S%z"),
    input         = opt$input,
    gam_threshold = opt$`gam-threshold`,
    r_version     = as.character(getRversion()),
    dplyr         = as.character(packageVersion("dplyr"))
  )
}

jsonlite::write_json(run_metadata(opt), file.path(opt$output, "run.json"),
                     auto_unbox = TRUE, pretty = TRUE)
```

Six months later, "which threshold produced this table" has an answer that does not depend on anyone's memory.

Then the real test, from the first lesson:

```
rm -rf outputs/
Rscript scripts/indicator_table.R --input data/raw/muac.csv --output outputs/tables
git status
```

If that does not restore the outputs exactly, something is not reproducible.

8.9 Where this course ends

You can build a project that reopens a year later with its package versions recorded, read any export without corrupting it, keep the labels a survey file carries, put categories in the order a report needs, aggregate to a numerator and a denominator that came from one call, reshape a flattened repeat group, and hand the result to someone else as a script rather than as a favour.

What this course deliberately did not teach is what to compute — which indicator, which denominator, which threshold, and how to defend it. That is *Data Analysis Foundations*, and the sector courses under **Sector Analysis** on the programme roadmap take it further into the classifications your cluster holds you to.

If your team works in Python rather than R, *Python for Programme Data* covers the same ground in that language — and the table of reversals in lesson 6 is the fastest way to move between them.

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/r-for-programme-data

Generated July 27, 2026.