CASSION · COURSE

Python for Programme Data

The Python a monitoring and evaluation officer actually needs — environments, pandas, dates, categorical codes and file formats — taught on survey and registry exports rather than on toy data.

Instructor	Pierre Robentz Cassion
Level	Beginner
Learner hours	20 h
Taught in	Python
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/python-for-programme-data

What you will be able to do

- Install and reproduce a working Python environment on a machine with no reliable internet, and explain why a system Python is the wrong place to work
- Read a CSV, Excel or fixed-width export without losing a leading zero, misreading a date, or letting a missing-value sentinel enter an average
- Select, filter and group a pandas DataFrame deliberately, and explain what SettingWithCopyWarning is telling you rather than silencing it
- Compute ages, reporting periods and age-band disaggregations correctly, including the month arithmetic a growth standard depends on
- Write a script another officer can run next quarter on a different export, without editing anything but its arguments

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 a Python 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 itself: what pandas is actually doing, which of its defaults will hurt you, and how to write something that still runs when you are not there.

Python only. Its sibling on the roadmap, *R for Programme Data*, does the same job in R. Elsewhere on this platform every technique is shown in both languages, because a team is usually split across them; here the split is the point, and covering both in one course would halve the depth of each.

Everything runs against real platform datasets — the MUAC screening register from twelve communes in Artibonite, a DHIS2-shaped vaccination extract, a school attendance file — with the leading zeros, sentinel codes, inconsistent dates and duplicate registrations those files actually carry. There are no toy DataFrames of three rows, because the errors this course exists to prevent do not appear at three rows.

You will not need internet in the room. Lesson 1 is about making sure of that.

Part I

An environment that will still run

CHAPTER 1

A Python you can reinstall without internet

Lesson 1 of 8 · 80 min

Why the system Python is the wrong place to work, what a virtual environment actually is, and how to carry a working install to a field laptop that has never seen a package index.

1.1 The failure this lesson prevents

An analysis runs on your laptop. Three months later the same script is run by someone else, on a different machine, and it either fails to start or — much worse — produces a different number.

That is not usually a bug in the analysis. It is a bug in the environment: a different pandas version whose default changed, a package installed for one project that broke another, a system Python the operating system upgraded underneath you.

This lesson is about making the environment a thing you declare, rather than a thing that accumulates.

1.2 Do not work in the system Python

macOS and most Linux distributions ship a Python that the operating system uses for its own tooling. On a shared field laptop, someone has usually installed a second one from a website, and a third arrived with an office suite.

Installing packages into any of them is a bad idea for a specific reason: the operating system depends on the versions it shipped. Upgrading pandas to satisfy your script can break a system utility, and there is nothing to warn you.

Newer Python builds now refuse outright:

```
error: externally-managed-environment

x This environment is externally managed
+-> To install Python packages system-wide, try apt install
    python3-xyz, where xyz is the package you are trying to
    install.
```

That error is correct and should not be worked around. There is a flag that overrides it. Using it is how a laptop ends up needing to be reimaged.

1.3 What a virtual environment is

A virtual environment is a directory holding its own copy of the interpreter's package directory. Activating it changes where `python` and `pip` look. That is the whole idea.

```
python3 -m venv .venv
source .venv/bin/activate       # Windows: .venv\Scripts\activate
python -m pip install pandas
```

Two consequences worth internalising:

- **Per project, not per machine.** Two analyses needing different pandas versions coexist without either knowing about the other.
- **Disposable.** If an environment gets into a bad state, delete the directory and rebuild it. Nothing of value lives there — which is only true if your dependencies are declared somewhere else.

1.4 uv, and why it matters more here than elsewhere

`pip` resolves and downloads packages one at a time from the network. On a connection that drops, a half-finished install leaves an environment that is neither the old one nor the new one.

`uv` is a faster resolver that produces a **lockfile**: an exact list of every package and version, including the ones your packages depend on.

```
# Install uv itself, once, on a machine that does have internet.
curl -LsSf https://astral.sh/uv/install.sh | sh

uv init muac-analysis
cd muac-analysis
uv add pandas openpyxl
uv run python -c "import pandas; print(pandas.__version__)"
```

`uv add` writes two files. `pyproject.toml` records what you asked for — `pandas` — and `uv.lock` records what you got, down to the exact version of every transitive dependency. **Commit both.** The first is your intent; the second is what makes next quarter's install identical to this one.

On the field laptop:

```
uv sync                          # reads uv.lock, installs exactly those versions
```

conda is the reasonable alternative and is common in this sector, particularly where a geospatial stack is involved — GDAL is far easier to install through conda than through pip. If your organisation already standardises on conda, use conda; the principle is identical and only the commands change.

1.5 Installing where there is no internet

This is the part most tutorials skip, and it is the part that matters on a deployment.

Download the packages once, on a connected machine, targeting the platform the field laptop runs:

```
mkdir wheels
uv pip download pandas openpyxl --dest wheels
```

Copy the `wheels` directory onto a USB stick along with your project, then on the offline machine:

```
uv venv
uv pip install --no-index --find-links wheels pandas openpyxl
```

`--no-index` tells the installer not to contact the package index at all, and `--find-links` points it at the directory instead. The install is then entirely local.

> Wheels are platform-specific. A wheel downloaded on macOS will not install on a Windows laptop, and one built for Python 3.12 will not install into 3.11. Download on a machine matching the target, or pass `--python-platform` and `--python-version` explicitly.

1.6 Pinning the Python version too

The packages are pinned; the interpreter should be as well. A script written against 3.12 and run on 3.9 fails on syntax that did not exist yet.

```
# pyproject.toml
[project]
name = "muac-analysis"
requires-python = ">=3.11"
dependencies = ["pandas>=2.2", "openpyxl>=3.1"]
```

```
uv python install 3.12
uv python pin 3.12
```

1.7 Verifying the environment before trusting it

Before running an analysis on a machine you did not set up, check that you are where you think you are:

```
import sys
import pandas as pd

print(sys.executable)          # should be inside .venv, not /usr/bin
print(sys.version)
print(pd.__version__)
```

If `sys.executable` is `/usr/bin/python3`, the environment is not activated and you are about to install into the system Python. That single line has saved more laptops than any amount of documentation.

For an analysis whose numbers matter, assert it:

```
import sys

assert sys.version_info >= (3, 11), f"needs Python 3.11+, running {sys.version}"
```

1.8 What to hand to a colleague

Everything needed to rebuild the environment, and nothing that was built:

Commit	Do not commit
`pyproject.toml`	`.venv/`
`uv.lock`	`wheels/`
`README.md` with the two commands	`__pycache__/`

A `.gitignore` covering the right column is part of the project template in the next lesson. The README needs two lines, not two pages:

```
uv sync
uv run python scripts/indicator_table.py --input data/raw/muac.csv
```

1.9 What comes next

The environment is reproducible. The next lesson is about the project around it — where the raw data goes, where outputs go, and why a path written as `C:\Users\marie\Desktop\data.csv` guarantees the script runs on exactly one machine.

CHAPTER 2

A project someone else can open

Lesson 2 of 8 · 70 min

Where raw data, code and outputs live, why a hardcoded path guarantees the script runs on one machine, and the one rule that makes an analysis rerunnable — never write into the folder you read from.

2.1 The shape of a project

Almost every analysis in this sector has the same four kinds of file, and giving each a fixed home removes a category of question no one should have to ask.

```
muac-analysis/
  data/
    raw/            the export exactly as it arrived, never edited
    interim/        intermediate files, safe to delete
  outputs/
    tables/
    figures/
  scripts/
    read_register.py
    indicator_table.py
  notebooks/
    exploration.ipynb
  pyproject.toml
  uv.lock
  README.md
  .gitignore
```

The important line is the first one.

2.2 data/raw is read-only

The export as it arrived is the only thing in the project you cannot reproduce. Everything else — the cleaned table, the figures, the indicator tables — can be regenerated by running the code again. The raw file cannot: the server it came from will have moved on.

So it is never edited, never overwritten, never sorted in Excel "just to look at it". If a correction arrives, it lands beside the original as a new file with a new name rather than replacing it.

That naming is worth being deliberate about:

```
data/raw/muac-screening-artibonite-2024.v1.csv
data/raw/muac-screening-artibonite-2024.v2.csv
```

not

```
data/raw/muac_final.csv
data/raw/muac_final_v2.csv
data/raw/muac_FINAL_corrected(1).csv
```

An analysis that ran on v1 must keep reproducing on v1. If the filename is overwritten, last quarter's number can never be explained again — and explaining last quarter's number is a thing you will be asked to do.

2.3 Paths that work on someone else's machine

This is the single most common reason a colleague's script fails on your laptop:

```
# Runs on exactly one computer.
muac = pd.read_csv("C:/Users/marie/Desktop/muac-screening-artibonite-2024.v1.csv")
```

The fix is to write paths relative to the project, and to let Python work out where the project is.

```
from pathlib import Path
import pandas as pd

# __file__ is this script; its parent is scripts/, whose parent is the project.
PROJECT = Path(__file__).resolve().parent.parent
RAW = PROJECT / "data" / "raw"
OUTPUTS = PROJECT / "outputs"

muac = pd.read_csv(RAW / "muac-screening-artibonite-2024.v1.csv")
```

Three things that buys you:

- **It runs from anywhere.** `python scripts/indicator_table.py` and `python /home/marie/muac-analysis/sc` both work, because the paths are computed from the script's own location rather than from the shell's current directory.
- **It works on Windows.** `Path` joins with `/` in your source and produces `\` on Windows. Never build a path by string concatenation.
- **It is greppable.** Every file the script touches is under `RAW` or `OUTPUTS`, so a reader can see the inputs and outputs at a glance.

In a notebook there is no `__file__`. Set the project root explicitly in the first cell instead, and keep the rest of the notebook relative to it:

```
from pathlib import Path

PROJECT = Path.cwd().parent      # notebook lives in notebooks/
assert (PROJECT / "data" / "raw").exists(), f"not the project root: {PROJECT}"
```

The assertion matters more than it looks. A notebook run from the wrong directory otherwise fails several cells later with a confusing error about a missing column.

2.4 Outputs are disposable, and that is the test

If deleting `outputs/` and rerunning the scripts does not restore it exactly, something in the analysis is not reproducible — a manual step, an edit made in Excel, a cell run out of order.

```
rm -rf outputs/
uv run python scripts/indicator_table.py
git status                       # should show nothing unexpected
```

Make this a habit before handing anything over. It is the cheapest possible check and it catches the failure that is hardest to diagnose later.

Create output directories from the code rather than expecting them to exist:

```
OUTPUTS = PROJECT / "outputs" / "tables"
OUTPUTS.mkdir(parents=True, exist_ok=True)

indicator_table.to_csv(OUTPUTS / "gam_by_commune.csv", index=False)
```

`parents=True` creates intermediate directories; `exist_ok=True` means a second run does not fail. A colleague cloning the project gets the folders without being told to make them.

2.5 What goes in version control

Commit	Do not commit
`scripts/`, `notebooks/`	`data/raw/` — see below
`pyproject.toml`, `uv.lock`	`data/interim/`
`README.md`	`outputs/`
`.gitignore`	`.venv/`, `__pycache__/`

```
.venv/
__pycache__/
*.pyc
data/raw/
data/interim/
outputs/
.ipynb_checkpoints/
```

Never commit raw beneficiary data, identified or pseudonymised. Git keeps every version of every file forever; a file deleted in a later commit is still in the history, and a repository that was internal on Monday can be shared on Friday. If the data is synthetic or already public, committing it is a convenience. If it is not, the raw directory stays out and the README says where the file comes from.

Credentials follow the same rule and are stricter: a DHIS2 token, a KoboToolbox API key or a database password never appears in a script, a notebook or a commit.

```
import os

# Read from the environment; the value never enters the repository.
token = os.environ["DHIS2_TOKEN"]
```

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

2.6 Notebooks and scripts do different jobs

Both belong in a project and they are not interchangeable.

A **notebook** is for looking: exploring a new export, checking a distribution, producing a figure you will paste into a report. It runs out of order, holds state you cannot see, and diffs badly, which makes it a poor place for anything that must run identically next quarter.

A **script** is for producing: it runs top to bottom, takes arguments, and either succeeds or fails. When an exploration turns into a number someone will report, move it into a script.

The practical workflow is to explore in `notebooks/`, then move the settled part into `scripts/` and import it back:

```python
# scripts/read_register.py
from pathlib import Path
import pandas as pd

def read_register(path: Path) -> pd.DataFrame:
    return pd.read_csv(
        path,
        dtype={"child_id": "string", "commune": "string"},
        na_values={"muac_mm": ["-99"]},
    )
```

```python
# In the notebook
import sys
sys.path.append(str(PROJECT / "scripts"))

from read_register import read_register

muac = read_register(RAW / "muac-screening-artibonite-2024.v1.csv")
```

Now the notebook and the script read the file the same way, and there is one place to fix when the sentinel changes.

2.7 A README that is actually read

Two commands and three sentences beat a page nobody finishes:

```
# MUAC screening indicator tables

Produces GAM and SAM rates by commune from the Artibonite screening register.

## Run

    uv sync
    uv run python scripts/indicator_table.py \
        --input data/raw/muac-screening-artibonite-2024.v1.csv \
        --output outputs/tables

## Data

data/raw is not committed. Download the register from the programme
share and place it there under its versioned filename.
```

2.8 What comes next

The project has a shape and the paths survive a change of machine. The next unit gets the data in: reading a CSV, an Excel workbook and a fixed-width export without losing a leading zero or misreading a date.

Part II

Getting the export in

CHAPTER 3

CSV, Excel and fixed-width, read without damage

Lesson 3 of 8 · 90 min

Encodings, separators, multi-sheet workbooks, merged header rows and column specifications — the format-specific traps that corrupt a file before you have looked at a single value.

3.1 What this lesson assumes

Data Analysis Foundations covers the principle: declare your types, declare your sentinels, and never let a reader infer either. This lesson takes that as given and goes into the formats themselves, because each one fails differently and the failures are not obvious from the data afterwards.

3.2 CSV is not one format

`read_csv` has around fifty parameters. Four of them account for most of the damage.

3.2.1 The separator

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

```
import pandas as pd

# Wrong: everything lands in one column named after the whole header line.
wrong = pd.read_csv("export.csv")
print(wrong.shape)              # (2736, 1)

right = pd.read_csv("export.csv", sep=";", decimal=",")
print(right.shape)              # (2736, 7)
```

The symptom is unmistakable once you know it: a DataFrame with exactly one column. Check `.shape` immediately after every read.

3.2.2 The encoding

Accented characters in commune names — Gonaïves, L'Estère, Anse-Rouge — are where this bites.

```
# UnicodeDecodeError, or worse, silently mangled: "Gona\xefves"
muac = pd.read_csv(path)

muac = pd.read_csv(path, encoding="utf-8")       # what you want
muac = pd.read_csv(path, encoding="latin-1")     # what Excel often produces
```

`latin-1` never raises, because every byte is a valid latin-1 character. That makes it a poor diagnostic and a decent fallback: if a file read as `latin-1` shows `GonaÃ¯ves`, it was UTF-8 all along and you told the reader otherwise.

3.2.3 Thousands separators turn numbers into text

```
# target_population arrives as "1,480" and pandas reads it as a string.
epi = pd.read_csv(path, thousands=",")
```

Without `thousands`, that column is object dtype and every arithmetic operation on it either fails or concatenates. Check `.dtypes` after reading, every time.

3.2.4 Leading zeros

The single most expensive default in this sector.

```
# facility_id "007" becomes the integer 7; the join to the facility list fails
# for exactly the facilities whose code had a leading zero.
epi = pd.read_csv(path)

epi = pd.read_csv(path, dtype={"facility_id": "string"})
```

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.

A defensive habit for a file whose columns you do not yet know:

```
# Read everything as text first, look, then convert deliberately.
peek = pd.read_csv(path, dtype="string", nrows=200)
print(peek.head())
print(peek.columns.tolist())
```

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

3.3 Excel

3.3.1 There is more than one sheet

```
book = pd.ExcelFile(path)
print(book.sheet_names)
# ['Cover', 'Data', 'Codebook', 'Sheet3']

data = pd.read_excel(path, sheet_name="Data")
```

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

`sheet_name=None` returns a dictionary of every sheet, which is how you handle a workbook with one sheet per month:

```
sheets = pd.read_excel(path, sheet_name=None)
monthly = pd.concat(sheets, names=["sheet"]).reset_index(level="sheet")
```

3.3.2 The header is not row 1

Programme workbooks routinely carry a title, a logo row and a blank line above the real header.

```
data = pd.read_excel(path, sheet_name="Data", skiprows=3)
```

The symptom is column names like `Unnamed: 0`, `Unnamed: 1`. If you see those, the header row is somewhere else.

3.3.3 Merged cells produce missing values

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["district"] = data["district"].ffill()
```

`ffill` is correct here and dangerous in general: it is only right because the blanks were created by merging, not by non-response. Never apply it to a column where a blank might mean "not answered".

3.3.4 Excel 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.

```
# 45306 is 2024-01-15
data["screening_date"] = pd.to_datetime(
    data["screening_date"], unit="D", origin="1899-12-30"
)
```

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

3.4 Fixed-width files

Still common from older health information systems and from ministry extracts. There is no separator; each field occupies a fixed range of characters.

```
CH00854Terre-Neuve    2024-01-15022m133false
CH01207Gonaives       2024-01-15034f118false
```

```
COLSPECS = [(0, 7), (7, 22), (22, 32), (32, 35), (35, 36), (36, 39), (39, 44)]
NAMES = ["child_id", "commune", "screening_date", "age_months", "sex",
         "muac_mm", "oedema"]

muac = pd.read_fwf(
    path,
    colspecs=COLSPECS,
    names=NAMES,
    dtype={"child_id": "string", "commune": "string"},
)

muac["commune"] = muac["commune"].str.strip()
```

Three notes:

- **Ranges are half-open**, like Python slices: `(0, 7)` is characters 0 to 6. Off-by-one here shifts every subsequent field by one character and produces a file that looks almost right.

- **Strip the padding.** Fixed-width fields are space-padded, so `"Terre-Neuve "` will not compare equal to `"Terre-Neuve"`.
- **Get the spec from the documentation, not by counting on screen.** `read_fwf` can infer `colspecs`, and it infers them from whitespace — which fails the moment a field is full-width or a value contains a space.

3.5 Verify the read before doing anything with it

Every read should be followed by the same four checks. Together they take one minute and catch nearly everything above.

```
def check(df, expected_rows=None):
    print("shape     ", df.shape)
    print("dtypes    ", df.dtypes.value_counts().to_dict())
    print("missing   ", df.isna().sum().to_dict())
    print("first row ", df.iloc[0].to_dict())
    if expected_rows is not None:
        assert len(df) == expected_rows, f"expected {expected_rows}, got {len(df)}"

muac = pd.read_csv(
    RAW / "muac-screening-artibonite-2024.v1.csv",
    dtype={"child_id": "string", "commune": "string", "sex": "string"},
    na_values={"muac_mm": ["-99"]},
)
check(muac, expected_rows=4218)
```

- **shape** catches the wrong separator and a truncated download.
- **dtypes** catches leading zeros lost, thousands separators, and a numeric column read as text.
- **missing** catches a wrong encoding and a sentinel not declared.
- **first row** catches a header offset — if the first row looks like a header, it was one.

The assertion on row count 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.

3.6 Reading a large file

If a file will not fit comfortably in memory, read the columns you need rather than all of them:

```
COLUMNS = ["student_id", "attendance_date", "present"]
attendance = pd.read_csv(path, usecols=COLUMNS, dtype={"student_id": "string"})
```

`usecols` is a large saving on a wide export. Beyond that, `chunksize` returns an iterator of pieces you aggregate as you go:

```
totals = []
for chunk in pd.read_csv(path, chunksize=100_000, dtype={"student_id": "string"}):
    totals.append(chunk.groupby("student_id")["present"].sum())

per_student = pd.concat(totals).groupby(level=0).sum()
```

The 70,000-row attendance file does not need this. A multi-year DHIS2 extract does.

3.7 What comes next

The file is in memory with its shape and types intact. The next lesson deals with what is written *inside* the columns: sentinel codes, categorical vocabularies, boolean values recorded five different ways, and the value labels a Stata file carries that a CSV throws away.

CHAPTER 4

Codes, sentinels and the 99 that becomes a mean

Lesson 4 of 8 · 85 min

Missing-value sentinels, categorical vocabularies, booleans recorded five ways, and Stata value labels — turning what the form recorded into what pandas can compute on.

4.1 A number that means "no answer"

Paper forms and the systems built to mirror them do not have a blank. They have a code. `99` means "not answered", `88` means "not applicable", `-99` means "not measured", and `999` means someone needed a third one.

None of them are values. All of them are numbers as far as pandas is concerned.

```
import pandas as pd

muac = pd.read_csv(RAW / "muac-screening-artibonite-2024.v1.csv")
print(muac["muac_mm"].mean())        # several millimetres below the truth
```

The result is wrong and, worse, *plausible*. Nothing about it looks like an error, and a MUAC mean is not a figure most readers can sanity-check by eye. A `99` in an age column is even quieter — it is a possible age.

4.1.1 Declare sentinels at read time

```
muac = pd.read_csv(
    RAW / "muac-screening-artibonite-2024.v1.csv",
    dtype={"child_id": "string", "commune": "string"},
    na_values={"muac_mm": ["-99"]},
)

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

Scope the sentinel to the column. `na_values=["-99"]` without a dictionary applies it to every column in the file, which is right until a column legitimately holds `-99`.

4.1.2 The sentinel list is documentation, not folklore

Sentinels come from the form's codebook. Write them down where the code can see them:

```
SENTINELS = {
    "muac_mm": ["-99"],           # not measured
    "age_months": ["99", "-1"],   # not answered / not applicable
}

muac = pd.read_csv(path, na_values=SENTINELS)
```

A sentinel you did not know about is invisible. This is worth one direct check per numeric column before trusting it:

```
for column in ["muac_mm", "age_months"]:
    print(column, sorted(muac[column].dropna().unique())[:5],
          sorted(muac[column].dropna().unique())[-5:])
```

Values clustered at the extremes — `98, 99` at the top of an age column, `-1` at the bottom — are sentinels, not observations. A histogram with a spike at exactly 99 is the same signal.

> Declaring a sentinel is not deciding what to do about the missing value. That decision — drop, impute, report separately — belongs to the analysis, and it has to be written down. Here you are only stopping a code from pretending to be a measurement.

4.2 Categorical vocabularies

`outcome` in the MUAC register takes four values and `sex` takes two. Declaring the vocabulary buys you an error when something outside it appears.

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

muac["outcome"] = muac["outcome"].astype(OUTCOMES)
print(muac["outcome"].isna().sum())
```

Count the missing immediately after converting. This is the trap: a value outside the declared categories becomes `NaN` rather than raising. A jump from zero to nine means nine rows carried a value you did not know about, and you want to see that now rather than discover it as a gap in a table three steps later.

To see what they were, compare before and after:

```
raw_values = set(muac_raw["outcome"].dropna().unique())
declared = set(OUTCOMES.categories)
print("unexpected:", raw_values - declared)
```

4.2.1 Ordered categories

Some vocabularies have an order, and declaring it makes comparison work:

```
LADDER = pd.CategoricalDtype(
    ["surface-water", "unimproved", "limited", "basic", "safely-managed"],
    ordered=True,
)

wash["service"] = wash["service"].astype(LADDER)
at_least_basic = wash["service"] >= "basic"
```

Without `ordered=True` that comparison raises. With it, the JMP ladder sorts and plots in the right order rather than alphabetically, which is the difference between a readable chart and one that puts "basic" between "unimproved" and "limited".

4.3 Booleans recorded five ways

The `oedema` column is the standard example. Two communes recorded `Y` and `N` in the first quarter; the rest recorded `true` and `false`.

```
print(muac["oedema"].value_counts(dropna=False))
```

Never cast such a column directly:

```
# Wrong in a way that is hard to see: every non-empty string is truthy,
# so "false" becomes True.
muac["oedema"] = muac["oedema"].astype(bool)
```

Map it explicitly, with an allow-list:

```
BOOLEANS = {
    "true": True, "TRUE": True, "Y": True, "y": True, "yes": True, "1": True,
    "false": False, "FALSE": False, "N": False, "n": False, "no": False, "0": False,
}

muac["oedema"] = (
    muac["oedema"].astype("string").str.strip().str.lower()
    .map({k.lower(): v for k, v in BOOLEANS.items()})
)

print(muac["oedema"].isna().sum())    # anything the map did not cover
```

An allow-list rather than a heuristic, for the reason that makes this lesson worth a session: **anything outside it stays missing and gets counted**, rather than being guessed at. A value the map does not cover is a question for whoever entered it.

Use pandas' nullable boolean when a genuine "not recorded" exists:

```
muac["oedema"] = muac["oedema"].astype("boolean")     # True / False / <NA>
```

`bool` cannot hold missing; `boolean` can. In this register 44 records have no oedema assessment at all, and collapsing those to `False` would silently assert that 44 children were checked and found clear.

4.4 Text that should be one value

Free-text-ish columns arrive with case, spacing and spelling variation. The WASH survey has a district written four ways.

```
print(wash["district"].value_counts())
# Nord-Ouest    727
# NORD-OUEST     33
# Nord Ouest     22
# nord-ouest     20
```

Ungrouped, that splits the district into four fragments, none of which looks alarming.

```
wash["district"] = (
    wash["district"].astype("string").str.strip().str.lower()
    .str.replace(r"\s+", "-", regex=True)
)
print(wash["district"].nunique())     # 3
```

Normalise **before** the first `groupby`, not after you notice the totals do not add up. And normalise into a canonical form you choose, rather than picking whichever spelling was most common — the most common spelling can change with the next export.

Where the variation is not mechanical — `Gonaives` versus `Gonaïves`, `St-Marc` versus `Saint-Marc` — a mapping table is the honest answer:

```
CANONICAL = {
    "gonaives": "Gonaïves",
    "st-marc": "Saint-Marc",
    "saint-marc": "Saint-Marc",
}
wash["district"] = wash["district"].map(CANONICAL).fillna(wash["district"])
```

Keep that table in the code, not in your head. It is a documented decision that someone will need to check.

4.5 Stata and SPSS carry their labels with them

A `.dta` from a survey firm holds both the code and its meaning: `1 = Yes`, `2 = No`, `9 = Don't know`.

```
survey = pd.read_stata(path)                          # values converted to labels
survey = pd.read_stata(path, convert_categoricals=False)   # raw codes
```

pandas gives you one or the other. To keep both — which is what you want, because the code is what the codebook documents and the label is what a reader needs — read the metadata separately:

```
with pd.io.stata.StataReader(path) as reader:
    labels = reader.value_labels()
    variables = reader.variable_labels()

print(variables["hh_size"])
print(labels.get("consent", {}))
```

R's `haven` preserves labels more completely than pandas does. If you are handed a `.dta` from a survey firm and the labels matter, reading it in R and writing a CSV plus a codebook is a legitimate step — and this is the one place in this course where the answer is "use the other language".

4.6 The read function, assembled

Everything above belongs in one function the notebook and the scripts share:

```
from pathlib import Path
import pandas as pd

SENTINELS = {"muac_mm": ["-99"]}
OUTCOMES = pd.CategoricalDtype(
    ["no-action", "referred-tsfp", "referred-otp", "referred-sc"]
)
BOOLEANS = {"true": True, "y": True, "yes": True, "1": True,
            "false": False, "n": False, "no": False, "0": False}

def read_register(path: Path) -> pd.DataFrame:
    muac = pd.read_csv(
        path,
        dtype={"child_id": "string", "commune": "string", "sex": "string"},
        na_values=SENTINELS,
```

```
    )
    muac["screening_date"] = pd.to_datetime(
        muac["screening_date"], format="%Y-%m-%d", errors="raise"
    )
    muac["outcome"] = muac["outcome"].astype(OUTCOMES)
    muac["oedema"] = (
        muac["oedema"].astype("string").str.strip().str.lower()
        .map(BOOLEANS).astype("boolean")
    )

    assert muac["child_id"].notna().all(), "every row needs an identifier"
    return muac
```

One function, one place to fix when the next export introduces a fifth way of writing "no".

4.7 What comes next

The table is in memory and every column means what it says. The next unit works it: selecting and filtering without the warning nobody reads, then grouping to the numerator and denominator an indicator actually needs.

Part III

Working the table

CHAPTER 5

Selecting and filtering, and the warning that became an error

Lesson 5 of 8 · 85 min

loc, iloc and boolean masks used deliberately — plus what SettingWithCopyWarning was warning about, why pandas 3 replaced it with ChainedAssignmentError, and what that changes for code you already have.

5.1 Three ways to select, and when each is right

```
import pandas as pd

muac = read_register(RAW / "muac-screening-artibonite-2024.v1.csv")
```

By label, with `loc`. Rows by index value, columns by name. This is the one to reach for by default, because it says what it means.

```
muac.loc[muac["commune"] == "Gonaives", ["child_id", "muac_mm", "outcome"]]
```

By position, with `iloc`. Rows and columns by integer position.

```
muac.iloc[0]           # first row
muac.iloc[:5, :3]      # first five rows, first three columns
```

`iloc` is for looking, not for logic. Position depends on the sort order of the file, and a script that says `iloc[:, 3]` breaks silently the day the export gains a column. Reserve it for inspection and for genuinely positional work.

A single column is a Series; a list of columns is a DataFrame:

```
muac["muac_mm"]              # Series
muac[["muac_mm"]]            # DataFrame with one column
muac[["commune", "muac_mm"]] # DataFrame with two
```

The double bracket is not a stylistic choice. Some functions need a DataFrame, and a Series will fail at a distance from where the mistake was made.

5.2 Boolean masks

A mask is a Series of `True`/`False` the same length as the frame.

```
severe = muac["muac_mm"] < 115
print(severe.sum())              # how many
print(severe.mean())             # what share
muac.loc[severe]
```

`.sum()` and `.mean()` on a boolean mask are the count and the proportion. That is the fastest way to answer "how many" and "what share" without a `groupby`.

5.2.1 Combining conditions

```
# Parentheses are mandatory: & binds tighter than <
under_five = (muac["age_months"] < 60) & (muac["muac_mm"] < 125)

# Use & | ~ -- not and / or / not, which cannot operate elementwise
either = (muac["muac_mm"] < 115) | (muac["oedema"] == True)
```

Writing `and` here raises `ValueError: The truth value of a Series is ambiguous`. The message is opaque the first time and always means the same thing: you used a scalar operator on a vector.

5.2.2 Missing values are not False

This is the trap that changes a caseload.

```
print(len(muac))                                  # 4218
print((muac["muac_mm"] < 125).sum())              # counts only measured children
print(muac["muac_mm"].isna().sum())               # the ones excluded silently
```

A comparison against `NA` yields `NA`, and `NA` is not selected. So a filter quietly drops every unmeasured child — which is correct if you meant "children measured below 125 mm" and wrong if you meant "children not known to be above 125 mm".

Be explicit about which you meant:

```
measured = muac["muac_mm"].notna()
gam = measured & (muac["muac_mm"] < 125)

print(f"assessed: {measured.sum()}, GAM cases: {gam.sum()}")
print(f"GAM rate: {gam.sum() / measured.sum():.3f}")
```

The denominator is the point. Dividing by `len(muac)` instead of `measured.sum()` reports a lower prevalence for a reason that has nothing to do with nutrition.

5.2.3 `isin`, `between`, `query`

```
north = muac["commune"].isin(["Gonaives", "Terre-Neuve", "Anse-Rouge"])
plausible = muac["muac_mm"].between(80, 220)
```

`between` is inclusive on both ends by default, and — the important part — it returns `False` for `NA`. So `~muac["muac_mm"].between(80, 220)` counts a missing measurement as an impossible one. Those are different failures and only one of them is a data-entry error:

```
impossible = muac["muac_mm"].notna() & ~muac["muac_mm"].between(80, 220)
```

`query` reads well for long conditions and is worth knowing:

```
muac.query("age_months < 60 and muac_mm < 125")
```

It is slower and it hides typos in a string, so prefer masks in a script that must fail loudly.

5.3 The warning that became an error

Older pandas produced a warning that generations of analysts learned to silence:

```
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer, col_indexer] = value instead
```

It was warning about **chained assignment** — selecting, then assigning to the selection:

```
muac[muac["commune"] == "Gonaives"]["muac_mm"] = 0
```

The first bracket produces a new object. Whether the assignment reached the original frame depended on the memory layout, which meant the same line could work on Monday and not on Tuesday. The warning existed because pandas could not tell you which had happened.

5.3.1 What pandas 3 does instead

Copy-on-Write is always on and cannot be disabled. Two things follow.

The behaviour is now defined. A selection never writes back to its parent — not sometimes, never. `SettingWithCopyWarning` no longer exists; `pd.errors.SettingWithCopyWarning` raises `AttributeError`.

Chained assignment tells you it did nothing.

```
import pandas as pd

df = pd.DataFrame({"a": [1, 2, 3], "b": [10, 20, 30]})
df[df["a"] > 1]["b"] = 0
```

```
ChainedAssignmentError: A value is being set on a copy of a
DataFrame or Series through chained assignment. Such chained
assignment never works ...
```

`ChainedAssignmentError` is a `Warning` subclass, so by default the line prints and continues. In a script whose numbers matter, promote it:

```
import warnings

warnings.simplefilter("error", pd.errors.ChainedAssignmentError)
```

Now the script stops instead of reporting a figure computed from an edit that never landed.

5.3.2 The correct form, and what changed for existing code

```
# One .loc, both axes, one operation.
muac.loc[muac["commune"] == "Gonaives", "muac_mm"] = pd.NA
```

If you are maintaining code written for pandas 1 or 2, the migration risk runs in one direction only: a chained assignment that *used* to modify the frame now does not. The number changes and nothing crashes. Grep for `]["` and for `.copy()` calls added to silence the

old warning — the copies are now unnecessary, and each one you remove is a real saving on a large frame.

5.3.3 `.copy()` and when you still want it

Under Copy-on-Write, `df[mask]` is already independent, so a defensive `.copy()` buys nothing. It is still worth writing when it documents intent:

```
# This subset is going to be modified and is not a view of anything.
gonaives = muac.loc[muac["commune"] == "Gonaives"].copy()
gonaives["flag"] = gonaives["muac_mm"] < 115
```

Adding a column to a selection you did not copy works and is the common case; the `.copy()` above is a note to the reader, not a requirement.

5.4 Adding and changing columns

`assign` returns a new frame and chains, which keeps a pipeline readable:

```
summary = (
    muac
    .loc[muac["muac_mm"].notna()]
    .assign(
        gam=lambda d: d["muac_mm"] < 125,
        sam=lambda d: d["muac_mm"] < 115,
    )
)
```

The `lambda d:` matters: it refers to the frame *at that point in the chain*, so `sam` can be defined against columns `assign` created a line earlier. Referring to `muac` directly inside the chain would use the pre-filter frame and misalign.

For conditions with more than two branches, `np.select` beats nested `where`:

```
import numpy as np

muac["band"] = np.select(
    [muac["muac_mm"] < 115, muac["muac_mm"] < 125],
    ["severe", "moderate"],
    default="normal",
)
```

Conditions are evaluated in order and the first match wins, so `< 115` must come first. Note that `default` also catches missing values, which is almost never what you want — set them back explicitly:

```
muac.loc[muac["muac_mm"].isna(), "band"] = pd.NA
```

5.5 Sorting, and why it is not a ranking

```
muac.sort_values(["commune", "screening_date"], ascending=[True, False])
```

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 and the sort order is not information.

`nlargest` is the readable form of "top N" and is faster than sorting the whole frame:

```
muac.groupby("commune")["muac_mm"].mean().nsmallest(5)
```

5.6 What comes next

You can select a subset and change it without wondering whether the change landed. The next lesson aggregates: `groupby`, named aggregation, and getting a numerator and a denominator out of the same operation so they cannot drift apart.

CHAPTER 6

Grouping to a numerator and a denominator

Lesson 6 of 8 · 90 min

groupby, named aggregation, and the two defaults that silently change a denominator — dropped missing groups and unobserved categories. Producing both halves of an indicator in one operation so they cannot drift.

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 for your code: **compute both in the same operation.**

Two separate calculations drift. One gets a filter the other does not, someone edits one line, and the ratio becomes a number nobody can reconstruct.

```
import pandas as pd

muac = read_register(RAW / "muac-screening-artibonite-2024.v1.csv")

measured = muac["muac_mm"].notna()
gam = measured & (muac["muac_mm"] < 125)

by_commune = (
    muac.assign(measured=measured, gam=gam)
    .groupby("commune")
    .agg(
        screened=("child_id", "size"),
        measured=("measured", "sum"),
        gam_cases=("gam", "sum"),
    )
)
by_commune["gam_rate"] = (by_commune["gam_cases"] / by_commune["measured"]).round(3)
```

```
             screened  measured  gam_cases  gam_rate
commune
Gros-Morne        361       351         48     0.137
Anse-Rouge        229       223         30     0.135
L-Estere          189       187         18     0.096
Terre-Neuve       241       235         21     0.089
```

Three columns and every one of them is load-bearing. `screened` is how many children came; `measured` is how many produced a usable MUAC and is the denominator; `gam_cases` is the numerator. Publishing the rate without the denominator beside it is how the argument starts.

Note that `screened` and `measured` differ — 4,218 children were screened and 4,146 have a MUAC. Dividing by the wrong one shifts the district rate by enough to matter and by too little to notice.

6.2 Named aggregation

The form used above is the one to standardise on:

```
.agg(
    output_name=("input_column", "function"),
    ...
)
```

It names the output column at the point the aggregation is defined, so there is no second step renaming `muac_mm_mean` to something a reader understands, and no multi-level column index to flatten.

The older forms still work and are worth recognising in someone else's code:

```
muac.groupby("commune")["muac_mm"].mean()              # one column, one function
muac.groupby("commune").agg({"muac_mm": ["mean", "std"]})  # MultiIndex columns
```

The second produces columns like `("muac_mm", "mean")`, which then need flattening. Named aggregation avoids the problem rather than solving it.

Functions can be your own:

```
def share_below(series, threshold=125):
    valid = series.dropna()
    return (valid < threshold).mean() if len(valid) else float("nan")

muac.groupby("commune").agg(
    gam_rate=("muac_mm", share_below),
    n=("muac_mm", "count"),
)
```

`count` excludes missing; `size` includes them. That single distinction is the difference between the two denominators above, and it is worth saying out loud every time you use one.

6.3 Two defaults that change a denominator

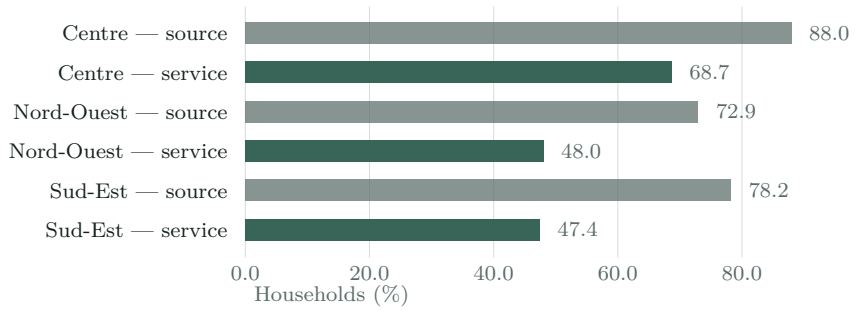

Figure 6.1: Improved water source against basic service, by district. Counting sources overstates coverage everywhere; the gap is households whose source is improved and more than thirty minutes away.

Both bars in each pair come from the same `groupby`. The only difference is which rows the numerator counts — and it moves the answer by twenty to thirty points in every district. That is what "the denominator is the point" means in practice.

6.3.1 Missing group keys are dropped

```
d = pd.DataFrame({"g": ["a", None, "a"], "v": [1, 2, 3]})

d.groupby("g")["v"].sum()                  # {'a': 4}
d.groupby("g", dropna=False)["v"].sum()    # {'a': 4, nan: 2}
```

By default `groupby` discards rows whose group key is missing. They vanish from the table **and** from the total, so the parts no longer sum to the whole and nothing says why.

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. Use `dropna=False` and decide deliberately, or assert the key is complete:

```
assert muac["commune"].notna().all(), "rows with no commune would be dropped"
```

6.3.2 Unobserved categories reappear as empty rows

If the group key is categorical, the default is now to return only observed categories:

```
c = pd.DataFrame({"k": pd.Categorical(["x", "y"], categories=["x", "y", "z"]),
                  "v": [1, 2]})

len(c.groupby("k", observed=True)["v"].sum())    # 2
len(c.groupby("k", observed=False)["v"].sum())   # 3 -- includes an empty "z"
```

Both are right for different questions. Reporting on facilities that submitted data wants `observed=True`; reporting coverage against a list of facilities that *should* have submitted wants `observed=False`, because a facility with zero rows is the finding.

Pass it explicitly. The default changed between pandas versions, and a script that relies on it produces a different table on a colleague's machine.

6.4 Grouping by more than one key

```
by_commune_month = (
    muac.assign(month=muac["screening_date"].dt.to_period("M"), gam=gam)
    .groupby(["commune", "month"], observed=True)
    .agg(measured=("muac_mm", "count"), gam_cases=("gam", "sum"))
)
```

The result has a MultiIndex. Two ways to work with it:

```
by_commune_month.loc["Gonaives"]                    # one commune, all months
by_commune_month.reset_index()                      # back to flat columns
```

`reset_index()` before writing to CSV, always — otherwise the index columns either disappear or arrive unnamed.

To turn the long result into the wide table a report wants:

```
wide = by_commune_month["gam_cases"].unstack("month", fill_value=0)
```

`fill_value=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.

6.5 transform: a group statistic on every row

`agg` collapses; `transform` returns a value per original row. This is how you compare a row against its own group without a join:

```
muac["commune_mean"] = muac.groupby("commune")["muac_mm"].transform("mean")
muac["vs_commune"] = muac["muac_mm"] - muac["commune_mean"]
```

Useful for flagging a site whose measurements sit systematically away from the rest — which is exactly the check the *SMART survey analysis* project uses to find a team measuring 0.5 kg light.

6.6 apply, and why to avoid it

`apply` runs a Python function per group and is the slowest thing in pandas by a wide margin. It is also the most flexible, so it is what people reach for first.

```
# Works, but slow, and returns something whose shape depends on the function.
muac.groupby("commune").apply(lambda g: g["muac_mm"].mean(), include_groups=False)
```

Prefer a named aggregation. Reach for `apply` when the operation genuinely needs the whole group frame — a per-group regression, a per-group ranking with ties broken on a second column — and pass `include_groups=False`, which is now required to avoid the grouping columns being handed to your function.

6.7 Pivot tables read better for a report

`pivot_table` is `groupby` plus `unstack` with a friendlier signature:

```
pd.crosstab(muac["commune"], muac["outcome"])

pd.crosstab(
    muac["commune"], muac["outcome"],
    values=muac["muac_mm"], aggfunc="mean",
).round(1)
```

`crosstab` with `normalize="index"` gives row proportions, which is usually what a referral table should show. Keep the counts beside them: a 50% referral rate on four children is a different fact from 50% on four hundred, and only the count distinguishes them.

6.8 Writing the result out

```
OUTPUTS = PROJECT / "outputs" / "tables"
OUTPUTS.mkdir(parents=True, exist_ok=True)

by_commune.reset_index().to_csv(OUTPUTS / "gam_by_commune.csv", index=False)
```

`index=False` after `reset_index()` — otherwise you get an unnamed integer column that someone will later open in Excel and sort by.

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

```
definitions = pd.DataFrame([
    ("GAM", "MUAC < 125 mm or bilateral pitting oedema",
     "children with a MUAC measurement or a recorded oedema assessment"),
], columns=["indicator", "numerator", "denominator"])

definitions.to_csv(OUTPUTS / "definitions.csv", index=False)
```

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

6.9 What comes next

The table aggregates correctly and both halves of every rate come from one operation. The next unit gets the last piece right — dates, ages and reporting periods, where an age in months decides which growth standard applies — and then wraps the whole thing in a script that runs on someone else's machine.

Part IV

Producing and handing over

CHAPTER 7

Dates, ages and reporting periods

Lesson 7 of 8 · 80 min

Age in completed months computed correctly rather than by dividing days, reporting periods that survive a partial month, and the boundary arithmetic a growth standard depends on.

7.1 Why this gets its own lesson

Date arithmetic looks like the easy part and produces some of the most expensive errors in this sector. An age in months decides which growth standard a child is compared against. A reporting period decides whether a facility counts as having reported. A month boundary decides whether a distribution falls in this quarter's figures or next.

None of those fail loudly.

7.2 Parsing, once, explicitly

```
import pandas as pd

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

Two habits worth forming:

- **State the format.** An inferred format can change between files as the mix of values changes — the same column parsed as day-first in one export and month-first in the next.
- **Use `errors="raise"`.** The alternative, `errors="coerce"`, turns every unparseable date into missing, so a file in the wrong date format imports "successfully" with an entirely empty date column.

`01/02/2024` is either 1 February or 2 January and the file will not tell you which. If an export gives you ambiguous dates, `dayfirst=True` is a decision you document, not a default you accept.

7.3 Age in completed months

The obvious approach divides days by an average month:

```
# Wrong often enough to matter.
age_months = ((visit_date - date_of_birth).dt.days / 30.4375).astype(int)
```

Over the first two thousand days of life this disagrees with the correct answer on 54 of them — always by one month, always at a boundary. A child at 60 days is reported as 1 month old when they are 2.

Compute completed months directly:

```
def age_in_months(dob: pd.Series, visit: pd.Series) -> pd.Series:
    months = (visit.dt.year - dob.dt.year) * 12 + (visit.dt.month - dob.dt.month)
    # Not yet reached the day-of-month, so the last month is not complete.
    return months - (visit.dt.day < dob.dt.day).astype(int)


dob   = pd.to_datetime(["2022-03-15", "2022-03-16", "2022-02-28"])
visit = pd.to_datetime(["2024-03-15", "2024-03-15", "2024-03-01"])

age_in_months(pd.Series(dob), pd.Series(visit)).tolist()   # [24, 23, 24]
```

The second child is one day short of two years and is 23 months, not 24. That single day is the whole reason this function exists.

7.3.1 Why one month matters here

WHO growth standards switch measurement at exactly 24 months. Below 24 months a child is measured lying down and compared against the weight-for-length curve; at 24 months and above, standing, against weight-for-height. The two measurements differ by about 0.7 cm and the two curves are different tables.

A child pushed from 23 to 24 months by a rounding rule is compared against the wrong standard, and their z-score moves. Do that to a few hundred children near the boundary and the prevalence estimate moves with them — which is why the *SMART survey analysis* project reports age heaping on whole years as a plausibility finding rather than a curiosity.

Age bands are the other place. `0-5`, `6-23`, `24-59` months are the standard nutrition bands, and a child at exactly 24 months belongs to the third:

```
BANDS = [0, 6, 24, 60]
LABELS = ["0-5m", "6-23m", "24-59m"]

muac["age_band"] = pd.cut(
    muac["age_months"], bins=BANDS, labels=LABELS, right=False, include_lowest=True
)
```

`right=False` makes each interval `[low, high)` — closed on the left, open on the right — so 24 months falls in `24-59m` and not in `6-23m`. The default is the other way round, and it is wrong for every age band this sector uses.

Check the boundaries rather than trusting them:

```
check = muac.loc[muac["age_months"].isin([5, 6, 23, 24, 59, 60]),
                 ["age_months", "age_band"]].drop_duplicates().sort_values("age_months")
print(check)
```

7.4 Reporting periods

A period is a month, quarter or year that data is reported *for*, and it is not the same as a date.

```
muac["month"] = muac["screening_date"].dt.to_period("M")

by_month = muac.groupby("month").size()
```

```
2024-01    207
```

```
2024-02    391
...
2024-11    375
2024-12    156
```

Periods sort correctly, print readably, and — unlike a string like `"2024-01"` — support arithmetic:

```
current = pd.Period("2024-06", freq="M")
print(current - 1)                # 2024-05
print(current.start_time, current.end_time)
```

Strings sort correctly too if you use ISO order, which is a good reason to use `%Y-%m` and never `%m/%Y`. But a string cannot answer "the previous month", and that question comes up in every trend calculation.

7.4.1 Partial periods are the trap

This register runs from 15 January to 13 December. January carries 207 screenings and December 156, against a monthly average near 375. Neither month is a decline; both are partial.

```
first, last = muac["screening_date"].min(), muac["screening_date"].max()
print(first.date(), last.date())

muac["partial_period"] = muac["month"].isin(
    [first.to_period("M"), last.to_period("M")]
)
```

Label them rather than dropping them. A reader who sees December fall off a chart concludes the programme collapsed; a reader who sees no December at all wonders where the data went. Naming the month partial is the only option that answers both.

The same logic applies to a period that has not finished. A month-to-date figure compared against completed months is the commonest way a dashboard shows a fictional decline every month, and it happens because both are called "the monthly total".

7.4.2 Aligning two sources on periods

The vaccination extract records a period start date rather than a date of service:

```
epi = pd.read_csv(path, parse_dates=["period"])
epi["month"] = epi["period"].dt.to_period("M")
print(epi["month"].nunique())      # 12
```

Converting both sources to `Period` before joining is what makes them line up. Joining a date to a period-start silently produces no matches for every month where one source used the last day and the other the first.

7.5 Resampling a time series

When the index is a datetime, `resample` aggregates by period without a manual grouping column:

```
daily = muac.set_index("screening_date")
weekly = daily.resample("W")["child_id"].size()
```

```
monthly = daily.resample("MS")["child_id"].size()      # MS = month start
```

`resample` fills gaps: a week with no screening appears as zero rather than being absent. That is right for a time series and wrong for a denominator — a school closed for a fortnight has no attendance days, not zero attendance, and the *School attendance* project shows what filling those in costs.

7.6 Durations

Subtracting two datetimes gives a `Timedelta`:

```
referrals["days_to_service"] = (
    referrals["service_date"] - referrals["referral_date"]
).dt.days
```

`.dt.days` truncates toward zero, so a 23-hour gap is 0 days. If same-day matters — and for a protection referral it does — say so explicitly rather than letting the truncation decide:

```
hours = (referrals["service_date"] - referrals["referral_date"]) / pd.Timedelta(hours=1)
referrals["same_day"] = hours < 24
```

A negative duration is a data-quality finding, not a value to take an absolute value of:

```
impossible = referrals["days_to_service"] < 0
print(f"service recorded before referral: {impossible.sum()}")
```

7.7 Time zones, briefly

Programme data is usually naive local dates and should stay that way. Attaching a time zone to a screening date invites a conversion that moves a record across a midnight boundary into the wrong reporting month.

Where timestamps genuinely carry a zone — server-side form submission times from CommCare or Kobo — convert once, at the boundary, and store the local date:

```
submissions["submitted_at"] = pd.to_datetime(
    submissions["submitted_at"], utc=True
).dt.tz_convert("America/Port-au-Prince")

submissions["submission_date"] = submissions["submitted_at"].dt.date
```

The submission timestamp and the date of the visit it records are different things, and reporting on the first when you mean the second shifts every evening submission into the following day.

7.8 What comes next

Every column now means what it says, including the ones made of dates. The last lesson turns the analysis into a script another officer can run next quarter on a different export, without editing anything but its arguments.

CHAPTER 8

A script another officer can run

Lesson 8 of 8 · 90 min

Arguments instead of edits, logging instead of print, failing loudly instead of producing a wrong number — turning the analysis into something that runs next quarter without you in the room.

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 notebook accumulates: a path written for one machine, a value changed by hand between runs, and a step that only works if you already know which cell to run first.

8.2 Arguments, not edits

```python
# scripts/indicator_table.py
import argparse
from pathlib import Path

def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(
        description="Compute GAM and SAM rates by commune from a screening register."
    )
    parser.add_argument(
        "--input", type=Path, required=True,
        help="Path to the screening register CSV.",
    )
    parser.add_argument(
        "--output", type=Path, required=True,
        help="Directory to write tables into. Created if missing.",
    )
    parser.add_argument(
        "--gam-threshold", type=int, default=125,
        help="MUAC cut-off in mm for global acute malnutrition (default: 125).",
    )
    parser.add_argument(
        "--min-assessment-rate", type=float, default=0.8,
        help="Communes below this assessment rate are flagged, not dropped.",
    )
    return parser.parse_args()
```

```
uv run python scripts/indicator_table.py \
    --input data/raw/muac-screening-artibonite-2024.v1.csv \
    --output outputs/tables
```

`type=Path` converts the string for you, and `required=True` means a forgotten argument produces a clear message rather than a `KeyError` two hundred lines in.

Put the thresholds in arguments and the defaults in the code. A threshold that is only ever changed by editing a line will eventually be changed and not changed back. One that is an argument with a documented default is visible in the command that produced the output, which is what you want when someone asks how last quarter's figure was computed.

`-help` is generated from the same declarations, and it is the documentation people actually read:

```
uv run python scripts/indicator_table.py --help
```

8.3 Fail loudly, early

A script that produces a wrong number silently is worse than one that crashes. Check what you assume, at the point you assume it.

```
def read_register(path: Path) -> pd.DataFrame:
    if not path.exists():
        raise SystemExit(f"Input file not found: {path}")

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

    expected = {"child_id", "commune", "screening_date", "muac_mm", "oedema"}
    missing = expected - set(muac.columns)
    if missing:
        raise SystemExit(f"Input is missing columns: {sorted(missing)}")

    if muac["child_id"].isna().any():
        raise SystemExit("Some rows have no child_id; refusing to continue.")

    return muac
```

`SystemExit` with a message exits with a non-zero status and prints one readable line, rather than a traceback the reader has to interpret. Use it for "this input is not what I was promised"; use `assert` for "this cannot happen unless the code is wrong".

Turn the pandas warning that means a silent failure into an error:

```
import warnings
import pandas as pd

warnings.simplefilter("error", pd.errors.ChainedAssignmentError)
```

Chained assignment never modifies the frame, and by default it only warns. In a script whose output someone will report, stopping is the right response.

8.4 Logging, not print

`print` goes to standard output alongside the results, has no severity, and cannot be turned down.

```
import logging

logging.basicConfig(
```

```
    level=logging.INFO,
    format="%(asctime)s %(levelname)-8s %(message)s",
    datefmt="%H:%M:%S",
)
log = logging.getLogger(__name__)

log.info("Read %d rows from %s", len(muac), args.input)
log.warning("%d communes below the assessment-rate floor", len(flagged))
log.info("Wrote %s", output_path)
```

```
09:14:02 INFO     Read 4218 rows from data/raw/muac-...v1.csv
09:14:02 WARNING  2 communes below the assessment-rate floor
09:14:02 INFO     Wrote outputs/tables/gam_by_commune.csv
```

Three things this buys that `print` does not: a timestamp, a severity someone can filter on, and the ability to add `-verbose` without touching every call.

Use the `%s` form rather than an f-string — the formatting is skipped entirely when the level is disabled.

Log the numbers that would let someone reconstruct the run: rows read, rows excluded and why, the denominator, the output path. A log that says "done" tells nobody anything.

8.5 The shape of the script

```
# scripts/indicator_table.py
"""Compute GAM and SAM rates by commune from a MUAC screening register."""

import argparse
import logging
from pathlib import Path

import pandas as pd

log = logging.getLogger(__name__)

GAM_DEFAULT, SAM_DEFAULT = 125, 115

def read_register(path: Path) -> pd.DataFrame:
    ...

def indicator_table(muac: pd.DataFrame, gam_mm: int, sam_mm: int) -> pd.DataFrame:
    measured = muac["muac_mm"].notna() | muac["oedema"].notna()
    gam = measured & ((muac["muac_mm"] < gam_mm) | (muac["oedema"] == True))
    sam = measured & ((muac["muac_mm"] < sam_mm) | (muac["oedema"] == True))

    table = (
        muac.assign(measured=measured, gam=gam, sam=sam)
        .groupby("commune", dropna=False, observed=True)
        .agg(
            screened=("child_id", "size"),
            assessed=("measured", "sum"),
            gam_cases=("gam", "sum"),
            sam_cases=("sam", "sum"),
        )
    )
    table["assessment_rate"] = (table["assessed"] / table["screened"]).round(3)
    table["gam_rate"] = (table["gam_cases"] / table["assessed"]).round(3)
    table["sam_rate"] = (table["sam_cases"] / table["assessed"]).round(3)
```

```
    return table.reset_index()

def main() -> None:
    args = parse_args()
    logging.basicConfig(level=logging.INFO, format="%(levelname)-8s %(message)s")

    muac = read_register(args.input)
    log.info("Read %d rows", len(muac))

    table = indicator_table(muac, args.gam_threshold, SAM_DEFAULT)

    thin = table.loc[table["assessment_rate"] < args.min_assessment_rate, "commune"]
    if len(thin):
        log.warning("Assessment rate below floor: %s", ", ".join(thin))

    args.output.mkdir(parents=True, exist_ok=True)
    destination = args.output / "gam_by_commune.csv"
    table.to_csv(destination, index=False)
    log.info("Wrote %s (%d communes)", destination, len(table))

if __name__ == "__main__":
    main()
```

Four things about that structure are deliberate.

Functions take arguments and return values. `indicator_table` does not read a file, does not know where output goes, and does not touch a global. That is what makes it testable and what lets the notebook import it.

`main()` is the only place that does input and output. Reading, writing and logging live there; everything else is computation.

The `if __name__ == "__main__"` guard means importing this module from a notebook runs nothing. Without it, `from indicator_table import indicator_table` executes the whole script.

Thin data is flagged, not dropped. A commune below the assessment-rate floor stays in the table with a warning beside it. Dropping it would change the district denominator and nothing in the output would say so.

8.6 Testing the part that computes

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

```
# tests/test_indicator_table.py
import pandas as pd
from indicator_table import indicator_table

def test_denominator_excludes_unmeasured_children():
    muac = pd.DataFrame({
        "child_id": ["a", "b", "c"],
        "commune": ["X", "X", "X"],
        "muac_mm": [110, 130, None],
        "oedema": [False, False, None],
    })

    table = indicator_table(muac, gam_mm=125, sam_mm=115)

    assert table.loc[0, "screened"] == 3
    assert table.loc[0, "assessed"] == 2      # the unmeasured child is not a denominator
    assert table.loc[0, "gam_rate"] == 0.5
```

That test encodes the decision from lesson 6 — which children are in the denominator — in a form that fails if someone changes it. It is worth writing for every indicator whose definition you had to argue about.

8.7 Making the run reproducible

Record what produced the output, beside the output:

```
import json
import subprocess
from datetime import datetime, timezone

def run_metadata(args) -> dict:
    return {
        "generated_at": datetime.now(timezone.utc).isoformat(timespec="seconds"),
        "input": str(args.input),
        "gam_threshold": args.gam_threshold,
        "pandas": pd.__version__,
        "commit": subprocess.run(
            ["git", "rev-parse", "--short", "HEAD"],
            capture_output=True, text=True,
        ).stdout.strip() or "not a git repository",
    }

(args.output / "run.json").write_text(json.dumps(run_metadata(args), indent=2))
```

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

Then the check from lesson 2, which is the real test:

```
rm -rf outputs/
uv run python scripts/indicator_table.py --input data/raw/muac.csv --output outputs/tables
git status
```

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

8.8 Handing it over

The README from lesson 2 needs one more line now — the command with real arguments:

```
## Run

    uv sync
    uv run python scripts/indicator_table.py \
        --input data/raw/muac-screening-artibonite-2024.v1.csv \
        --output outputs/tables

Thresholds default to MUAC < 125 mm (GAM) and < 115 mm (SAM).
Pass --gam-threshold to change it; the value used is recorded in
outputs/tables/run.json.
```

Two commands, and the thresholds visible without opening the code. That is the whole handover.

8.9 Where this course ends

You can build an environment that reinstalls offline, read any export without corrupting it, turn its codes into values, aggregate to a numerator and a denominator that came from the same operation, get the date arithmetic right, 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 in **Sector Analysis** on the programme roadmap take it further into the classifications your cluster holds you to.

If your team works in R rather than Python, *R for Programme Data* covers the same ground in that language.

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/python-for-programme-data

Generated July 27, 2026.