CASSION · FORMATION

Python pour les données de programme

Le Python dont un chargé de suivi-évaluation a réellement besoin — environnements, pandas, dates, codes catégoriels et formats de fichiers — enseigné sur des exports d'enquêtes et de registres plutôt que sur des données factices.

Formateur	Pierre Robentz Cassion
Niveau	Débutant
Heures apprenant	20 h
Enseignée en	Python
Mise à jour	27 juillet 2026
Secteurs	Santé publique · Nutrition
Normes et méthodologies	Gestion axée sur les résultats (GAR) · Normes OMS de croissance de l'enfant

Consulter et exécuter la formation en ligne sur data-analysis.cassion.dev/fr/cours/python-for-programme-data

Ce que vous saurez faire

— Installer et reproduire un environnement Python fonctionnel sur une machine sans connexion fiable, et expliquer pourquoi un Python système est le mauvais endroit où travailler

— Lire un export CSV, Excel ou à largeur fixe sans perdre un zéro initial, sans se tromper de date, et sans laisser une valeur sentinelle entrer dans une moyenne

— Sélectionner, filtrer et regrouper un DataFrame pandas délibérément, et expliquer ce que dit le SettingWithCopyWarning plutôt que de le masquer

— Calculer correctement âges, périodes de rapportage et désagrégations par tranche d'âge, y compris l'arithmétique en mois dont dépend un standard de croissance

— Écrire un script qu'un autre chargé pourra exécuter au trimestre suivant sur un autre export, sans rien modifier d'autre que ses arguments

Place de cette formation dans le programme

Fondamentaux — Passer d'un export brut de programme à un tableau exploitable, dans celui des deux langages — Python ou R — que votre équipe utilise déjà.

Prérequis

Aucun. Cette formation ne suppose aucune expérience préalable de la programmation.

Sommaire

À propos de cette formation

Ceci est un cours de Python, non un cours d'analyse. Il suppose que vous savez déjà ce qu'est un indicateur et pourquoi un dénominateur compte — c'est ce qu'enseigne *Fondamentaux de l'analyse de données* — et consacre son temps au langage lui-même : ce que pandas fait réellement, lesquels de ses réglages par défaut vous nuiront, et comment écrire quelque chose qui s'exécute encore quand vous n'êtes plus là.

Python uniquement. Son pendant à la feuille de route, *R pour les données de programme*, fait le même travail en R. Ailleurs sur cette plateforme, chaque technique est montrée dans les deux langages, car une équipe est le plus souvent partagée entre les deux ; ici la séparation est justement le principe, et couvrir les deux dans un seul cours diviserait par deux la profondeur de chacun.

Tout s'exécute sur de vrais jeux de données de la plateforme — le registre de dépistage par PB de douze communes de l'Artibonite, un extrait de vaccination au format DHIS2, un fichier de présence scolaire — avec les zéros initiaux, les codes sentinelles, les dates incohérentes et les inscriptions dupliquées que ces fichiers portent réellement. Aucun DataFrame factice de trois lignes : les erreurs que ce cours existe pour prévenir n'apparaissent pas sur trois lignes.

Vous n'aurez pas besoin d'internet dans la salle. La leçon 1 sert précisément à vous en assurer.

Première partie

Un environnement qui tiendra

CHAPITRE 1

Un Python réinstallable sans internet

Leçon 1 sur 8 · 80 min

Pourquoi le Python système est le mauvais endroit où travailler, ce qu'est réellement un environnement virtuel, et comment transporter une installation fonctionnelle vers un portable de terrain qui n'a jamais vu d'index de paquets.

1.1 L'échec que cette leçon prévient

Une analyse s'exécute sur votre portable. Trois mois plus tard, le même script est lancé par quelqu'un d'autre, sur une autre machine, et soit il ne démarre pas, soit — bien pire — il produit un chiffre différent.

Ce n'est généralement pas un défaut de l'analyse. C'est un défaut de l'environnement : une version de pandas différente dont un réglage par défaut a changé, un paquet installé pour un projet qui en a cassé un autre, un Python système que le système d'exploitation a mis à jour sous vos pieds.

Cette leçon vise à faire de l'environnement une chose que vous déclarez, plutôt qu'une chose qui s'accumule.

1.2 Ne travaillez pas dans le Python système

macOS et la plupart des distributions Linux livrent un Python que le système d'exploitation utilise pour son propre outillage. Sur un portable de terrain partagé, quelqu'un en a généralement installé un deuxième depuis un site web, et un troisième est arrivé avec une suite bureautique.

Installer des paquets dans l'un d'eux est une mauvaise idée pour une raison précise : le système d'exploitation dépend des versions qu'il a livrées. Mettre pandas à jour pour satisfaire votre script peut casser un utilitaire système, et rien ne vous en avertira.

Les versions récentes de Python refusent désormais purement et simplement :

```
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.
```

Cette erreur est justifiée et ne doit pas être contournée. Il existe une option pour passer outre. L'utiliser, c'est ainsi qu'un portable finit par devoir être réinstallé entièrement.

1.3 Ce qu'est un environnement virtuel

Un environnement virtuel est un répertoire contenant sa propre copie du répertoire de paquets de l'interpréteur. L'activer change l'endroit où `python` et `pip` vont chercher. Toute l'idée est là.

```
python3 -m venv .venv
```

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

Deux conséquences à intérioriser :

— **Par projet, non par machine.** Deux analyses exigeant des versions de pandas différentes coexistent sans que l'une sache que l'autre existe.

— **Jetable.** Si un environnement se retrouve dans un mauvais état, supprimez le répertoire et reconstruisez-le. Rien de précieux n'y réside — ce qui n'est vrai que si vos dépendances sont déclarées ailleurs.

1.4 uv, et pourquoi il compte davantage ici qu'ailleurs

`pip` résout et télécharge les paquets un par un depuis le réseau. Sur une connexion qui tombe, une installation à moitié terminée laisse un environnement qui n'est ni l'ancien ni le nouveau.

`uv` est un résolveur plus rapide qui produit un **fichier de verrouillage** : la liste exacte de chaque paquet et de chaque version, y compris ceux dont vos paquets dépendent.

```
# Installer uv lui-meme, une fois, sur une machine effectivement connectee.
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` écrit deux fichiers. `pyproject.toml` consigne ce que vous avez demandé — `pandas` — et `uv.lock` consigne ce que vous avez obtenu, jusqu'à la version exacte de chaque dépendance transitive. **Versionnez les deux.** Le premier est votre intention ; le second est ce qui rend l'installation du trimestre prochain identique à celle-ci.

Sur le portable de terrain :

```
uv sync                         # lit uv.lock, installe exactement ces versions
```

conda est l'alternative raisonnable et reste courante dans ce secteur, en particulier lorsqu'une pile géospatiale est en jeu — GDAL s'installe bien plus facilement par conda que par pip. Si votre organisation a déjà standardisé sur conda, utilisez conda ; le principe est identique et seules les commandes changent.

1.5 Installer là où il n'y a pas d'internet

C'est la partie que la plupart des tutoriels sautent, et c'est celle qui compte sur un déploiement.

Téléchargez les paquets une fois, sur une machine connectée, en ciblant la plateforme du portable de terrain :

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

Copiez le répertoire `wheels` sur une clé USB avec votre projet, puis sur la machine hors ligne :

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

`-no-index` indique à l'installateur de ne pas contacter du tout l'index de paquets, et `-find-links` le dirige vers le répertoire à la place. L'installation est alors entièrement locale.

> Les wheels sont spécifiques à une plateforme. Une wheel téléchargée sous macOS ne s'installera pas sur un portable Windows, et une wheel construite pour Python 3.12 ne s'installera pas dans 3.11. Téléchargez sur une machine correspondant à la cible, ou passez explicitement `-python-platform` et `-python-version`.

1.6 Épingler aussi la version de Python

Les paquets sont épinglés ; l'interpréteur devrait l'être également. Un script écrit pour 3.12 et exécuté sous 3.9 échoue sur une syntaxe qui n'existait pas encore.

```
# 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 Vérifier l'environnement avant de lui faire confiance

Avant d'exécuter une analyse sur une machine que vous n'avez pas configurée, vérifiez que vous êtes bien là où vous le croyez :

```
import sys
import pandas as pd

print(sys.executable)          # doit se trouver dans .venv, non dans /usr/bin
print(sys.version)
print(pd.__version__)
```

Si `sys.executable` vaut `/usr/bin/python3`, l'environnement n'est pas activé et vous êtes sur le point d'installer dans le Python système. Cette seule ligne a sauvé plus de portables que n'importe quelle documentation.

Pour une analyse dont les chiffres comptent, transformez-la en assertion :

```
import sys

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

1.8 Ce qu'il faut remettre à un collègue

Tout ce qui permet de reconstruire l'environnement, et rien de ce qui a été construit :

À versionner	À ne pas versionner
`pyproject.toml`	`.venv/`
`uv.lock`	`wheels/`
`README.md` avec les deux commandes	`__pycache__/`

Un `.gitignore` couvrant la colonne de droite fait partie du gabarit de projet de la leçon suivante.

Le README a besoin de deux lignes, non de deux pages :

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

1.9 Ce qui vient ensuite

L'environnement est reproductible. La leçon suivante porte sur le projet qui l'entoure — où vont les données brutes, où vont les sorties, et pourquoi un chemin écrit `C:\Users\marie\Bureau\data.csv` garantit que le script s'exécutera sur exactement une machine.

CHAPITRE 2

Un projet qu'une autre personne peut ouvrir

Leçon 2 sur 8 · 70 min

Où vivent les données brutes, le code et les sorties, pourquoi un chemin en dur garantit que le script ne tourne que sur une machine, et la règle qui rend une analyse rejouable — ne jamais écrire dans le dossier que l'on lit.

2.1 La forme d'un projet

Presque toute analyse de ce secteur comporte les quatre mêmes sortes de fichiers, et donner à chacune un emplacement fixe supprime une catégorie de questions que personne ne devrait avoir à poser.

```
muac-analysis/
  data/
    raw/            l'export tel qu'il est arrive, jamais modifie
    interim/        fichiers intermediaires, supprimables sans risque
  outputs/
    tables/
    figures/
  scripts/
    read_register.py
    indicator_table.py
  notebooks/
    exploration.ipynb
  pyproject.toml
  uv.lock
  README.md
  .gitignore
```

La ligne importante est la première.

2.2 data/raw est en lecture seule

L'export tel qu'il est arrivé est la seule chose du projet que vous ne pouvez pas reproduire. Tout le reste — le tableau nettoyé, les figures, les tableaux d'indicateurs — peut être régénéré en réexécutant le code. Le fichier brut, non : le serveur d'où il vient aura changé.

Il n'est donc jamais modifié, jamais écrasé, jamais trié dans Excel « juste pour regarder ». Si une correction arrive, elle se pose à côté de l'original sous un nouveau nom plutôt que de le remplacer.

Ce nommage mérite d'être délibéré :

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

et non

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

Une analyse exécutée sur la v1 doit continuer à se reproduire sur la v1. Si le nom de fichier est écrasé, le chiffre du trimestre dernier ne pourra plus jamais être expliqué — et expliquer le chiffre du trimestre dernier fait partie de ce qu'on vous demandera.

2.3 Des chemins qui fonctionnent sur la machine d'un autre

C'est la raison la plus fréquente pour laquelle le script d'un collègue échoue sur votre portable :

```
# S'execute sur exactement un ordinateur.
muac = pd.read_csv("C:/Users/marie/Bureau/muac-screening-artibonite-2024.v1.csv")
```

Le correctif consiste à écrire des chemins relatifs au projet, et à laisser Python déterminer où se trouve le projet.

```
from pathlib import Path
import pandas as pd

# __file__ est ce script ; son parent est scripts/, dont le parent est le projet.
PROJECT = Path(__file__).resolve().parent.parent
RAW = PROJECT / "data" / "raw"
OUTPUTS = PROJECT / "outputs"

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

Trois avantages :

— **Il s'exécute depuis n'importe où.** `python scripts/indicator_table.py` et `python /home/marie/muac-analysis/scripts/indicator_table.py` fonctionnent tous deux, car les chemins sont calculés à partir de l'emplacement du script lui-même et non du répertoire courant du terminal.

— **Il fonctionne sous Windows.** `Path` assemble avec `/` dans votre source et produit `\` sous Windows. Ne construisez jamais un chemin par concaténation de chaînes.

— **Il est repérable.** Chaque fichier touché par le script se trouve sous `RAW` ou `OUTPUTS` : un lecteur voit les entrées et les sorties d'un coup d'œil.

Dans un notebook, `__file__` n'existe pas. Définissez plutôt la racine du projet explicitement dans la première cellule, et gardez le reste du notebook relatif à elle :

```
from pathlib import Path

PROJECT = Path.cwd().parent      # le notebook vit dans notebooks/
assert (PROJECT / "data" / "raw").exists(), f"pas la racine du projet : {PROJECT}"
```

L'assertion compte plus qu'il n'y paraît. Sans elle, un notebook lancé depuis le mauvais répertoire échoue plusieurs cellules plus loin, sur une erreur déroutante concernant une colonne absente.

2.4 Les sorties sont jetables, et c'est le test

Si supprimer `outputs/` puis réexécuter les scripts ne le restaure pas à l'identique, quelque chose dans l'analyse n'est pas reproductible — une étape manuelle, une modification faite dans Excel, une cellule exécutée dans le désordre.

```
rm -rf outputs/
uv run python scripts/indicator_table.py
git status                      # ne doit rien afficher d'inattendu
```

Prenez-en l'habitude avant toute transmission. C'est la vérification la moins coûteuse qui soit et elle attrape la défaillance la plus difficile à diagnostiquer plus tard.

Créez les répertoires de sortie depuis le code plutôt que d'attendre qu'ils existent :

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

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

`parents=True` crée les répertoires intermédiaires ; `exist_ok=True` fait qu'une seconde exécution n'échoue pas. Un collègue qui clone le projet obtient les dossiers sans qu'on le lui dise.

2.5 Ce qui entre dans le contrôle de version

À versionner	À ne pas versionner
`scripts/`, `notebooks/`	`data/raw/` — voir ci-dessous
`pyproject.toml`, `uv.lock`	`data/interim/`
`README.md`	`outputs/`
`.gitignore`	`.venv/`, `__pycache__/`

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

Ne versionnez jamais de données brutes de bénéficiaires, identifiées ou pseudonymisées. Git conserve chaque version de chaque fichier pour toujours ; un fichier supprimé dans un commit ultérieur reste dans l'historique, et un dépôt interne le lundi peut être partagé le vendredi. Si les données sont synthétiques ou déjà publiques, les versionner est un confort. Sinon, le répertoire brut reste dehors et le README indique d'où vient le fichier.

Les identifiants de connexion suivent la même règle, en plus strict : un jeton DHIS2, une clé d'API KoboToolbox ou un mot de passe de base de données n'apparaissent jamais dans un script, un notebook ou un commit.

```
import os

# Lu depuis l'environnement ; la valeur n'entre jamais dans le depot.
token = os.environ["DHIS2_TOKEN"]
```

Gardez la valeur dans un fichier `.env` couvert par `.gitignore`, et documentez le nom de la variable — non sa valeur — dans le README.

2.6 Notebooks et scripts font des métiers différents

Les deux ont leur place dans un projet et ne sont pas interchangeables.

Un **notebook** sert à regarder : explorer un nouvel export, vérifier une distribution, produire une figure à coller dans un rapport. Il s'exécute dans le désordre, porte un état invisible et se compare mal en diff, ce qui en fait un mauvais endroit pour ce qui doit s'exécuter à l'identique au trimestre prochain.

Un **script** sert à produire : il s'exécute de haut en bas, prend des arguments, et réussit ou échoue. Quand une exploration devient un chiffre que quelqu'un va rapporter, déplacez-la dans un script.

Le flux de travail pratique consiste à explorer dans `notebooks/`, puis à déplacer la partie stabilisée dans `scripts/` et à la réimporter :

```
# 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"]},
    )
```

```
# Dans le 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")
```

Le notebook et le script lisent désormais le fichier de la même façon, et il n'y a qu'un seul endroit à corriger quand la sentinelle change.

2.7 Un README réellement lu

Deux commandes et trois phrases valent mieux qu'une page que personne ne termine :

```
# Tableaux d'indicateurs du depistage PB

Produit les taux de MAG et de MAS par commune a partir du registre de
depistage de l'Artibonite.

## Executer

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

## Donnees

data/raw n'est pas versionne. Telechargez le registre depuis le partage
du programme et placez-le la sous son nom de fichier versionne.
```

2.8 Ce qui vient ensuite

Le projet a une forme et les chemins survivent à un changement de machine. L'unité suivante fait entrer les données : lire un CSV, un classeur Excel et un export à largeur fixe sans perdre un zéro initial ni se tromper de date.

Deuxième partie

Faire entrer l'export

CHAPITRE 3

CSV, Excel et largeur fixe, lus sans dommage

Leçon 3 sur 8 · 90 min

Encodages, séparateurs, classeurs multi-feuilles, lignes d'en-tête fusionnées et spécifications de colonnes — les pièges propres à chaque format, qui corrompent un fichier avant même que vous n'ayez regardé une seule valeur.

3.1 Ce que cette leçon suppose

Fondamentaux de l'analyse de données couvre le principe : déclarez vos types, déclarez vos sentinelles, et ne laissez jamais un lecteur en inférer l'un ou l'autre. Cette leçon le tient pour acquis et entre dans les formats eux-mêmes, car chacun échoue différemment et les défaillances ne se voient plus dans les données ensuite.

3.2 Le CSV n'est pas un format unique

`read_csv` compte une cinquantaine de paramètres. Quatre d'entre eux expliquent l'essentiel des dégâts.

3.2.1 Le séparateur

Un fichier produit sur une machine Windows française ou espagnole est très souvent séparé par des points-virgules, car la virgule y est le séparateur décimal.

```
import pandas as pd

# Faux : tout atterrit dans une colonne unique nommee d'apres l'en-tete entier.
wrong = pd.read_csv("export.csv")
print(wrong.shape)              # (2736, 1)

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

Le symptôme est sans ambiguïté une fois connu : un DataFrame à exactement une colonne. Vérifiez `.shape` immédiatement après chaque lecture.

3.2.2 L'encodage

Les caractères accentués dans les noms de communes — Gonaïves, L'Estère, Anse-Rouge — sont l'endroit où cela mord.

```
# UnicodeDecodeError, ou pire, silencieusement abime : "Gona\xefves"
muac = pd.read_csv(path)

muac = pd.read_csv(path, encoding="utf-8")       # ce que vous voulez
muac = pd.read_csv(path, encoding="latin-1")     # ce qu'Excel produit souvent
```

`latin-1` ne lève jamais d'erreur, car tout octet est un caractère latin-1 valide. Cela en fait un mauvais diagnostic et un repli correct : si un fichier lu en `latin-1` affiche `GonaÃ¯ves`, il était en UTF-8 depuis le début et vous avez dit le contraire au lecteur.

3.2.3 Les séparateurs de milliers transforment les nombres en texte

```
# target_population arrive en "1,480" et pandas le lit comme une chaine.
epi = pd.read_csv(path, thousands=",")
```

Sans `thousands`, cette colonne est de type objet et toute opération arithmétique dessus échoue ou concatène. Vérifiez `.dtypes` après lecture, à chaque fois.

3.2.4 Les zéros initiaux

Le réglage par défaut le plus coûteux de ce secteur.

```
# facility_id "007" devient l'entier 7 ; la jointure a la liste des formations
# echoue exactement pour celles dont le code portait un zero initial.
epi = pd.read_csv(path)

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

La règle se généralise : **si vous n'allez jamais faire d'arithmétique dessus, ce n'est pas un nombre.** Codes de formations sanitaires, numéros de téléphone, identifiants de grappes, numéros de ménages et codes administratifs sont du texte qui s'écrit avec des chiffres.

Une habitude défensive pour un fichier dont vous ne connaissez pas encore les colonnes :

```
# Tout lire en texte d'abord, regarder, puis convertir deliberement.
peek = pd.read_csv(path, dtype="string", nrows=200)
print(peek.head())
print(peek.columns.tolist())
```

Deux cents lignes suffisent à voir la forme et coûtent peu sur un gros fichier.

3.3 Excel

3.3.1 Il y a plus d'une feuille

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

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

`read_excel` sans `sheet_name` renvoie la **première** feuille, qui dans un classeur de programme est le plus souvent une page de garde. Listez toujours les feuilles d'abord.

`sheet_name=None` renvoie un dictionnaire de toutes les feuilles, ce qui permet de traiter un classeur à une feuille par mois :

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

3.3.2 L'en-tête n'est pas en ligne 1

Les classeurs de programme portent couramment un titre, une ligne de logo et une ligne vide au-dessus du véritable en-tête.

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

Le symptôme, ce sont des noms de colonnes comme `Unnamed: 0`, `Unnamed: 1`. Si vous les voyez, la ligne d'en-tête est ailleurs.

3.3.3 Les cellules fusionnées produisent des valeurs manquantes

Une cellule fusionnée porte sa valeur en haut à gauche et rien dans le reste. Quand un nom de district est fusionné sur ses formations sanitaires, seule la première ligne de chaque district en porte un.

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

`ffill` est correct ici et dangereux en général : il n'est juste que parce que les blancs proviennent d'une fusion, non d'une non-réponse. Ne l'appliquez jamais à une colonne où un blanc pourrait signifier « non renseigné ».

3.3.4 Les dates Excel

Excel stocke les dates comme un nombre de jours depuis le 30/12/1899. `read_excel` les convertit généralement, mais une colonne typée en texte dans le classeur arrive sous forme de numéro de série brut.

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

Si une colonne de dates arrive en entiers à cinq chiffres, voilà pourquoi.

3.4 Fichiers à largeur fixe

Encore courants depuis les anciens systèmes d'information sanitaire et depuis les extractions ministérielles. Il n'y a pas de séparateur ; chaque champ occupe une plage fixe de caractères.

```
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()
```

Trois remarques :

— **Les plages sont semi-ouvertes**, comme les tranches Python : `(0, 7)` couvre les caractères 0 à 6. Une erreur d'un rang ici décale tous les champs suivants d'un caractère et produit un fichier qui paraît presque juste.

— **Retirez le remplissage.** Les champs à largeur fixe sont complétés par des espaces : `"Terre-Neuve "` ne sera pas égal à `"Terre-Neuve"`.

— **Prenez la spécification dans la documentation, pas en comptant à l'écran.** `read_fwf` sait inférer `colspecs`, et il les infère depuis les espaces — ce qui échoue dès qu'un champ est plein ou qu'une valeur contient une espace.

3.5 Vérifier la lecture avant d'en faire quoi que ce soit

Chaque lecture devrait être suivie des quatre mêmes contrôles. Ensemble ils prennent une minute et attrapent presque tout ce qui précède.

```
def check(df, expected_rows=None):
    print("forme     ", df.shape)
    print("types     ", df.dtypes.value_counts().to_dict())
    print("manquants ", df.isna().sum().to_dict())
    print("1re ligne ", df.iloc[0].to_dict())
    if expected_rows is not None:
        assert len(df) == expected_rows, f"attendu {expected_rows}, obtenu {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)
```

— **la forme** attrape le mauvais séparateur et un téléchargement tronqué.

— **les types** attrapent les zéros initiaux perdus, les séparateurs de milliers et une colonne numérique lue comme du texte.

— **les manquants** attrapent un mauvais encodage et une sentinelle non déclarée.

— **la première ligne** attrape un décalage d'en-tête — si elle ressemble à un en-tête, c'en était un.

L'assertion sur le nombre de lignes est celle qu'on saute. Un fichier qui arrive avec 4 190 lignes au lieu de 4 218 a perdu quelque chose entre le serveur et vous, et le script doit le dire plutôt que de rapporter discrètement une charge de cas plus faible.

3.6 Lire un gros fichier

Si un fichier ne tient pas confortablement en mémoire, lisez les colonnes nécessaires plutôt que toutes :

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

`usecols` économise beaucoup sur un export large. Au-delà, `chunksize` renvoie un itérateur de morceaux que vous agrégez au fil de l'eau :

```
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()
```

Le fichier de présence de 70 000 lignes n'en a pas besoin. Une extraction DHIS2 pluriannuelle, si.

3.7 Ce qui vient ensuite

Le fichier est en mémoire, forme et types intacts. La leçon suivante traite de ce qui est écrit *à l'intérieur* des colonnes : codes sentinelles, vocabulaires catégoriels, valeurs booléennes saisies de cinq façons différentes, et les étiquettes de valeurs qu'un fichier Stata porte et qu'un CSV jette.

CHAPITRE 4

Codes, sentinelles et le 99 qui entre dans une moyenne

Leçon 4 sur 8 · 85 min

Valeurs sentinelles, vocabulaires catégoriels, booléens saisis de cinq façons et étiquettes de valeurs Stata — transformer ce que le formulaire a enregistré en ce que pandas peut calculer.

4.1 Un nombre qui signifie « pas de réponse »

Les formulaires papier et les systèmes bâtis à leur image n'ont pas de blanc. Ils ont un code. `99` signifie « non répondu », `88` « sans objet », `-99` « non mesuré », et `999` existe parce que quelqu'un en a eu besoin d'un troisième.

Aucun n'est une valeur. Tous sont des nombres du point de vue de pandas.

```
import pandas as pd

muac = pd.read_csv(RAW / "muac-screening-artibonite-2024.v1.csv")
print(muac["muac_mm"].mean())        # plusieurs millimetres sous la verite
```

Le résultat est faux et, pire, *plausible*. Rien n'y ressemble à une erreur, et une moyenne de PB n'est pas un chiffre que la plupart des lecteurs peuvent vérifier à l'œil. Un `99` dans une colonne d'âge est encore plus discret — c'est un âge possible.

4.1.1 Déclarer les sentinelles à la lecture

```
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())
```

Restreignez la sentinelle à sa colonne. `na_values=["-99"]` sans dictionnaire l'applique à toutes les colonnes du fichier, ce qui reste juste jusqu'à ce qu'une colonne porte légitimement `-99`.

4.1.2 La liste des sentinelles est une documentation, non une tradition orale

Les sentinelles viennent du dictionnaire de codes du formulaire. Écrivez-les là où le code peut les voir :

```
SENTINELS = {
    "muac_mm": ["-99"],           # non mesure
    "age_months": ["99", "-1"],   # non repondu / sans objet
}

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

Une sentinelle que vous ignorez est invisible. Cela vaut un contrôle direct par colonne numérique avant de lui faire confiance :

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

Des valeurs agglutinées aux extrêmes — `98, 99` en haut d'une colonne d'âge, `-1` en bas — sont des sentinelles, non des observations. Un histogramme avec un pic exactement sur 99 est le même signal.

> Déclarer une sentinelle n'est pas décider quoi faire de la valeur manquante. Cette décision — supprimer, imputer, rapporter séparément — relève de l'analyse, et doit être consignée. Ici vous empêchez seulement un code de se faire passer pour une mesure.

4.2 Vocabulaires catégoriels

`outcome` dans le registre PB prend quatre valeurs et `sex` en prend deux. Déclarer le vocabulaire vous vaut une alerte dès qu'autre chose apparaît.

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

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

Comptez les manquants immédiatement après la conversion. C'est là le piège : une valeur hors des catégories déclarées devient `NaN` au lieu de lever une erreur. Un passage de zéro à neuf signifie que neuf lignes portaient une valeur que vous ignoriez, et vous voulez le voir maintenant plutôt que le découvrir comme un trou dans un tableau trois étapes plus loin.

Pour savoir lesquelles, comparez avant et après :

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

4.2.1 Catégories ordonnées

Certains vocabulaires ont un ordre, et le déclarer fait fonctionner la comparaison :

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

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

Sans `ordered=True` cette comparaison lève une erreur. Avec, l'échelle du JMP se trie et se trace dans le bon ordre plutôt qu'alphabétiquement — c'est la différence entre un graphique lisible et un graphique qui place « basic » entre « unimproved » et « limited ».

4.3 Des booléens saisis de cinq façons

La colonne `oedema` en est l'exemple type. Deux communes ont saisi `Y` et `N` au premier trimestre ; les autres `true` et `false`.

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

Ne convertissez jamais une telle colonne directement :

```
# Faux d'une maniere difficile a voir : toute chaine non vide est vraie,
# donc "false" devient True.
muac["oedema"] = muac["oedema"].astype(bool)
```

Faites une correspondance explicite, avec une liste d'autorisation :

```
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())    # ce que la correspondance n'a pas couvert
```

Une liste d'autorisation plutôt qu'une heuristique, pour la raison qui justifie une séance entière sur cette leçon : **tout ce qui en sort reste manquant et est compté**, au lieu d'être deviné. Une valeur que la table ne couvre pas est une question à poser à qui l'a saisie.

Utilisez le booléen nullable de pandas quand un véritable « non renseigné » existe :

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

`bool` ne peut pas porter de manquant ; `boolean` le peut. Dans ce registre, 44 enregistrements n'ont aucune évaluation d'œdèmes, et les ramener à `False` affirmerait silencieusement que 44 enfants ont été examinés et déclarés indemnes.

4.4 Du texte qui devrait être une seule valeur

Les colonnes proches du texte libre arrivent avec des variations de casse, d'espacement et d'orthographe. L'enquête EAH porte un district écrit de quatre façons.

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

Sans regroupement, cela éclate le district en quatre fragments dont aucun ne paraît alarmant.

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

```
print(wash["district"].nunique())     # 3
```

Normalisez **avant** le premier `groupby`, non après avoir remarqué que les totaux ne tombent pas juste. Et normalisez vers une forme canonique que vous choisissez, plutôt que de retenir l'orthographe la plus fréquente — la plus fréquente peut changer au prochain export.

Là où la variation n'est pas mécanique — `Gonaives` contre `Gonaïves`, `St-Marc` contre `Saint-Marc` — une table de correspondance est la réponse honnête :

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

Gardez cette table dans le code, non dans votre tête. C'est une décision documentée que quelqu'un devra vérifier.

4.5 Stata et SPSS transportent leurs étiquettes

Un `.dta` issu d'un institut de sondage porte à la fois le code et son sens : `1 = Oui`, `2 = Non`, `9 = Ne sait pas`.

```
survey = pd.read_stata(path)                        # valeurs converties en etiquettes
survey = pd.read_stata(path, convert_categoricals=False)   # codes bruts
```

pandas vous donne l'un ou l'autre. Pour garder les deux — ce que vous voulez, car le code est ce que documente le dictionnaire et l'étiquette est ce dont un lecteur a besoin — lisez les métadonnées séparément :

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

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

Le paquet `haven` de R préserve les étiquettes plus complètement que pandas. Si on vous remet un `.dta` d'un institut de sondage et que les étiquettes comptent, le lire sous R et écrire un CSV accompagné d'un dictionnaire est une étape légitime — et c'est le seul endroit de ce cours où la réponse est « utilisez l'autre langage ».

4.6 La fonction de lecture, assemblée

Tout ce qui précède appartient à une fonction unique que le notebook et les scripts partagent :

```
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(), "chaque ligne exige un identifiant"
    return muac
```

Une fonction, un seul endroit à corriger quand le prochain export introduira une cinquième façon d'écrire « non ».

4.7 Ce qui vient ensuite

Le tableau est en mémoire et chaque colonne dit ce qu'elle signifie. L'unité suivante le travaille : sélectionner et filtrer sans l'avertissement que personne ne lit, puis regrouper jusqu'au numérateur et au dénominateur qu'exige réellement un indicateur.

Troisième partie

Travailler le tableau

CHAPITRE 5

Sélectionner et filtrer, et l'avertissement devenu erreur

Leçon 5 sur 8 · 85 min

loc, iloc et masques booléens utilisés délibérément — plus ce que signalait le SettingWithCopyWarning, pourquoi pandas 3 l'a remplacé par ChainedAssignmentError, et ce que cela change pour le code que vous avez déjà.

5.1 Trois façons de sélectionner, et quand chacune convient

```
import pandas as pd

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

Par étiquette, avec `loc`. Les lignes par valeur d'index, les colonnes par nom. C'est celle vers laquelle aller par défaut, car elle dit ce qu'elle fait.

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

Par position, avec `iloc`. Lignes et colonnes par position entière.

```
muac.iloc[0]           # premiere ligne
muac.iloc[:5, :3]      # cinq premieres lignes, trois premieres colonnes
```

`iloc` sert à regarder, non à raisonner. La position dépend de l'ordre du fichier, et un script qui dit `iloc[:, 3]` casse silencieusement le jour où l'export gagne une colonne. Réservez-le à l'inspection et au travail réellement positionnel.

Une colonne seule est une Series ; une liste de colonnes est un DataFrame :

```
muac["muac_mm"]              # Series
muac[["muac_mm"]]            # DataFrame a une colonne
muac[["commune", "muac_mm"]] # DataFrame a deux colonnes
```

Le double crochet n'est pas un choix esthétique. Certaines fonctions exigent un DataFrame, et une Series échouera loin de l'endroit où l'erreur a été commise.

5.2 Masques booléens

Un masque est une Series de `True`/`False` de même longueur que le tableau.

```
severe = muac["muac_mm"] < 115
print(severe.sum())              # combien
print(severe.mean())             # quelle part
muac.loc[severe]
```

`.sum()` et `.mean()` sur un masque booléen donnent l'effectif et la proportion. C'est la façon la plus rapide de répondre à « combien » et « quelle part » sans `groupby`.

5.2.1 Combiner des conditions

```
# Les parentheses sont obligatoires : & lie plus fort que <
under_five = (muac["age_months"] < 60) & (muac["muac_mm"] < 125)

# Utilisez & | ~ -- non and / or / not, qui ne savent pas operer element par element
either = (muac["muac_mm"] < 115) | (muac["oedema"] == True)
```

Écrire `and` ici lève `ValueError: The truth value of a Series is ambiguous`. Le message est opaque la première fois et signifie toujours la même chose : vous avez utilisé un opérateur scalaire sur un vecteur.

5.2.2 Les valeurs manquantes ne sont pas False

C'est le piège qui change une charge de cas.

```
print(len(muac))                                  # 4218
print((muac["muac_mm"] < 125).sum())              # ne compte que les enfants mesures
print(muac["muac_mm"].isna().sum())               # ceux exclus silencieusement
```

Une comparaison avec `NA` vaut `NA`, et `NA` n'est pas sélectionné. Un filtre écarte donc discrètement tout enfant non mesuré — ce qui est juste si vous vouliez dire « enfants mesurés en dessous de 125 mm » et faux si vous vouliez dire « enfants dont on ne sait pas qu'ils sont au-dessus de 125 mm ».

Soyez explicite sur ce que vous vouliez dire :

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

print(f"evalues : {measured.sum()}, cas de MAG : {gam.sum()}")
print(f"taux de MAG : {gam.sum() / measured.sum():.3f}")
```

Le dénominateur est tout l'enjeu. Diviser par `len(muac)` au lieu de `measured.sum()` rapporte une prévalence plus faible pour une raison sans rapport avec la nutrition.

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

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

`between` est inclusif aux deux bornes par défaut et — c'est l'essentiel — renvoie `False` pour `NA`. Donc `~muac["muac_mm"].between(80, 220)` compte une mesure manquante comme une mesure impossible. Ce sont deux défauts différents et un seul est une erreur de saisie :

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

`query` se lit bien pour les conditions longues et mérite d'être connu :

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

Il est plus lent et masque les fautes de frappe dans une chaîne : préférez les masques dans un script qui doit échouer bruyamment.

5.3 L'avertissement devenu erreur

Les anciennes versions de pandas produisaient un avertissement que des générations d'analystes ont appris à masquer :

```
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
```

Il signalait une **affectation chaînée** — sélectionner, puis affecter à la sélection :

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

Le premier crochet produit un nouvel objet. Que l'affectation atteigne ou non le tableau d'origine dépendait de l'agencement mémoire, ce qui faisait que la même ligne pouvait fonctionner lundi et pas mardi. L'avertissement existait parce que pandas ne pouvait pas vous dire lequel des deux s'était produit.

5.3.1 Ce que fait pandas 3 à la place

Le Copy-on-Write est toujours actif et ne peut plus être désactivé. Deux conséquences.

Le comportement est désormais défini. Une sélection n'écrit jamais dans son parent — non pas parfois, jamais. `SettingWithCopyWarning` n'existe plus ; `pd.errors.SettingWithCopyWarning` lève `AttributeError`.

L'affectation chaînée vous dit qu'elle n'a rien fait.

```
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` dérive de `Warning` : par défaut la ligne s'affiche et l'exécution continue. Dans un script dont les chiffres comptent, promouvez-la :

```
import warnings

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

Le script s'arrête alors au lieu de rapporter un chiffre calculé à partir d'une modification qui n'a jamais eu lieu.

5.3.2 La forme correcte, et ce qui change pour le code existant

```
# Un seul .loc, les deux axes, une seule operation.
muac.loc[muac["commune"] == "Gonaives", "muac_mm"] = pd.NA
```

Si vous maintenez du code écrit pour pandas 1 ou 2, le risque de migration va dans un seul sens : une affectation chaînée qui *modifiait* le tableau ne le modifie plus. Le chiffre change et rien ne plante. Cherchez `]["` et les appels à `.copy()` ajoutés pour faire taire l'ancien

avertissement — ces copies sont désormais inutiles, et chacune retirée est une économie réelle sur un gros tableau.

5.3.3 `.copy()` et les cas où vous le voulez encore

Sous Copy-on-Write, `df[mask]` est déjà indépendant : un `.copy()` défensif n'apporte rien. Il reste utile quand il documente une intention :

```
# Ce sous-ensemble va etre modifie et n'est la vue de rien.
gonaives = muac.loc[muac["commune"] == "Gonaives"].copy()
gonaives["flag"] = gonaives["muac_mm"] < 115
```

Ajouter une colonne à une sélection non copiée fonctionne et c'est le cas courant ; le `.copy()` ci-dessus est une note au lecteur, non une nécessité.

5.4 Ajouter et modifier des colonnes

`assign` renvoie un nouveau tableau et se chaîne, ce qui garde une chaîne de traitement lisible :

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

Le `lambda d:` compte : il désigne le tableau *à ce point de la chaîne*, si bien que `sam` peut se définir à partir de colonnes créées par `assign` une ligne plus tôt. Référencer `muac` directement dans la chaîne utiliserait le tableau d'avant-filtre et désalignerait tout.

Pour des conditions à plus de deux branches, `np.select` vaut mieux que des `where` imbriqués :

```
import numpy as np

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

Les conditions sont évaluées dans l'ordre et la première qui correspond l'emporte : `< 115` doit donc venir en premier. Noter que `default` attrape aussi les valeurs manquantes, ce qui n'est presque jamais souhaité — remettez-les explicitement :

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

5.5 Trier n'est pas classer

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

Trier range des lignes. Cela ne dit rien de la réalité des écarts entre lignes voisines — un point que développe le projet *Tableau de bord de programme nutritionnel*, où neuf communes sur douze ont des intervalles de confiance qui se recouvrent et où l'ordre de tri n'est pas une information.

`nlargest` est la forme lisible du « top N » et est plus rapide qu'un tri du tableau entier :

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

5.6 Ce qui vient ensuite

Vous savez sélectionner un sous-ensemble et le modifier sans vous demander si la modification a eu lieu. La leçon suivante agrège : `groupby`, agrégation nommée, et l'obtention d'un numérateur et d'un dénominateur issus de la même opération, pour qu'ils ne puissent pas diverger.

CHAPITRE 6

Regrouper jusqu'à un numérateur et un dénominateur

Leçon 6 sur 8 · 90 min

groupby, agrégation nommée, et les deux réglages par défaut qui changent silencieusement un dénominateur — groupes manquants écartés et catégories non observées. Produire les deux moitiés d'un indicateur en une seule opération pour qu'elles ne puissent pas diverger.

6.1 Un indicateur, ce sont deux nombres

Presque tout chiffre rapporté dans ce secteur est un numérateur sur un dénominateur, et presque toute dispute sur un chiffre est une dispute sur le dénominateur. La conséquence pratique pour votre code : **calculez les deux dans la même opération.**

Deux calculs séparés divergent. L'un reçoit un filtre que l'autre n'a pas, quelqu'un modifie une ligne, et le rapport devient un chiffre que personne ne peut reconstituer.

```
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
```

Trois colonnes, et chacune porte quelque chose. `screened` est le nombre d'enfants venus ; `measured` le nombre ayant produit un PB exploitable, et c'est le dénominateur ; `gam_cases` est le numérateur. Publier le taux sans le dénominateur à côté, c'est ainsi que la dispute commence.

Noter que `screened` et `measured` diffèrent — 4 218 enfants ont été dépistés et 4 146 ont un PB. Diviser par le mauvais décale le taux du district assez pour compter et trop peu pour se voir.

6.2 L'agrégation nommée

La forme employée ci-dessus est celle à standardiser :

```
.agg(
    nom_de_sortie=("colonne_d_entree", "fonction"),
    ...
)
```

Elle nomme la colonne de sortie à l'endroit où l'agrégation est définie : plus de seconde étape pour renommer `muac_mm_mean` en quelque chose de compréhensible, et pas d'index de colonnes à plusieurs niveaux à aplatir.

Les formes plus anciennes fonctionnent encore et méritent d'être reconnues dans le code d'autrui :

```
muac.groupby("commune")["muac_mm"].mean()               # une colonne, une fonction
muac.groupby("commune").agg({"muac_mm": ["mean", "std"]})  # colonnes MultiIndex
```

La seconde produit des colonnes comme `("muac_mm", "mean")`, qu'il faut ensuite aplatir. L'agrégation nommée évite le problème plutôt que de le résoudre.

Les fonctions peuvent être les vôtres :

```
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` exclut les manquants ; `size` les inclut. Cette seule distinction fait la différence entre les deux dénominateurs ci-dessus, et mérite d'être énoncée à voix haute chaque fois que vous employez l'un des deux.

6.3 Deux réglages par défaut qui changent un dénominateur

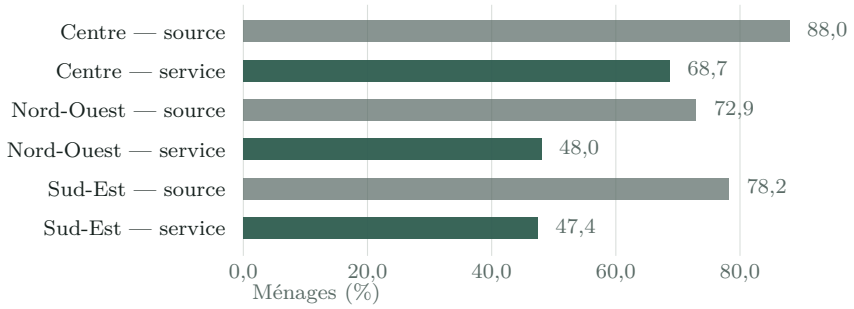

FIGURE 6.1 – Source d'eau améliorée contre service de base, par district. Compter les sources surestime la couverture partout ; l'écart correspond aux ménages dont la source est améliorée et située à plus de trente minutes.

Les deux barres de chaque paire proviennent du même `groupby`. La seule différence tient aux lignes que compte le numérateur — et elle déplace la réponse de vingt à trente points

dans chaque district. Voilà ce que signifie en pratique « le dénominateur est tout l'enjeu ».

6.3.1 Les clés de groupe manquantes sont écartées

```
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}
```

Par défaut, `groupby` écarte les lignes dont la clé de groupe est manquante. Elles disparaissent du tableau **et** du total : les parties ne font plus le tout, et rien ne dit pourquoi.

Dans ce secteur, la clé manquante est généralement la plus intéressante : une formation sanitaire sans district saisi, un ménage sans code de site, un enfant sans commune. Utilisez `dropna=False` et décidez délibérément, ou vérifiez que la clé est complète :

```
assert muac["commune"].notna().all(), "des lignes sans commune seraient ecartees"
```

6.3.2 Les catégories non observées réapparaissent en lignes vides

Si la clé de groupe est catégorielle, le comportement par défaut est désormais de ne renvoyer que les catégories observées :

```
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 -- inclut un "z" vide
```

Les deux sont justes, pour des questions différentes. Rapporter sur les formations ayant transmis des données appelle `observed=True` ; rapporter la couverture face à une liste de formations qui *auraient dû* transmettre appelle `observed=False`, car une formation à zéro ligne est justement le constat.

Passez-le explicitement. Le comportement par défaut a changé d'une version de pandas à l'autre, et un script qui s'y fie produit un autre tableau sur la machine d'un collègue.

6.4 Regrouper sur plusieurs clés

```
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"))
)
```

Le résultat porte un MultiIndex. Deux façons de travailler avec :

```
by_commune_month.loc["Gonaives"]                    # une commune, tous les mois
by_commune_month.reset_index()                      # retour a des colonnes plates
```

`reset_index()` avant toute écriture en CSV, systématiquement — sinon les colonnes d'index disparaissent ou arrivent sans nom.

Pour transformer le résultat long en tableau large attendu par un rapport :

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

`fill_value=0` est sans risque ici, car une commune-mois sans cas en avait réellement zéro. Ce serait faux pour un taux, où l'absence signifie « non calculé » et non « zéro pour cent » — distinction à vérifier chaque fois que vous y recourez.

6.5 transform : une statistique de groupe sur chaque ligne

`agg` réduit ; `transform` renvoie une valeur par ligne d'origine. C'est ainsi qu'on compare une ligne à son propre groupe sans jointure :

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

Utile pour signaler un site dont les mesures s'écartent systématiquement du reste — c'est exactement le contrôle qu'emploie le projet *Analyse d'enquête SMART* pour repérer une équipe mesurant 0,5 kg trop léger.

6.6 apply, et pourquoi l'éviter

`apply` exécute une fonction Python par groupe et constitue de loin l'opération la plus lente de pandas. C'est aussi la plus souple, donc celle vers laquelle on se tourne d'abord.

```
# Fonctionne, mais lent, et renvoie un objet dont la forme depend de la fonction.
muac.groupby("commune").apply(lambda g: g["muac_mm"].mean(), include_groups=False)
```

Préférez une agrégation nommée. Réservez `apply` aux opérations exigeant réellement le tableau du groupe entier — une régression par groupe, un classement par groupe avec égalités départagées sur une seconde colonne — et passez `include_groups=False`, désormais nécessaire pour éviter que les colonnes de regroupement ne soient transmises à votre fonction.

6.7 Les tableaux croisés se lisent mieux dans un rapport

`pivot_table` est un `groupby` suivi d'un `unstack`, avec une signature plus agréable :

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

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

`crosstab` avec `normalize="index"` donne les proportions par ligne, ce qu'un tableau de référencement doit généralement montrer. Gardez les effectifs à côté : un taux de référencement de 50 % sur quatre enfants n'est pas le même fait que 50 % sur quatre cents, et seul l'effectif les distingue.

6.8 Écrire le résultat

```
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` après `reset_index()` — sinon vous obtenez une colonne d'entiers sans nom que quelqu'un ouvrira plus tard dans Excel et utilisera pour trier.

Pour un tableau destiné à être lu plutôt que calculé, écrivez la définition avec lui :

```
definitions = pd.DataFrame([
    ("MAG", "PB < 125 mm ou oedemes bilateraux prenant le godet",
     "enfants avec une mesure de PB ou une evaluation d'oedemes saisie"),
], columns=["indicateur", "numerateur", "denominateur"])

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

Un nombre et sa définition voyagent ensemble, sinon la dispute mensuelle sur le dénominateur recommence.

6.9 Ce qui vient ensuite

Le tableau agrège correctement et les deux moitiés de chaque taux proviennent d'une seule opération. L'unité suivante règle la dernière pièce — dates, âges et périodes de rapportage, où un âge en mois décide du standard de croissance applicable — puis enveloppe l'ensemble dans un script qui s'exécute sur la machine de quelqu'un d'autre.

Quatrième partie

Produire et transmettre

CHAPITRE 7

Dates, âges et périodes de rapportage

Leçon 7 sur 8 · 80 min

L'âge en mois révolus calculé correctement plutôt qu'en divisant des jours, des périodes de rapportage qui survivent à un mois partiel, et l'arithmétique de frontière dont dépend un standard de croissance.

7.1 Pourquoi cela mérite une leçon entière

L'arithmétique des dates paraît être la partie facile et produit certaines des erreurs les plus coûteuses du secteur. Un âge en mois décide du standard de croissance auquel un enfant est comparé. Une période de rapportage décide si une formation sanitaire compte comme ayant rapporté. Une frontière de mois décide si une distribution tombe dans les chiffres de ce trimestre ou du suivant.

Aucun de ces cas n'échoue bruyamment.

7.2 Analyser la date, une fois, explicitement

```
import pandas as pd

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

Deux habitudes à prendre :

— **Énoncez le format.** Un format inféré peut changer d'un fichier à l'autre selon la composition des valeurs — la même colonne lue en jour-d'abord dans un export et en mois-d'abord dans le suivant.

— **Utilisez `errors="raise"`.** L'alternative, `errors="coerce"`, transforme toute date illisible en valeur manquante : un fichier au mauvais format de date s'importe donc « avec succès », avec une colonne de dates entièrement vide.

`01/02/2024` est soit le 1er février, soit le 2 janvier, et le fichier ne vous dira pas lequel. Si un export vous livre des dates ambiguës, `dayfirst=True` est une décision que vous documentez, non un réglage que vous acceptez.

7.3 L'âge en mois révolus

L'approche évidente divise des jours par un mois moyen :

```
# Faux assez souvent pour que cela compte.
age_months = ((visit_date - date_of_birth).dt.days / 30.4375).astype(int)
```

Sur les deux mille premiers jours de vie, ce calcul diverge de la bonne réponse sur 54 d'entre eux — toujours d'un mois, toujours à une frontière. Un enfant de 60 jours est rapporté à 1 mois alors qu'il en a 2.

Calculez directement les mois révolus :

```
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)
    # Le quantieme n'est pas atteint : le dernier mois n'est pas revolu.
    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]
```

Le deuxième enfant est à un jour de ses deux ans : il a 23 mois, non 24. Ce seul jour est toute la raison d'être de cette fonction.

7.3.1 Pourquoi un mois compte ici

Les standards de croissance de l'OMS changent de mesure exactement à 24 mois. En dessous de 24 mois, l'enfant est mesuré couché et comparé à la courbe poids-pour-longueur ; à partir de 24 mois, debout, contre le poids-pour-taille. Les deux mesures diffèrent d'environ 0,7 cm et les deux courbes sont des tables différentes.

Un enfant poussé de 23 à 24 mois par une règle d'arrondi est comparé au mauvais standard, et son score Z se déplace. Faites cela à quelques centaines d'enfants près de la frontière et l'estimation de prévalence se déplace avec eux — c'est pourquoi le projet *Analyse d'enquête SMART* rapporte l'attraction des âges vers les années entières comme un constat de plausibilité et non comme une curiosité.

Les tranches d'âge sont l'autre endroit. `0-5`, `6-23`, `24-59` mois sont les tranches nutritionnelles standard, et un enfant à exactement 24 mois appartient à la troisième :

```
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` rend chaque intervalle `[bas, haut)` — fermé à gauche, ouvert à droite — de sorte que 24 mois tombe dans `24-59m` et non dans `6-23m`. Le comportement par défaut est l'inverse, et il est faux pour toutes les tranches d'âge employées dans ce secteur.

Vérifiez les frontières plutôt que de leur faire confiance :

```
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 Les périodes de rapportage

Une période est un mois, un trimestre ou une année *pour lequel* on rapporte, et ce n'est pas la même chose qu'une 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
```

Les périodes se trient correctement, s'affichent lisiblement et — contrairement à une chaîne comme `"2024-01"` — acceptent l'arithmétique :

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

Les chaînes se trient aussi correctement si vous employez l'ordre ISO, bonne raison d'utiliser `%Y-%m` et jamais `%m/%Y`. Mais une chaîne ne sait pas répondre à « le mois précédent », et cette question surgit dans tout calcul de tendance.

7.4.1 Les périodes partielles sont le piège

Ce registre court du 15 janvier au 13 décembre. Janvier porte 207 dépistages et décembre 156, contre une moyenne mensuelle proche de 375. Aucun de ces deux mois n'est une baisse ; les deux sont partiels.

```
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")]
)
```

Étiquetez-les plutôt que de les supprimer. Un lecteur qui voit décembre s'effondrer sur un graphique conclut que le programme s'est écroulé ; un lecteur qui ne voit aucun décembre se demande où sont passées les données. Nommer le mois partiel est la seule option qui répond aux deux.

La même logique vaut pour une période inachevée. Un cumul en cours de mois comparé à des mois révolus est la façon la plus courante dont un tableau de bord affiche une baisse fictive tous les mois, et cela arrive parce que les deux s'appellent « le total mensuel ».

7.4.2 Aligner deux sources sur des périodes

L'extraction de vaccination enregistre une date de début de période plutôt qu'une date de prestation :

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

Convertir les deux sources en `Period` avant la jointure est ce qui les fait coïncider. Joindre une date à un début de période ne produit silencieusement aucune correspondance pour chaque mois où une source a utilisé le dernier jour et l'autre le premier.

7.5 Rééchantillonner une série temporelle

Quand l'index est un datetime, `resample` agrège par période sans colonne de regroupement manuelle :

```
daily = muac.set_index("screening_date")
weekly = daily.resample("W")["child_id"].size()
monthly = daily.resample("MS")["child_id"].size()      # MS = debut de mois
```

`resample` comble les trous : une semaine sans dépistage apparaît à zéro plutôt que d'être absente. C'est juste pour une série temporelle et faux pour un dénominateur — une école fermée quinze jours n'a pas de jours de présence, et non une présence nulle ; le projet *Assiduité scolaire* montre ce que coûte le remplissage.

7.6 Les durées

Soustraire deux datetimes donne un `Timedelta` :

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

`.dt.days` tronque vers zéro : un écart de 23 heures vaut 0 jour. Si le même jour compte — et pour un référencement en protection, c'est le cas — dites-le explicitement plutôt que de laisser la troncature décider :

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

Une durée négative est un constat de qualité des données, non une valeur dont on prend la valeur absolue :

```
impossible = referrals["days_to_service"] < 0
print(f"service saisi avant le referencement : {impossible.sum()}")
```

7.7 Les fuseaux horaires, brièvement

Les données de programme sont généralement des dates locales naïves et doivent le rester. Attacher un fuseau à une date de dépistage invite une conversion qui fera franchir minuit à un enregistrement et le fera basculer dans le mauvais mois de rapportage.

Là où des horodatages portent réellement un fuseau — heures de soumission des formulaires CommCare ou Kobo côté serveur — convertissez une fois, à la frontière, et stockez la date locale :

```
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
```

L'horodatage de soumission et la date de la visite qu'il enregistre sont deux choses différentes, et rapporter la première quand on veut dire la seconde décale toute soumission du soir vers le lendemain.

7.8 Ce qui vient ensuite

Chaque colonne dit désormais ce qu'elle signifie, y compris celles faites de dates. La dernière leçon transforme l'analyse en un script qu'un autre chargé pourra exécuter au trimestre prochain sur un autre export, sans rien modifier d'autre que ses arguments.

CHAPITRE 8

Un script qu'un autre chargé peut exécuter

Leçon 8 sur 8 · 90 min

Des arguments au lieu de modifications, de la journalisation au lieu de print, un échec bruyant au lieu d'un chiffre faux — transformer l'analyse en quelque chose qui tourne au trimestre prochain sans vous dans la pièce.

8.1 Le test de la transmission

Une analyse est terminée quand quelqu'un d'autre peut l'exécuter sur l'export du trimestre prochain sans modifier le code. Non pas « sans trop le modifier » — sans le modifier.

Ce critère élimine les trois habitudes que tout notebook accumule : un chemin écrit pour une machine, une valeur changée à la main entre deux exécutions, et une étape qui ne fonctionne que si l'on sait déjà quelle cellule lancer d'abord.

8.2 Des arguments, non des modifications

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

def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(
        description="Calcule les taux de MAG et MAS par commune depuis un registre."
    )
    parser.add_argument(
        "--input", type=Path, required=True,
        help="Chemin du CSV du registre de depistage.",
    )
    parser.add_argument(
        "--output", type=Path, required=True,
        help="Repertoire ou ecrire les tableaux. Cree s'il manque.",
    )
    parser.add_argument(
        "--gam-threshold", type=int, default=125,
        help="Seuil de PB en mm pour la malnutrition aigue globale (defaut : 125).",
    )
    parser.add_argument(
        "--min-assessment-rate", type=float, default=0.8,
        help="Les communes sous ce taux d'evaluation sont signalees, non retirees.",
    )
    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` convertit la chaîne pour vous, et `required=True` fait qu'un argument oublié produit un message clair plutôt qu'un `KeyError` deux cents lignes plus loin.

Mettez les seuils dans les arguments et les valeurs par défaut dans le code. Un seuil qu'on ne change qu'en éditant une ligne finira par être changé et non remis. Un seuil qui est un argument avec un défaut documenté est visible dans la commande qui a produit la sortie — ce que vous voulez quand on vous demandera comment le chiffre du trimestre dernier a été calculé.

`-help` est généré depuis les mêmes déclarations, et c'est la documentation que les gens lisent réellement :

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

8.3 Échouer bruyamment, tôt

Un script qui produit un chiffre faux en silence est pire qu'un script qui plante. Vérifiez ce que vous supposez, à l'endroit où vous le supposez.

```
def read_register(path: Path) -> pd.DataFrame:
    if not path.exists():
        raise SystemExit(f"Fichier d'entree introuvable : {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"Colonnes absentes de l'entree : {sorted(missing)}")

    if muac["child_id"].isna().any():
        raise SystemExit("Des lignes n'ont pas de child_id ; arret.")

    return muac
```

`SystemExit` avec un message sort en statut non nul et affiche une ligne lisible, plutôt qu'une trace d'appels que le lecteur doit interpréter. Employez-le pour « cette entrée n'est pas celle qu'on m'avait promise » ; employez `assert` pour « ceci ne peut pas arriver à moins que le code soit faux ».

Transformez en erreur l'avertissement pandas qui signale une défaillance silencieuse :

```
import warnings
import pandas as pd

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

L'affectation chaînée ne modifie jamais le tableau, et par défaut elle se contente d'avertir. Dans un script dont quelqu'un rapportera la sortie, s'arrêter est la bonne réponse.

8.4 Journaliser, non imprimer

`print` part sur la sortie standard au milieu des résultats, n'a pas de niveau de gravité et ne peut pas être atténué.

```
import logging

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

log.info("Lu %d lignes depuis %s", len(muac), args.input)
log.warning("%d communes sous le plancher de taux d'evaluation", len(flagged))
log.info("Ecrit %s", output_path)
```

```
09:14:02 INFO     Lu 4218 lignes depuis data/raw/muac-...v1.csv
09:14:02 WARNING  2 communes sous le plancher de taux d'evaluation
09:14:02 INFO     Ecrit outputs/tables/gam_by_commune.csv
```

Trois choses que `print` n'apporte pas : un horodatage, un niveau de gravité filtrable, et la possibilité d'ajouter `-verbose` sans toucher à chaque appel.

Utilisez la forme `%s` plutôt qu'une f-string — le formatage est entièrement évité quand le niveau est désactivé.

Journalisez les nombres qui permettraient de reconstituer l'exécution : lignes lues, lignes exclues et pourquoi, dénominateur, chemin de sortie. Un journal qui dit « terminé » n'apprend rien à personne.

8.5 La forme du script

```
# scripts/indicator_table.py
"""Calcule les taux de MAG et MAS par commune depuis un registre de PB."""

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("Lu %d lignes", 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("Taux d'evaluation sous le plancher : %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("Ecrit %s (%d communes)", destination, len(table))

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

Quatre aspects de cette structure sont délibérés.

Les fonctions prennent des arguments et renvoient des valeurs. `indicator_table` ne lit aucun fichier, ignore où va la sortie et ne touche à aucune variable globale. C'est ce qui la rend testable et ce qui permet au notebook de l'importer.

`main()` est le seul endroit qui fait des entrées-sorties. Lecture, écriture et journalisation y vivent ; tout le reste est du calcul.

Le garde `if __name__ == "__main__"` fait qu'importer ce module depuis un notebook n'exécute rien. Sans lui, `from indicator_table import indicator_table` exécute tout le script.

Les données minces sont signalées, non supprimées. Une commune sous le plancher de taux d'évaluation reste dans le tableau avec un avertissement à côté. La supprimer changerait le dénominateur du district et rien dans la sortie ne le dirait.

8.6 Tester la partie qui calcule

Une fois le calcul devenu une fonction prenant un tableau, le tester tient en trois lignes :

```
# 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      # l'enfant non mesure n'est pas au
    denominateur
    assert table.loc[0, "gam_rate"] == 0.5
```

Ce test encode la décision de la leçon 6 — quels enfants entrent au dénominateur — sous une forme qui échoue si quelqu'un la change. Il mérite d'être écrit pour tout indicateur dont la définition a fait débat.

8.7 Rendre l'exécution reproductible

Consignez ce qui a produit la sortie, à côté de la sortie :

```
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 "pas un depot git",
    }

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

Six mois plus tard, « quel seuil a produit ce tableau » a une réponse qui ne dépend de la mémoire de personne.

Puis le contrôle de la leçon 2, qui est le véritable test :

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

Si cela ne restaure pas les sorties à l'identique, quelque chose n'est pas reproductible.

8.8 Transmettre

Le README de la leçon 2 a besoin d'une ligne de plus désormais — la commande avec ses vrais arguments :

```
## Executer

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

Les seuils valent par defaut PB < 125 mm (MAG) et < 115 mm (MAS).
Passez --gam-threshold pour changer ; la valeur employee est consignee
dans outputs/tables/run.json.
```

Deux commandes, et les seuils visibles sans ouvrir le code. Toute la transmission est là.

8.9 Où ce cours s'arrête

Vous savez construire un environnement réinstallable hors ligne, lire n'importe quel export sans le corrompre, transformer ses codes en valeurs, agréger vers un numérateur et un dénominateur issus de la même opération, maîtriser l'arithmétique des dates, et remettre le résultat à quelqu'un d'autre sous forme de script plutôt que de service rendu.

Ce que ce cours n'a délibérément pas enseigné, c'est *quoi* calculer — quel indicateur, quel dénominateur, quel seuil, et comment le défendre. C'est l'objet de *Fondamentaux de l'analyse de données*, et les cours sectoriels du module **Analyses sectorielles** de la feuille de route le poussent plus loin, jusqu'aux classifications que votre cluster vous impose.

Si votre équipe travaille en R plutôt qu'en Python, *R pour les données de programme* couvre le même terrain dans ce langage.

À propos de cette édition

Ce PDF est produit à partir de la même source que la version web de la formation ; les deux ne peuvent donc pas diverger. L'édition en ligne comporte le code exécutable, les jeux de données téléchargeables et les corrections publiées depuis la production de ce fichier.

> Tous les jeux de données cités dans cette formation sont synthétiques ou entièrement anonymisés. Aucun enregistrement ne renvoie à une personne réelle.

Cette formation est publiée en anglais et en français. L'édition française est une adaptation professionnelle reprenant la terminologie employée par l'UNICEF, l'OMS, le PAM, OCHA et les ministères nationaux — et non une traduction littérale.

Consulter et exécuter la formation en ligne sur data-analysis.cassion.dev/fr/cours/python-for-programme-data

Généré le 27 juillet 2026.