PYTHON FOR DATA ENGINEERING

# Foundations, tooling and a capstone build

A practical path through Python for people who will use it to move, shape and monitor data — not just to script. Each topic links to a video and ends with working code you can run.

01 — GETTING STARTED

# Why Python for Data Engineering

### Glue for everything

One language to move data between databases, APIs, files and warehouses, instead of a different tool for each hop.

### A huge ecosystem

pandas, Airflow, SQLAlchemy, PySpark, requests — the standard data tools are all Python, and they interoperate.

### Readable under pressure

Code a teammate can read and fix during a failed overnight run is worth more than code that is merely clever.

Before the syntax: a data engineer mostly writes code that reads something, changes its shape, and writes it somewhere else, reliably, on a schedule. Everything below builds toward that.

▶ **Watch:** Python for Beginners — Full Course (freeCodeCamp.org) — https://www.youtube.com/watch?v=eWRfhZUzrAc

02 — LANGUAGE BASICS

# Variables, Data Types & Operators

- **Variables:** No type keyword — a name is bound to a value with =, and can be rebound to a different type later.
- **Core types:** int, float, str, bool, and None — the value that means "nothing here", not zero and not an empty string.
- **f-strings:** The standard way to build strings from variables: f"{name} loaded {count} rows".

```
rows_loaded = 0
source_name = "orders.csv"
is_valid = rows_loaded > 0

print(f"{source_name}: {rows_loaded} rows")
print(f"valid={is_valid}")
```

▶ **Watch:** Python for Beginners — Full Course (freeCodeCamp.org) — https://www.youtube.com/watch?v=eWRfhZUzrAc

03 — LANGUAGE BASICS

# Control Flow: if / for / while

- **if / elif / else:** Branches on a condition. Indentation is the block — there are no braces to forget or misplace.
- **for:** Iterates over anything iterable: a list, a file, a database cursor, an API page of results.
- **while:** Repeats until a condition is false — the natural shape for "keep polling until the job finishes".

```
for row in load_rows():
    if row["amount"] is None:
        continue  # skip incomplete rows
    elif row["amount"] < 0:
        raise ValueError(f"negative amount: {row}")
    else:
        process(row)
```

▶ **Watch:** If, Else & Elif Statements (Corey Schafer) — https://www.youtube.com/watch?v=DZwmZ8Usvnk

04 — STRUCTURE

# Functions & Modules

- **Functions:** def gives a block of logic a name and inputs — the unit you test, reuse and reason about.
- **Type hints:** def clean(row: dict) -> dict documents intent; tools like mypy can check it, but Python won't enforce it at runtime.
- **Modules:** Any .py file is importable. A pipeline is usually a handful of small modules, not one long script.

```
def clean_row(row: dict) -> dict:
    row["email"] = row["email"].strip().lower()
    return row

# in another file:
from cleaning import clean_row
rows = [clean_row(r) for r in raw_rows]
```

▶ **Watch:** Functions (Corey Schafer) — https://www.youtube.com/watch?v=9Os0o3wzS_I

05 — STRUCTURE

# Core Data Structures

### list

An ordered, changeable sequence. A batch of rows read from a file usually starts life as a list.

### dict

Key-value pairs. The natural shape for one row of semi-structured data — a JSON record or a CSV line.

### tuple / set

tuple: fixed, ordered (a coordinate, a return value). set: unique, unordered — de-duplicating IDs.

```
row = {"id": 104, "country": "ZA", "amount": 199.50}
seen_ids = set()
for r in rows:
    if r["id"] not in seen_ids:
        seen_ids.add(r["id"])
        cleaned.append(r)
```

▶ **Watch:** Lists, Tuples, Sets & Dictionaries (Programming and Math Tutorials) — https://www.youtube.com/watch?v=R-HLU9Fl5ug

06 — I/O

# Working with Files: pathlib, CSV & JSON

- **pathlib:** Path("data") / "orders.csv" builds a path that works the same on Linux, macOS and Windows.
- **with open(...):** Always open files with with — it closes the file automatically, even if the code inside raises.
- **csv & json:** Standard-library modules for the two formats a pipeline touches constantly — no extra install needed.

```
import csv, json
from pathlib import Path

path = Path("data") / "orders.csv"
with open(path, newline="") as f:
    rows = list(csv.DictReader(f))

Path("out.json").write_text(json.dumps(rows))
```

▶ **Watch:** Read & Write Files in Python (techTFQ) — https://www.youtube.com/watch?v=Jsnw6HLASZA

07 — TOOLING

# Virtual Environments & Packages

- **Why isolate:** Two projects on one machine will eventually need two different versions of the same library.
- **venv:** Python's built-in tool for a project-local, disposable environment — no separate install required.
- **requirements.txt:** Pins the exact packages a project needs, so a teammate (or a Docker build) can reproduce it exactly.

```
python3 -m venv .venv
source .venv/bin/activate   # or .venv\Scripts\activate
pip install pandas requests
pip freeze > requirements.txt

# later, or on another machine:
pip install -r requirements.txt
```

▶ **Watch:** Virtual Environments — venv (Corey Schafer) — https://www.youtube.com/watch?v=Kg1Yvry_Ydk

08 — THE DATA STACK

# Pandas Basics

- **DataFrame:** A table in memory — rows and named columns, with vectorised operations instead of manual loops.
- **Read / write anything:** read_csv, read_sql, read_json, and their to_ counterparts cover most sources a pipeline touches.
- **Vectorised over looped:** df[df.amount > 0] filters an entire column at once — far faster than a Python-level for loop.

```
import pandas as pd

df = pd.read_csv("orders.csv")
df = df[df["amount"] > 0]
daily = df.groupby("date")["amount"].sum()
daily.to_csv("daily_totals.csv")
```

▶ **Watch:** Pandas Tutorial, Part 1 (Corey Schafer) — https://www.youtube.com/watch?v=ZyhVh-qRZPA

09 — THE DATA STACK

# Calling APIs & Working with JSON

- **requests:** The de facto standard for HTTP in Python — not in the standard library, but installed everywhere.
- **Status codes:** Always check response.status_code (or call .raise_for_status()) before trusting the response body.
- **Pagination:** Most real APIs page their results — a pipeline usually loops until the API says there is no next page.

```
import requests

resp = requests.get(
    "https://api.example.com/orders",
    params={"page": 1}, timeout=10,
)
resp.raise_for_status()
orders = resp.json()["results"]
```

▶ **Watch:** Requests Tutorial (Corey Schafer) — https://www.youtube.com/watch?v=tb8gHvYlCFs

10 — RELIABILITY

# Error Handling & Logging

- **try / except:** Catches a specific failure and decides what to do — never bare except:, which also swallows real bugs.
- **logging, not print:** print disappears when a pipeline runs unattended; logging.info/warning/error goes somewhere you can read later.
- **Fail loudly:** A pipeline that silently skips a broken batch is worse than one that stops and pages someone.

```
import logging
logging.basicConfig(level=logging.INFO)
log = logging.getLogger(__name__)

try:
    load_batch(batch)
except ValueError as e:
    log.error(f"batch {batch.id} rejected: {e}")
    raise
```

▶ **Watch:** Try/Except Error Handling (Corey Schafer) — https://www.youtube.com/watch?v=NIWwJbo-9_8

11 — CAPSTONE

# Project: An Open-Source Data Pipeline

Every topic above shows up together in one running system: a self-hosted, containerised ETL platform built entirely on open-source tools. It moves data from source databases and flat files into a target database, documents the structure of every table it loads, and makes that data available for analytics — through a web interface, with no manual scripting required to add a new pipeline.

- **Apache Airflow:** orchestrates and schedules every pipeline run
- **PostgreSQL:** the target database and the metadata store
- **Redis & Celery:** the task queue behind Airflow's workers
- **Flask:** a web UI for building pipelines and browsing the data catalogue, instead of hand-written config files
- **Docker Compose:** the whole stack starts with a single command

▶ **Watch:** Source code on GitHub — https://github.com/pleasurengobeni/datapipeline

12 — WHERE TO GO NEXT

# Your Python Journey

## 01 Write it

Variables, control flow, functions — comfortable enough to write a script without looking everything up.

## 02 Shape it

Lists, dicts and pandas — comfortable turning messy input into a clean table.

## 03 Connect it

Files, APIs and a database — comfortable moving data in and out of a pipeline.

## 04 Run it for real

Virtual environments, error handling, logging — comfortable shipping something that runs unattended.

### Continue learning

Real Python (realpython.com) for depth on any topic above · The datapipeline repo for a working system to read end to end · docs.python.org for the language reference itself.