

M C D A 5 5 1 1

Current Practices in Computing and Data Science

# Data Processing in Python

The Modern Python Data Stack

Somto Muotoe

January 2026

# Agenda

## Part 1: File Operations

- Why pathlib over os.path?
- Path manipulation basics
- Reading & writing files
- Directory operations

## Part 2: Pandas

- DataFrame fundamentals
- Reading data files
- Data manipulation
- Aggregations & grouping

## Part 3: Polars

- Why Polars over Pandas?
- Lazy vs Eager evaluation
- Expression API
- Performance comparison

## Part 4: DuckDB

- What is DuckDB?
- SQL on DataFrames
- When to use DuckDB
- Integration patterns

# Part 1

File Operations with pathlib

# Why pathlib?

## The Old Way: os.path

```
import os

# Joining paths - error prone
path = os.path.join("data", "sales", "2024.csv")

# Reading a file
with open(path, 'r') as f:
    content = f.read()

# Getting file info
exists = os.path.exists(path)
is_file = os.path.isfile(path)
name = os.path.basename(path)
```

String-based, verbose, platform issues

## The Modern Way: pathlib

```
from pathlib import Path

# Joining paths - clean and safe
path = Path("data") / "sales" / "2024.csv"

# Reading a file
content = path.read_text()

# Getting file info
exists = path.exists()
is_file = path.is_file()
name = path.name
```

Object-oriented, readable, cross-platform

# Path Basics

## Creating Paths

```
from pathlib import Path

# Current directory
cwd = Path.cwd()

# Home directory
home = Path.home() # C:\Users\somto

# From string
data_path = Path("data/sales")

# Absolute path
abs_path = Path("C:\Users\somto\projects")

# Joining paths with /
file_path = data_path / "report.csv"
```

## Path Properties

```
path = Path("data/sales/2024_q1.csv")

path.name      # "2024_q1.csv"
path.stem      # "2024_q1"
path.suffix    # ".csv"
path.parent    # Path("data/sales")
path.parts     # ("data", "sales", "2024_q1.csv")

# Resolve to absolute
path.resolve() # Full absolute path

# Check path type
path.is_file() # True/False
path.is_dir()  # True/False
path.exists()  # True/False
```

# Reading & Writing Files

## Reading Files

```
from pathlib import Path

path = Path("data/config.json")

# Read entire file as text
content = path.read_text()

# Read as bytes (for binary files)
binary = path.read_bytes()

# With encoding
content = path.read_text(encoding="utf-8")
```

## Writing Files

```
from pathlib import Path
import json

path = Path("output/results.json")

# Ensure parent directory exists
path.parent.mkdir(parents=True, exist_ok=True)

# Write text
path.write_text("Hello, World!")

# Write JSON data
data = {"name": "test", "value": 42}
path.write_text(json.dumps(data, indent=2))

# Write bytes
path.write_bytes(b"binary data")
```

Tip: `read_text()` and `write_text()` handle file opening/closing via context managers automatically

# Directory Operations

```
from pathlib import Path

# Create directories
Path("output/reports").mkdir(parents=True, exist_ok=True)

# List directory contents
for item in Path("data").iterdir():
    print(item.name, "- dir" if item.is_dir() else "- file")

# Find files with glob patterns
csv_files = list(Path("data").glob("*.csv"))                      # Direct children only
all_csvs = list(Path("data").rglob("*.csv"))                        # All subdirectories

# Pattern matching examples
Path(".").glob("*.py")          # All Python files
Path(".").glob("**/*.json")     # All JSON files (recursive)
Path(".").glob("data_202[0-9].csv") # data_2020.csv through data_2029.csv
```

Note: `rglob("*.csv")` is shorthand for `glob("**/*.csv")` - both search recursively

# Practical Example: Organizing Files

```
from pathlib import Path
import shutil

def organize_downloads():
    downloads = Path.home() / "Downloads"
    categories = {
        "images": [".jpg", ".png", ".gif"],
        "documents": [".pdf", ".docx", ".txt"],
        "data": [".csv", ".json", ".parquet"],
    }
    for path in downloads.iterdir(): # iterate over directory contents
        if not path.is_file():
            continue
        for category, extensions in categories.items():
            if path.suffix.lower() in extensions:
                dest_dir = downloads / category
                dest_dir.mkdir(exist_ok=True)
                shutil.move(path, dest_dir / path.name) # move to category folder
                break
```

Combines `iterdir()`, `is_file()`, `suffix`, `mkdir()`, and path joining

# Exercise: File Operations

## Task: Create a Student Profile System

1. Create a directory named `profiles`
2. Save your profile as JSON (name, languages, hobby)
3. List all `.json` files in the directory
4. Read and display your profile

```
from pathlib import Path
import json

def quick_profile_update():
    # 1. Create profiles directory
    # TODO: Use Path().mkdir()

    # 2. Create and save profile
    my_profile = {"name": "Your Name", "languages": ["Python"]}
    # TODO: Write to profiles/my_profile.json

    # 3. List all profiles
    # TODO: Use glob("*.json")

    # 4. Read and print
    # TODO: Use read_text() and json.loads()
```

# Part 2

Pandas: Fundamentals

# What is Pandas?

A Python library for data manipulation and analysis, providing fast, flexible data structures for working with tabular (spreadsheet-like) data.

## The De Facto Standard

- Created in 2008 by Wes McKinney
- Built on NumPy
- ~95% of Python data analysis uses pandas
- Massive ecosystem and community
- Excellent documentation

## Core Data Structures

- Series: 1D labeled array
- DataFrame: 2D labeled table

```
import pandas as pd

# Series: 1D labeled array
prices = pd.Series([1.20, 0.50, 0.80])
# 0    1.20
# 1    0.50
# 2    0.80

# DataFrame: 2D labeled table
df = pd.DataFrame({
    "product": ["Apple", "Banana", "Orange"],
    "price": [1.20, 0.50, 0.80],
})
#   product  price
# 0    Apple    1.20
# 1   Banana    0.50
# 2   Orange    0.80
```

# Reading Data

## Common File Formats

```
import pandas as pd
from pathlib import Path

data = Path("data")

# CSV files
df = pd.read_csv(data / "sales.csv")

# Excel files
df = pd.read_excel(data / "report.xlsx")

# JSON files
df = pd.read_json(data / "records.json")

# Parquet files (columnar format)
df = pd.read_parquet(data / "large_data.parquet")
```

## Quick Data Inspection

```
# First/last rows
df.head()          # First 5 rows
df.tail(10)        # Last 10 rows

# Shape and info
df.shape           # (rows, columns)
df.info()          # Column types, memory
df.describe()      # Statistics

# Column names and types
df.columns         # Column names
df.dtypes          # Data types

# Sample data
df.sample(5)       # Random 5 rows
```

# Selecting Data

## Column Selection

```
# Single column (returns Series)
df["product"]
df.product      # Dot notation

# Multiple columns (returns DataFrame)
df[["product", "price"]]
```

## Row Selection

```
# By index position
df.iloc[0]        # First row
df.iloc[0:5]       # First 5 rows
df.iloc[[0, 2, 4]] # Specific rows

# By label
df.loc[0]         # Row with index 0
df.loc[0:5]        # Rows 0 through 5
```

## Filtering

```
# Boolean filtering
df[df["price"] > 10]

# Multiple conditions
df[(df["price"] > 10) & (df["quantity"] > 50)]

# Using query (cleaner syntax)
df.query("price > 10 and quantity > 50")

# Check if value in list
df[df["product"].isin(["Apple", "Banana"])]
```

## Combined Selection

```
# Rows and columns together
df.loc[0:5, ["product", "price"]]
df.iloc[0:5, 0:2]
```

# Data Manipulation

## Creating New Columns

```
# Direct assignment
df["revenue"] = df["quantity"] * df["price"]

# Using assign (returns new DataFrame)
df = df.assign(
    revenue=df["quantity"] * df["price"],
    tax=df["price"] * 0.1
)

# Conditional values
df["size"] = df["quantity"].apply(
    lambda x: "large" if x > 100 else "small"
)

# Using np.where
import numpy as np
df["size"] = np.where(
    df["quantity"] > 100, "large", "small"
)
```

## Modifying Data

```
# Rename columns
df = df.rename(columns={
    "old_name": "new_name"
})

# Drop columns
df = df.drop(columns=["unwanted_col"])

# Fill missing values
df["price"] = df["price"].fillna(0)

# Replace values
df["status"] = df["status"].replace({
    "Y": "Yes", "N": "No"
})

# Sort
df = df.sort_values("price", ascending=False)
```

# Aggregations & Grouping

## Basic Aggregations

```
# Single column
df["price"].sum()
df["price"].mean()
df["price"].max()
df["quantity"].nunique() # Unique count

# Multiple aggregations
df["price"].agg(["sum", "mean", "std"])

# Across DataFrame
df[["price", "quantity"]].sum()
```

## GroupBy Operations

```
# Group and aggregate
df.groupby("product")["revenue"].sum()

# Multiple aggregations
df.groupby("product").agg({
    "revenue": ["sum", "mean"],
    "quantity": "sum",
    "customer_id": "nunique"
})

# Group by multiple columns
df.groupby(["product", "region"]).sum()

# Transform (keeps original shape)
df["pct_of_total"] = (
    df["revenue"] /
    df.groupby("product")["revenue"].transform("sum")
)
```

# Writing Data

```
from pathlib import Path

output = Path("output")
output.mkdir(exist_ok=True)

df.to_csv(output / "results.csv", index=False)           # CSV - most common
df.to_excel(output / "report.xlsx", sheet_name="Sales") # Excel
df.to_parquet(output / "data.parquet")                  # Parquet - compressed, fast
df.to_json(output / "data.json", orient="records")      # JSON
```

CSV - Universal format, human-readable, opens in Excel/Google Sheets. Use when sharing data with others.

Parquet - Binary columnar format, compressed, fast for analytics. Use for large datasets and data pipelines.

Excel - Native spreadsheet format with multiple sheets. Use when stakeholders need to open in Excel directly.

JSON - Structured text format. Use for web APIs or when data has nested structures.

# Part 3

Polars: The Modern Alternative

# What is Polars?

A DataFrame library written in Rust, designed as a faster and more memory-efficient alternative to pandas.

## The Problem with Pandas

- Single-threaded - Can't use multiple CPU cores
- High memory usage - Loads everything into RAM
- GIL-bound - Limited by Python's interpreter lock
- Inconsistent API - Many ways to do the same thing

## How Polars Solves It

- Multi-threaded - Uses all cores by default
- Lazy evaluation - Builds a plan first, then optimizes before running
- Rust-powered - Runs outside Python, bypasses GIL
- Consistent API - One clear way to do things

10-100x faster than pandas on large datasets

Lazy evaluation optimizes queries before execution

Handles larger-than-RAM datasets via streaming

# Polars vs Pandas: Side by Side

## Pandas

```
import pandas as pd
from pathlib import Path

df = pd.read_csv(Path("data") / "sales.csv")

# Calculate revenue and filter
df["revenue"] = df["quantity"] * df["price"]
result = df[df["revenue"] > 1000]

# Group and aggregate (named aggregation)
summary = df.groupby("product").agg(
    total_revenue=("revenue", "sum"),
    avg_revenue=("revenue", "mean"),
    total_qty=("quantity", "sum")
)
```

## Polars

```
import polars as pl
from pathlib import Path

df = pl.read_csv(Path("data") / "sales.csv")

# Calculate revenue and filter
revenue = pl.col("quantity") * pl.col("price")
result = df.with_columns(
    revenue.alias("revenue")
).filter(pl.col("revenue") > 1000)

# Group and aggregate
summary = df.group_by("product").agg(
    pl.col("revenue").sum().alias("total_revenue"),
    pl.col("revenue").mean().alias("avg_revenue"),
    pl.col("quantity").sum().alias("total_qty")
)
```

Polars syntax is more verbose but explicit - you always know what's happening

# Expressions vs Contexts

Expressions describe *what* to do. Unlike pandas (which runs each step immediately), Polars collects expressions and optimizes them before execution.

## Expressions (the "what")

```
pl.col("price")           # reference column  
pl.col("price") * 1.1     # computation  
pl.col("x").sum()         # aggregation  
pl.lit("USD")            # literal value
```

Expressions are composable:

```
pl.col("name").str.to_lowercase().str.strip_chars()  
pl.col("price").round(2).cast(pl.Int32).alias("cents")
```

## Contexts (the "where")

```
df.select( ... )          # choose/transform columns  
df.with_columns( ... )    # add new columns  
df.filter( ... )          # filter rows  
df.group_by( ... ).agg( ... ) # aggregate
```

Contexts accept expressions and define what to do with the result.

Pattern: `df.context(expression) → df.select(pl.col("price") * 1.1)`

# Expression API in Action

```
import polars as pl
from pathlib import Path

df = pl.read_csv(Path("data") / "sales_data.csv")

result = df.select(
    pl.col("product"),                                # Select column
    pl.col("price").round(2).alias("rounded_price"),   # Transform + rename
    (pl.col("quantity") * pl.col("price")).alias("revenue"), # Compute
    pl.col("date").str.to_date().alias("parsed_date"),  # Parse dates
    pl.lit("USD").alias("currency")                   # Add constant
)
```

`select()` - Choose columns

```
df.select(
    pl.col("name"),
    pl.col("price") * 2
)
```

`with_columns()` - Add columns

```
df.with_columns(
    (pl.col("a") + pl.col("b"))
    .alias("sum")
)
```

`filter()` - Filter rows

```
df.filter(
    pl.col("price") > 100
)
```

# Lazy Evaluation

For large data, Polars can build a query plan first and optimize before running.

## Eager (runs immediately)

```
df = pl.read_csv(data)
result = df.filter(pl.col("x") > 0)
```

Each operation executes as it's called. Simple, but loads all data into memory.

## Lazy (builds plan first)

```
lf = pl.scan_csv(data)
query = lf.filter(pl.col("x") > 0)
    .select(["a", "b"])
result = query.collect() # runs here
```

Builds a plan, optimizes it, then executes. Only loads needed columns.

## Why Lazy Wins for Large Data

### Query Optimization

Reorders operations, removes unused work

### Memory Efficiency

Filters before loading, streams large files

### Smarter Parallelism

Parallelizes across the entire query plan

# Performance Comparison

Task: Calculate revenue and sort by store (3 columns, 1M/10M/100M rows)

## Pandas

```
df_pd = pd.DataFrame(data)
df_pd["revenue"] = df_pd["quantity"] * df_pd["price"]
result = df_pd.sort_values(["store_id", "revenue"],
                           ascending=[True, False])
```

## Polars

```
df_pl = pl.DataFrame(data)
rev = pl.col("quantity") * pl.col("price")
result = df_pl.with_columns(rev.alias("revenue")).sort(
    ["store_id", "revenue"], descending=[False, True])
```

Rows	Pandas	Polars	Speedup
1 million	0.25s	0.03s	9x
10 million	4.1s	0.38s	11x
100 million	75s	4.8s	16x

The speedup increases with data size - Polars shines on large datasets

# Common Operations Cheat Sheet

## Reading/Writing

```
from pathlib import Path
data = Path("data")

# Read
df = pl.read_csv(data / "file.csv")
df = pl.read_parquet(data / "file.parquet")
lf = pl.scan_csv(data / "file.csv") # Lazy

# Write
df.write_csv(data / "out.csv")
df.write_parquet(data / "out.parquet")
```

## Filtering

```
df.filter(pl.col("x") > 10)
df.filter(
    (pl.col("x") > 10) & (pl.col("y") < 5)
)
df.filter(pl.col("name").is_in(["A", "B"]))
```

## Aggregations

```
df.group_by("category").agg(
    pl.col("value").sum().alias("total"),
    pl.col("value").mean().alias("average"),
    pl.len().alias("count"),
    pl.col("id").n_unique().alias("unique_ids")
)
```

## Sorting & Limiting

```
df.sort("price", descending=True)
df.sort(["cat", "price"], descending=[True, False])
df.head(10)
df.tail(5)
```

# When to Use Polars vs Pandas

## Use Polars When

- Working with **large datasets** (>100MB)
- Performance is critical
- Memory is constrained
- Building **data pipelines**
- Starting a new project
- Need **lazy evaluation**

*Time to process 111 million rows: 5.2 seconds*

## Use Pandas When

- Working with **small datasets**
- Need specific **library integrations**
- Team familiarity matters
- Using **legacy code**
- Need **extensive ecosystem** (statsmodels, sklearn)
- Quick exploratory analysis

Tip: Convert between them with Polars: `pl_df.to_pandas()` and `pl.from_pandas(pd_df)`

# Part 4

DuckDB: SQL Meets DataFrames

# What is DuckDB?

An in-process SQL database designed for fast analytical queries. Think "SQLite, but for data analysis."

## How It Works

- Runs inside your Python process (no server)
- Columnar storage (reads only needed columns)
- Vectorized execution (operates on entire columns, not row-by-row like traditional databases)
- Queries files directly (CSV, Parquet, JSON) without reading into memory.

## Why It Exists

Traditional databases are built for transactions (OLTP). DuckDB is built for analytics (OLAP) - aggregations, joins, and scans on large tables.

## Key Features

- SQL interface - familiar syntax
- Fast - optimized for analytics
- Reads anything - CSV, Parquet, JSON, pandas, polars
- Connects anywhere - PostgreSQL, MySQL, SQLite

```
import duckdb

result = duckdb.sql("""
    SELECT product, SUM(sales)
    FROM 'sales_data.csv'
    GROUP BY product
""")
```

# DuckDB Basics

```
import duckdb
con = duckdb.connect()                                     # In-memory (default)
con = duckdb.connect("data/my_data.duckdb")                # Persistent database
```

## Query Files Directly

```
# CSV
result = con.execute("""
    SELECT * FROM read_csv_auto('data/sales.csv')
    WHERE price > 100
""").df()

# Parquet (supports globs)
result = con.execute("""
    SELECT * FROM 'data/*.parquet'
""").df()

# JSON
result = con.execute("""
    SELECT * FROM read_json_auto('data/records.json')
""").df()
```

## Query DataFrames

```
import pandas as pd
import polars as pl

df_pandas = pd.read_csv("data/sales.csv")
df_polars = pl.read_csv("data/sales.csv")

# Query pandas with SQL
result = con.execute("""
    SELECT product, AVG(price)
    FROM df_pandas
    GROUP BY product
""").df()

# Query polars, return as polars
result = con.execute("""
    SELECT * FROM df_polars WHERE qty > 10
""").pl()
```

# SQL for Data Analysis

```
import duckdb
con = duckdb.connect()

result = con.execute("""
    WITH monthly_sales AS (
        SELECT date_trunc('month', date) as month, product,
               SUM(quantity * price) as revenue, COUNT(*) as transactions
        FROM read_csv_auto('data/sales_data.csv')
        GROUP BY date_trunc('month', date), product
    )
    SELECT *, 
           revenue / SUM(revenue) OVER (PARTITION BY month) as pct_of_month
    FROM monthly_sales
    ORDER BY month, revenue DESC
""").df()
```

Standard SQL features work:

- CTEs - `WITH` clauses for multi-step queries
- Window functions - `OVER`, `PARTITION BY`
- Date functions - `date_trunc`, `extract`

Query external databases directly:

```
ATTACH 'postgres://user:pass@host/db' AS pg;
SELECT * FROM pg.sales WHERE revenue > 1000;
```

# When to Use DuckDB

## Perfect For

- Ad-hoc analysis on files
- SQL-first workflows
- Large file processing
- Joining multiple data sources
- Complex aggregations
- Quick data exploration

## Use With Polars

### When

- Need programmatic transformations
- Building data pipelines
- Streaming processing
- Need Python ecosystem
- Complex business logic

## Avoid When

- Need OLTP (transactions)
- High concurrency writes
- Need a server database
- Real-time streaming
- Simple operations on small data

**Rule of thumb:** If you're working with files larger than your RAM, try DuckDB first

# DuckDB + Polars Integration

Best of both worlds: SQL queries with Polars performance

```
import polars as pl
import duckdb

df = pl.read_parquet("data/large_data.parquet") # Read with Polars
con = duckdb.connect()

# Use DuckDB for complex SQL on Polars DataFrame
result = con.execute("""
    SELECT store_id, date_trunc('week', timestamp) as week,
           SUM(quantity * price) as weekly_revenue,
           COUNT(DISTINCT customer_id) as unique_customers
    FROM df
    GROUP BY store_id, date_trunc('week', timestamp)
    HAVING SUM(quantity * price) > 10000
""").pl() # Return as Polars DataFrame

# Continue processing in Polars
final = result.with_columns(
    (pl.col("weekly_revenue") / pl.col("unique_customers")).alias("rev_per_customer")
)
```

**Workflow:** Load with Polars -> Query with DuckDB SQL -> Process result in Polars

# Performance Comparison: Polars vs DuckDB

Task: Calculate revenue and sort by store (3 columns, 1M/10M/100M rows)

## Polars

```
df_pl = pl.DataFrame(data)
rev = pl.col("quantity") * pl.col("price")
result = df_pl.with_columns(
    rev.alias("revenue")
).sort(["store_id", "revenue"],
       descending=[False, True])
```

## DuckDB

```
result = con.execute("""
    SELECT *, quantity * price as revenue
    FROM data
    ORDER BY store_id ASC, revenue DESC
""").pl()
```

Rows	Polars	DuckDB	Notes
1M	0.03s	0.45s	Polars 16x faster
10M	0.37s	0.64s	Polars 1.7x faster
100M	4.9s	9.3s	Polars 1.9x faster

Choose based on: SQL familiarity -> DuckDB | Programmatic pipelines -> Polars

# Summary: Choosing the Right Tool

Tool	Best For	Syntax	Speed
pathlib	File/directory operations	Object-oriented	N/A
Pandas	Small data, ecosystem compatibility	DataFrame API	Baseline
Polars	Large data, pipelines, performance	Expression API	9-16x faster
DuckDB	SQL analysis, file queries, joins	SQL	5-10x faster

## Recommended Stack

1. pathlib for file operations
2. Polars as primary DataFrame library
3. DuckDB for complex SQL queries
4. Pandas only when needed for compatibility

## Quick Decision Guide

- Small data + quick analysis -> Pandas
- Large data + pipelines -> Polars
- Data larger than RAM + multiple files -> DuckDB
- Mix and match as needed!

# Hands-on Exercise

## Analyze Sales Data with All Three Libraries

Using `data/sales_data.csv`:

1. Pandas: Read data, calculate total revenue per product
2. Polars: Same analysis - compare the syntax
3. DuckDB: Find the best sales day using SQL

```
# Start here
from pathlib import Path
import pandas as pd
import polars as pl
import duckdb

data_path = Path("data") / "sales_data.csv"

# Your code ...
```

Bonus: Time each approach and compare performance!

