Week 4:
Working with Data in Python

Dr Christian Engels
ce50@st-andrews.ac.uk

FI5699 Dissertation Module

Department of Finance
University of St Andrews Business School

Introduction

What We Will Cover Today

This is a hands-on session. You are welcome to have your laptop open and follow along.

1. **Polars Data Structures** — DataFrames, Series, and data types
2. **Reading & Writing Data** — CSV and Excel I/O with Polars
3. **Expressions & Contexts** — the Polars way of querying data
4. **Working with Expressions** — arithmetic, comparisons, conditionals
5. **Working with Stock Returns** — downloading Yahoo Finance data
6. **Combining DataFrames** — joins and concatenation
7. **Reshaping Data** — pivot and unpivot
8. **Visualising Data** — building plots with `plotnine`

By the end of today, you will be able to load real financial data, query, transform, reshape, and visualise it in Python.

Table of Contents

Recap: What You Built in Week 2

Concept	What you learned
Positron	Console, Terminal, and Editor
`uv`	`uv init`, `uv add` to manage packages
Variables	`price = 142.50`, types (`int`, `float`, `str`, `bool`)
Lists & Dicts	`["AAPL", "MSFT"]` and `{"ticker": "AAPL"}`
Loops	`for` and `while` — automating repetitive tasks

Today we move from Python basics to **working with real data** using `polars` and `plotnine`.

Companion Code: week-4-code

Every concept in this lecture has a runnable Python script in the week-4-code folder.

Script	Covers
01_data_structures.py	DataFrames, Series, types, nulls
02_reading_writing_data.py	CSV & Excel I/O
03_expressions_contexts.py	select, with_columns, filter, group_by
04_working_with_expressions.py	Arithmetic, conditionals, casting
05_real_stock_data.py	Yahoo Finance data & returns
06_combining_dataframes.py	Joins & concatenation
07_reshaping_data.py	Pivot & unpivot
08_visualising_data.py	plotnine charts

Open week-4-code in Positron and run each script to see the concepts in the .py files.

Setting Up: Two Steps

1. **Open the folder in Positron**

   `File` → `Open Folder…` → select `week-4-code`

2. **Install dependencies**

   Open a Terminal in Positron and run:

   ```
   uv sync
   ```

   This reads `pyproject.toml` and installs Polars, plotnine, tidyfinance, and all other packages into a virtual environment.

3. **Select the interpreter**

   Click `Select Interpreter` in the top-right corner of Positron and choose the `.venv` Python that `uv sync` just created. This ensures Positron uses the correct packages when you run scripts.

Once the interpreter is set, you are ready to go.
Open any script and press `Ctrl+Shift+Enter` to run it.

From Lists to DataFrames

In Week 2 you used lists and dictionaries to organise small amounts of data. But your dissertation will involve thousands of rows — stock prices, returns, firm characteristics.

	Lists / Dicts	Polars
Data size	A handful of values	Millions of rows
Operations	One value at a time	Entire columns at once
Speed	Pure Python (slow)	Written in Rust (fast)
Structure	Ad hoc	Typed columns, schemas

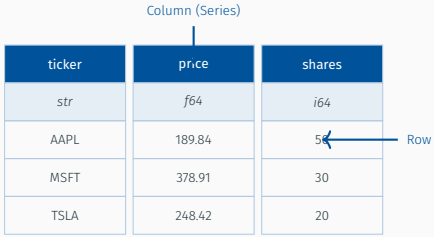
Think of Polars as a **supercharged spreadsheet** that you control with code.

Polars Data Structures

A DataFrame Is a Table with Typed Columns

A DataFrame is the core Polars data structure:

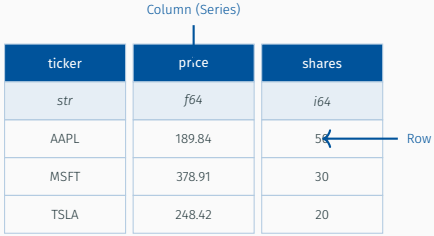
- A table of rows and columns — like a spreadsheet



A DataFrame Is a Table with Typed Columns

A DataFrame is the core Polars data structure:

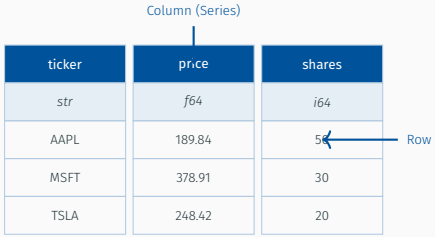
- A table of rows and columns — like a spreadsheet
- Each column has a **name** and a **data type**



A DataFrame Is a Table with Typed Columns

A DataFrame is the core Polars data structure:

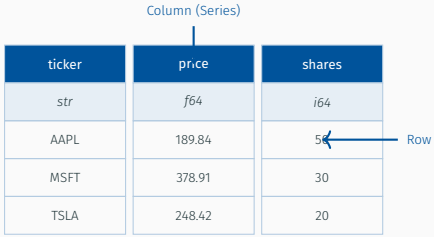
- A table of rows and columns — like a spreadsheet
- Each column has a **name** and a **data type**
- Each column is internally a Series



A DataFrame Is a Table with Typed Columns

A DataFrame is the core Polars data structure:

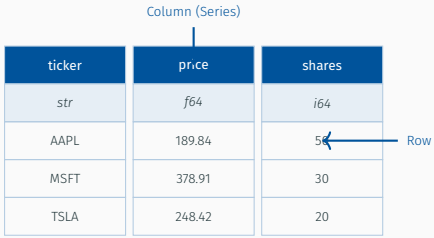
- A table of rows and columns — like a spreadsheet
- Each column has a **name** and a **data type**
- Each column is internally a Series
- All values in a column share the same type



A DataFrame Is a Table with Typed Columns

A DataFrame is the core Polars data structure:

- A table of rows and columns — like a spreadsheet
- Each column has a **name** and a **data type**
- Each column is internally a Series
- All values in a column share the same type
- Different columns can have different types



Creating Your First Polars DataFrame

```python
import polars as pl

stocks = pl.DataFrame({
    "ticker": ["AAPL", "MSFT", "TSLA", "AMZN", "GOOGL"],
    "price":  [189.84, 378.91, 248.42, 178.25, 141.80],
    "shares": [50, 30, 20, 40, 35],
    "sector": ["Tech", "Tech", "Auto", "Retail", "Tech"],
})
print(stocks)
```

```
shape: (5, 4)
+--------+--------+--------+--------+
| ticker | price  | shares | sector |
| ---    | ---    | ---    | ---    |
| str    | f64    | i64    | str    |
+========+========+========+========+
| AAPL   | 189.84 | 50     | Tech   |
| MSFT   | 378.91 | 30     | Tech   |
| TSLA   | 248.42 | 20     | Auto   |
| AMZN   | 178.25 | 40     | Retail |
| GOOGL  | 141.8  | 35     | Tech   |
+--------+--------+--------+--------+
```

Every Column Has a Data Type

Polars infers types automatically, but it helps to know the common ones:

Type	Polars name	Example values
Whole number	`Int64`	Shares: 50, 30, 20
Decimal number	`Float64`	Price: 189.84, 378.91
Text	`String`	Ticker: “AAPL”, “MSFT”
True / False	`Boolean`	Is profitable: true, false
Date	`Date`	2024-01-15
Date + time	`Datetime`	2024-01-15 09:30:00
Missing value	`Null`	Unknown or not applicable

You can see each column’s type in the `---` row of the printed output.

A Series Is a Single Column

A Series is a one-dimensional array — one column of a DataFrame:

```
prices = pl.Series("price", [189.84, 378.91, 248.42])
print(prices)
print(f"Type:  {prices.dtype}")
print(f"Mean:  {prices.mean():.2f}")
```

```
shape: (3,)
Series: 'price' [f64]
[
    189.84
    378.91
    248.42
]
Type:  Float64
Mean:  272.39
```

You rarely create Series directly — they arise when you extract a column from a DataFrame.

Five Ways to Look at Your Data

Method	What it shows	Example
`.head(n)`	First n rows (default 5)	`stocks.head(3)`
`.tail(n)`	Last n rows	`stocks.tail(3)`
`.shape`	(rows, columns) tuple	`stocks.shape`
`.columns`	List of column names	`stocks.columns`
`.schema`	Column names and types	`stocks.schema`

```
print(stocks.shape)
print(stocks.columns)
```

```
(5, 4)
['ticker', 'price', 'shares', 'sector']
```

Summary Statistics with `describe()`

```
print(stocks.describe())
```

```
shape: (9, 5)
+-------------+--------+------------+-----------+--------+
| statistic   | ticker | price      | shares    | sector |
| ---         | ---    | ---        | ---       | ---    |
| str         | str    | f64        | f64       | str    |
+=============+========+============+===========+========+
| count       | 5      | 5.0        | 5.0       | 5      |
| null_count  | 0      | 0.0        | 0.0       | 0      |
| mean        | null   | 227.444    | 35.0      | null   |
| std         | null   | 92.939425  | 11.18034  | null   |
| min         | AAPL   | 141.8      | 20.0      | Auto   |
| 25%         | null   | 178.25     | 30.0      | null   |
| 50%         | null   | 189.84     | 35.0      | null   |
| 75%         | null   | 248.42     | 40.0      | null   |
| max         | TSLA   | 378.91     | 50.0      | Tech   |
+-------------+--------+------------+-----------+--------+
```

One command gives you count, mean, standard deviation, min, max, and quartiles — the same summary statistics you would compute in Excel or Stata.

Null Means Missing

Polars uses `null` for missing data. Every data type supports it — unlike Excel's blank cells, a `null` is explicit and typed.

Why it matters:

- Real-world data has gaps (no price on a holiday, missing filings)

`null` vs NaN:

When you see **`null`** in your output, it means the data is **genuinely missing**, not zero.

Null Means Missing

Polars uses `null` for missing data. Every data type supports it — unlike Excel's blank cells, a `null` is explicit and typed.

Why it matters:

- Real-world data has gaps (no price on a holiday, missing filings)
- Polars propagates `null` through calculations: `null + 5 = null`

`null` vs NaN:

When you see **`null`** in your output, it means the data is **genuinely missing**, not zero.

Null Means Missing

Polars uses `null` for missing data. Every data type supports it — unlike Excel's blank cells, a `null` is explicit and typed.

Why it matters:

- Real-world data has gaps (no price on a holiday, missing filings)
- Polars propagates `null` through calculations: `null + 5 = null`
- You can detect them with `.is_null()` or fill them with `.fill_null()`

`null` vs NaN:

When you see `null` in your output, it means the data is **genuinely missing**, not zero.

Null Means Missing

Polars uses `null` for missing data. Every data type supports it — unlike Excel's blank cells, a `null` is explicit and typed.

Why it matters:

- Real-world data has gaps (no price on a holiday, missing filings)
- Polars propagates `null` through calculations: `null + 5 = null`
- You can detect them with `.is_null()` or fill them with `.fill_null()`

`null` vs NaN:

- `null` = "no value exists"

When you see `null` in your output, it means the data is **genuinely missing**, not zero.

Null Means Missing

Polars uses `null` for missing data. Every data type supports it — unlike Excel's blank cells, a `null` is explicit and typed.

Why it matters:

- Real-world data has gaps (no price on a holiday, missing filings)
- Polars propagates `null` through calculations: `null + 5 = null`
- You can detect them with `.is_null()` or fill them with `.fill_null()`

`null` vs `NaN`:

- `null` = "no value exists"
- `NaN` = "a float calculation failed" (e.g. 0/0)

When you see `null` in your output, it means the data is **genuinely missing**, not zero.

Null Means Missing

Polars uses `null` for missing data. Every data type supports it — unlike Excel's blank cells, a `null` is explicit and typed.

Why it matters:

- Real-world data has gaps (no price on a holiday, missing filings)
- Polars propagates `null` through calculations: `null + 5 = null`
- You can detect them with `.is_null()` or fill them with `.fill_null()`

`null` vs `NaN`:

- `null` = "no value exists"
- `NaN` = "a float calculation failed" (e.g. 0/0)
- In Polars, use `null` for missing data

When you see `null` in your output, it means the data is **genuinely missing**, not zero.

Null Means Missing

Polars uses `null` for missing data. Every data type supports it — unlike Excel's blank cells, a `null` is explicit and typed.

Why it matters:

- Real-world data has gaps (no price on a holiday, missing filings)
- Polars propagates `null` through calculations: `null + 5 = null`
- You can detect them with `.is_null()` or fill them with `.fill_null()`

`null` vs `NaN`:

- `null` = "no value exists"
- `NaN` = "a float calculation failed" (e.g. 0/0)
- In Polars, use `null` for missing data
- `NaN` is only for `Float64` columns

When you see `null` in your output, it means the data is **genuinely missing**, not zero.

Exercise 1: Build and Inspect a DataFrame

Try It Yourself (5 minutes)

1. Create a new file `week4.py` and add `import polars as pl` at the top
2. Create a DataFrame called `portfolio` with columns:
   - `"ticker"`: at least 4 stocks you know
   - `"price"`: a price for each (make them up)
   - `"shares"`: how many shares of each
3. Print the DataFrame
4. Print `portfolio.shape` and `portfolio.describe()`
5. **Bonus:** Create a `pl.Series` of monthly returns and compute its `.mean()`

Reading and Writing Data

Reading a CSV File

Most financial data starts life as a CSV file. Polars reads them with `pl.read_csv()`:

```
import polars as pl

df = pl.read_csv("stock_prices.csv")
print(df.head())
```

Function	What it does
`pl.read_csv(path)`	Read an entire CSV into a DataFrame immediately
`pl.scan_csv(path)`	Return a `LazyFrame` — delays reading until you call `.collect()`

`scan_csv` is faster for large files because Polars can optimise the query before reading the data.

Useful `read_csv` Parameters

```python
df = pl.read_csv(
    "stock_prices.csv",
    separator=",",           # default; use "\t" for TSV
    has_header=True,         # first row is column names
    skip_rows=0,             # skip N rows at the top
    n_rows=1000,             # read only the first 1000 rows
    null_values=["NA", ""], # treat these strings as null
    try_parse_dates=True,    # auto-detect date columns
)
```

`try_parse_dates=True` is especially useful for financial data — it converts date strings into proper `Date` columns automatically.

Writing a CSV File

After transforming your data, save it with `.write_csv()`:

```
stocks = pl.DataFrame({
    "ticker": ["AAPL", "MSFT", "TSLA"],
    "price":  [189.84, 378.91, 248.42],
    "shares": [50, 30, 20],
})

stocks.write_csv("my_portfolio.csv")
```

This writes a standard comma-separated file with a header row. You can customise:

Parameter	Effect
`separator="\t"`	Write a tab-separated file instead
`include_header=False`	Omit the header row

Lazy Reading with `scan_csv`

For large files, `pl.scan_csv()` returns a LazyFrame instead of loading everything:

```python
lf = pl.scan_csv("large_dataset.csv")

# Build a query without reading data yet
result = (
    lf
    .filter(pl.col("ticker") == "AAPL")
    .select("date", "close")
    .collect()  # NOW it reads and executes
)
```

`read_csv`

- Reads entire file immediately

`scan_csv`

Lazy Reading with `scan_csv`

For large files, `pl.scan_csv()` returns a LazyFrame instead of loading everything:

```python
lf = pl.scan_csv("large_dataset.csv")

# Build a query without reading data yet
result = (
    lf
    .filter(pl.col("ticker") == "AAPL")
    .select("date", "close")
    .collect()  # NOW it reads and executes
)
```

`read_csv`

- Reads entire file immediately
- Returns a `DataFrame`

`scan_csv`

Lazy Reading with `scan_csv`

For large files, `pl.scan_csv()` returns a LazyFrame instead of loading everything:

```python
lf = pl.scan_csv("large_dataset.csv")

# Build a query without reading data yet
result = (
    lf
    .filter(pl.col("ticker") == "AAPL")
    .select("date", "close")
    .collect()  # NOW it reads and executes
)
```

`read_csv`

- Reads entire file immediately
- Returns a `DataFrame`
- Simple and direct

`scan_csv`

Lazy Reading with `scan_csv`

For large files, `pl.scan_csv()` returns a LazyFrame instead of loading everything:

```python
lf = pl.scan_csv("large_dataset.csv")

# Build a query without reading data yet
result = (
    lf
    .filter(pl.col("ticker") == "AAPL")
    .select("date", "close")
    .collect()  # NOW it reads and executes
)
```

`read_csv`

- Reads entire file immediately
- Returns a `DataFrame`
- Simple and direct
- Fine for small–medium files

`scan_csv`

Lazy Reading with `scan_csv`

For large files, `pl.scan_csv()` returns a LazyFrame instead of loading everything:

```python
lf = pl.scan_csv("large_dataset.csv")

# Build a query without reading data yet
result = (
    lf
    .filter(pl.col("ticker") == "AAPL")
    .select("date", "close")
    .collect()  # NOW it reads and executes
)
```

`read_csv`

- Reads entire file immediately
- Returns a `DataFrame`
- Simple and direct
- Fine for small–medium files

`scan_csv`

- Defers reading until `.collect()`

Lazy Reading with `scan_csv`

For large files, `pl.scan_csv()` returns a LazyFrame instead of loading everything:

```python
lf = pl.scan_csv("large_dataset.csv")

# Build a query without reading data yet
result = (
    lf
    .filter(pl.col("ticker") == "AAPL")
    .select("date", "close")
    .collect()  # NOW it reads and executes
)
```

`read_csv`

- Reads entire file immediately
- Returns a `DataFrame`
- Simple and direct
- Fine for small–medium files

`scan_csv`

- Defers reading until `.collect()`
- Returns a `LazyFrame`

Lazy Reading with `scan_csv`

For large files, `pl.scan_csv()` returns a LazyFrame instead of loading everything:

```python
lf = pl.scan_csv("large_dataset.csv")

# Build a query without reading data yet
result = (
    lf
    .filter(pl.col("ticker") == "AAPL")
    .select("date", "close")
    .collect()  # NOW it reads and executes
)
```

`read_csv`

- Reads entire file immediately
- Returns a `DataFrame`
- Simple and direct
- Fine for small–medium files

`scan_csv`

- Defers reading until `.collect()`
- Returns a `LazyFrame`
- Polars optimises the query first

Lazy Reading with `scan_csv`

For large files, `pl.scan_csv()` returns a LazyFrame instead of loading everything:

```python
lf = pl.scan_csv("large_dataset.csv")

# Build a query without reading data yet
result = (
    lf
    .filter(pl.col("ticker") == "AAPL")
    .select("date", "close")
    .collect()  # NOW it reads and executes
)
```

`read_csv`

- Reads entire file immediately
- Returns a `DataFrame`
- Simple and direct
- Fine for small–medium files

`scan_csv`

- Defers reading until `.collect()`
- Returns a `LazyFrame`
- Polars optimises the query first
- Better for large files

Reading Excel Files

Polars can also read `.xlsx` files directly:

```
df = pl.read_excel("portfolio.xlsx")

# Read a specific sheet
df = pl.read_excel("portfolio.xlsx", sheet_name="Returns")
```

Required dependency: install one of these Excel engines:

```
$ uv add fastexcel    # fastest (recommended, default)
$ uv add openpyxl     # slower but handles tricky files
```

Without a `sheet_name`, Polars reads the **first sheet** by default.

Writing Excel Files

Save a DataFrame to Excel with `.write_excel()`:

```
stocks.write_excel("my_portfolio.xlsx")

# Specify a worksheet name
stocks.write_excel("my_portfolio.xlsx", worksheet="Holdings")
```

Required dependency:

```
$ uv add xlsxwriter
```

Performance tip: CSV and Parquet are much faster than Excel for large datasets. Use Excel when you need to share with non-programmers; use CSV or Parquet for your own analysis pipelines.

Exercise 2: Reading and Writing Files

Try It Yourself (5 minutes)

1. Create a DataFrame with at least 4 rows of stock data (ticker, price, shares)
2. Write it to a CSV file with `.write_csv("portfolio.csv")`
3. Read it back with `pl.read_csv("portfolio.csv")` and print the result — does it match?
4. **Bonus:** Write to Excel with `.write_excel("portfolio.xlsx")` and read it back
5. **Bonus:** Try `pl.scan_csv("portfolio.csv").collect()` — same result?

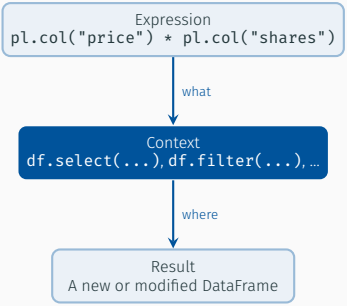
Expressions and Contexts

Polars Separates *What* to Compute from *Where*

An expression is a recipe for a computation. It describes **what** to do, but does nothing until you place it in a context.

- Expression: “multiply price by shares”

This separation is what makes Polars fast and flexible.

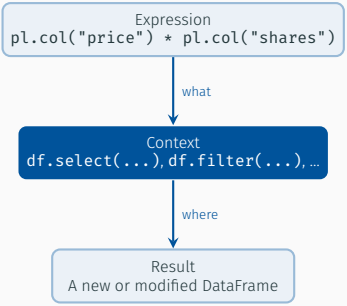


Polars Separates *What* to Compute from *Where*

An expression is a recipe for a computation. It describes **what** to do, but does nothing until you place it in a context.

- Expression: “multiply price by shares”
- Context: “add it as a new column” or “replace all columns” or “use it to filter rows”

This separation is what makes Polars fast and flexible.



Context 1: `select` — Pick and Transform Columns

`select` returns **only** the columns you ask for:

```python
result = stocks.select(
    "ticker",
    (pl.col("price") * pl.col("shares")).alias("position_value"),
)
print(result)
```

```
shape: (5, 2)
+--------+----------------+
| ticker | position_value |
| ---    | ---            |
| str    | f64            |
+========+================+
| AAPL   | 9492.0         |
| MSFT   | 11367.3        |
| TSLA   | 4968.4         |
| AMZN   | 7130.0         |
| GOOGL  | 4963.0         |
+--------+----------------+
```

Context 2: `with_columns` — Add New Columns

`with_columns` keeps **all existing** columns and adds new ones:

```
result = stocks.with_columns(
    position_value = pl.col("price") * pl.col("shares"),
)
print(result)
```

```
shape: (5, 5)
+--------+--------+--------+--------+----------------+
| ticker | price  | shares | sector | position_value |
| ---    | ---    | ---    | ---    | ---            |
| str    | f64    | i64    | str    | f64            |
+========+========+========+========+================+
| AAPL   | 189.84 | 50     | Tech   | 9492.0         |
| MSFT   | 378.91 | 30     | Tech   | 11367.3        |
| TSLA   | 248.42 | 20     | Auto   | 4968.4         |
| AMZN   | 178.25 | 40     | Retail | 7130.0         |
| GOOGL  | 141.8  | 35     | Tech   | 4963.0         |
+--------+--------+--------+--------+----------------+
```

Context 3: `filter` — Keep Matching Rows

`filter` keeps only the rows where a condition is `True`:

```
result = stocks.filter(pl.col("price") > 200)
print(result)
```

```
shape: (2, 4)
+--------+--------+--------+--------+
| ticker | price  | shares | sector |
| ---    | ---    | ---    | ---    |
| str    | f64    | i64    | str    |
+========+========+========+========+
| MSFT   | 378.91 | 30     | Tech   |
| TSLA   | 248.42 | 20     | Auto   |
+--------+--------+--------+--------+
```

Think of `filter` as Excel's AutoFilter: only rows that match your condition survive.

Context 4: `group_by` — Aggregate by Group

`group_by` splits the data into groups and applies aggregation functions:

```
result = stocks.group_by("sector").agg(
    pl.col("price").mean().alias("avg_price"),
    pl.len().alias("count"),
)
print(result.sort("sector"))
```

```
shape: (3, 3)
+--------+-----------+-------+
| sector | avg_price | count |
| ---    | ---       | ---   |
| str    | f64       | u32   |
+========+===========+=======+
| Auto   | 248.42    | 1     |
| Retail | 178.25    | 1     |
| Tech   | 236.85    | 3     |
+--------+-----------+-------+
```

Like a pivot table in Excel: group rows by a category, then summarise each group.

The Four Contexts at a Glance

Context	What it does	Excel analogy
`select`	Pick and transform columns	Choosing which columns to display
`with_columns`	Add or overwrite columns	Adding a formula column next to existing data
`filter`	Keep matching rows	AutoFilter on a column
`group_by`	Aggregate by group	Pivot table: group and summarise

Master these four and you can answer almost any data question. Everything else builds on them.

Exercise 3: Expressions and Contexts

Try It Yourself (5 minutes)
Using your `portfolio` DataFrame from Exercise 1:

1. Use `.select()` to show only the ticker and a new column `value` (price × shares)
2. Use `.with_columns()` to add a `value` column while keeping all originals
3. Use `.filter()` to show only stocks with price above 100
4. **Bonus:** Use `.group_by()` if you have a sector column — compute the mean price per sector

`.alias("name")` renames a computed column. Keyword syntax (`value = ...`) also works in `with_columns` and `select`.

Working with Expressions

Column Arithmetic Works Element-Wise

Polars applies operations to **entire columns at once** — no loops needed:

```python
result = stocks.select(
    "ticker",
    (pl.col("price") + 10).alias("price_adj"),     # scalar
    (pl.col("price") * pl.col("shares")).alias("value"),  # column
)
print(result)
```

```
shape: (5, 3)
+--------+-----------+---------+
| ticker | price_adj | value   |
| ---    | ---       | ---     |
| str    | f64       | f64     |
+========+===========+=========+
| AAPL   | 199.84    | 9492.0  |
| MSFT   | 388.91    | 11367.3 |
| TSLA   | 258.42    | 4968.4  |
| AMZN   | 188.25    | 7130.0  |
| GOOGL  | 151.8     | 4963.0  |
+--------+-----------+---------+
```

Comparisons Return Boolean Columns

```python
result = stocks.select(
    "ticker", "price",
    is_expensive = pl.col("price") > 200,
    is_tech = pl.col("sector") == "Tech",
)
print(result)
```

```
shape: (5, 4)
+--------+--------+--------------+---------+
| ticker | price  | is_expensive | is_tech |
| ---    | ---    | ---          | ---     |
| str    | f64    | bool         | bool    |
+========+========+==============+=========+
| AAPL   | 189.84 | false        | true    |
| MSFT   | 378.91 | true         | true    |
| TSLA   | 248.42 | true         | false   |
| AMZN   | 178.25 | false        | false   |
| GOOGL  | 141.8  | false        | true    |
+--------+--------+--------------+---------+
```

Combining Conditions with & and |

Use & (and) and | (or) to combine Boolean expressions. **Parentheses are required.**

```python
# Tech stocks priced above 150
result = stocks.filter(
    (pl.col("price") > 150) & (pl.col("sector") == "Tech")
)
print(result)
```

```
shape: (2, 4)
+--------+--------+--------+--------+
| ticker | price  | shares | sector |
| ---    | ---    | ---    | ---    |
| str    | f64    | i64    | str    |
+========+========+========+========+
| AAPL   | 189.84 | 50     | Tech   |
| MSFT   | 378.91 | 30     | Tech   |
+--------+--------+--------+--------+
```

GOOGL (141.80) is Tech but not above 150, so it is excluded. Both conditions must be true.

Conditionals: `when` / `then` / `otherwise`

Create new values based on conditions — like Excel's `IF()` but chainable:

```
result = stocks.select(
    "ticker", "price",
    size = pl.when(pl.col("price") > 300).then(pl.lit("Large"))
             .when(pl.col("price") > 200).then(pl.lit("Mid"))
             .otherwise(pl.lit("Small"))
)
print(result)
```

```
shape: (5, 3)
+--------+--------+-------+
| ticker | price  | size  |
| ---    | ---    | ---   |
| str    | f64    | str   |
+========+========+=======+
| AAPL   | 189.84 | Small |
| MSFT   | 378.91 | Large |
| TSLA   | 248.42 | Mid   |
| AMZN   | 178.25 | Small |
| GOOGL  | 141.8  | Small |
+--------+--------+-------+
```

Casting: Changing Data Types

Use `.cast()` to convert a column's type:

```python
result = stocks.select(
    "ticker",
    pl.col("price").cast(pl.Int64).alias("price_int"),
    pl.col("shares").cast(pl.Float64).alias("shares_f"),
)
print(result)
```

```
shape: (5, 3)
+--------+-----------+----------+
| ticker | price_int | shares_f |
| ---    | ---       | ---      |
| str    | i64       | f64      |
+========+===========+==========+
| AAPL   | 189       | 50.0     |
| MSFT   | 378       | 30.0     |
| TSLA   | 248       | 20.0     |
| AMZN   | 178       | 40.0     |
| GOOGL  | 141       | 35.0     |
+--------+-----------+----------+
```

Note: Floats are truncated (not rounded) when cast to integers. 189.84 becomes 189.

Expression Expansion: Many Columns at Once

Instead of writing the same expression for each column, pass **multiple names** to `pl.col()`:

```python
# Apply .mean() to both price and shares in one expression
result = stocks.select(
    pl.col("price", "shares").mean(),
)
print(result)
```

```
shape: (1, 2)
+---------+--------+
| price   | shares |
| ---     | ---    |
| f64     | f64    |
+==========+========+
| 227.444 | 35.0   |
+---------+--------+
```

One expression, two columns. Polars “expands” the expression for each column name automatically.

Naming Results with `alias`, `prefix`, and `suffix`

```python
# .alias() renames a single result
pl.col("price").mean().alias("avg_price")

# .name.prefix() / .name.suffix() for batches
result = stocks.select(
    pl.col("price", "shares").mean().name.prefix("avg_"),
)
print(result)
```

```
shape: (1, 2)
+-----------+------------+
| avg_price | avg_shares |
| ---       | ---        |
| f64       | f64        |
+============+============+
| 227.444   | 35.0       |
+-----------+------------+
```

Use `.alias()` for one column. Use `.name.prefix()` or `.name.suffix()` when expanding across several.

Exercise 4: Expressions

Try It Yourself (5 minutes)
Using the `stocks` DataFrame:

1. Add a Boolean column `is_large` that is `True` when `price > 200`
2. Create a column `weight` as each stock's value divided by the total portfolio value:
   - Hint: `pl.col("value")/ pl.col("value").sum()`
3. Use `when/then/otherwise` to label stocks as "Buy" (price < 200) or "Hold"
4. **Bonus:** Use expression expansion to compute `.max()` of both price and shares

Working with Stock Returns

Getting Real Data from Yahoo Finance

The `tidyfinance` package provides a clean interface to download stock prices:

```
$ uv add tidyfinance
```

```python
import tidyfinance as tf

prices = tf.download_data(
    domain="stock_prices",
    symbols="AAPL",
    start_date="2000-01-01",
    end_date="2024-12-31",
)
print(prices.head())
```

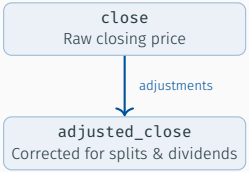
This returns a DataFrame with columns: `symbol`, `date`, `volume`, `open`, `low`, `high`, `close`, and `adjusted_close`.

Why Adjusted Close?

Stock prices are affected by events **after market close**:

- **Stock splits** — a 2-for-1 split halves the price but doubles shares

The adjusted close corrects for these events, giving you the "true" price trajectory for computing returns.



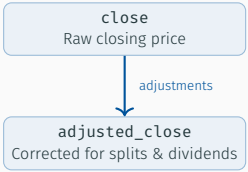
Always use adjusted close when computing returns.

Why Adjusted Close?

Stock prices are affected by events **after market close**:

- **Stock splits** — a 2-for-1 split halves the price but doubles shares
- **Dividends** — cash paid to shareholders reduces the stock price

The adjusted close corrects for these events, giving you the "true" price trajectory for computing returns.



Always use adjusted close when computing returns.

Daily Returns: The Percentage Change

A return measures the percentage change in price from one day to the next:

$$r_t = \frac{p_t}{p_{t-1}} - 1$$

```
returns = (
    prices
    .sort("date")
    .with_columns(
        ret=pl.col("adjusted_close").pct_change()
    )
    .select("symbol", "date", "ret")
    .drop_nulls()
)
print(returns.head())
```

`.pct_change()` computes $(x_t/x_{t-1}) - 1$ for each row. The first row is `null` because there is no previous price.

Summary Statistics of Returns

```python
print(returns.select("ret").describe())
```

```
shape: (9, 2)
+------------+-----------+
| statistic  | ret       |
| ---        | ---       |
| str        | f64       |
+============+===========+
| count      | 6297.0    |
| null_count | 0.0       |
| mean       | 0.001     |
| std        | 0.02      |
| min        | -0.519    |
| 25%        | -0.008    |
| 50%        | 0.001     |
| 75%        | 0.012     |
| max        | 0.139     |
+------------+-----------+
```

Summary statistics reveal the distribution of daily returns: mean, volatility (std), and the range of outcomes.

Downloading Multiple Stocks at Once

Pass a list of symbols to download several stocks in one call:

```
symbols = ["AAPL", "MSFT", "TSLA", "AMZN", "GOOGL"]

prices_daily = tf.download_data(
    domain="stock_prices",
    symbols=symbols,
    start_date="2020-01-01",
    end_date="2024-12-31",
)
print(prices_daily.shape)
print(prices_daily.head())
```

The result is a **long-format** DataFrame — one row per stock per day. The `symbol` column identifies which stock each row belongs to.

Computing Returns for Multiple Stocks

Use `.over("symbol")` to compute returns **within each stock**:

```
returns_daily = (
    prices_daily
    .sort("symbol", "date")
    .with_columns(
        ret=pl.col("adjusted_close")
            .pct_change()
            .over("symbol")
    )
    .select("symbol", "date", "ret")
    .drop_nulls()
)
print(returns_daily.head())
```

`.over("symbol")` is a **window function** — it applies `pct_change()` separately within each stock, preventing returns from crossing stock boundaries.

From Daily to Monthly Returns

Compound daily returns within each month to get monthly returns:

```
returns_monthly = (
    returns_daily
    .with_columns(
        month=pl.col("date").dt.month_start()
    )
    .group_by("symbol", "month")
    .agg(
        ret_monthly=(
            (pl.col("ret") + 1).product() - 1
        )
    )
    .sort("symbol", "month")
)
print(returns_monthly.head())
```

The formula: $(1 + r_1)(1 + r_2) \cdots (1 + r_n) - 1$ compounds daily returns into a single monthly return.

Saving Your Data for Later

Once downloaded, save the data so you do not need to re-download it every time:

```python
# Save to CSV
returns_daily.write_csv("returns_daily.csv")

# Read it back later
returns_daily = pl.read_csv(
    "returns_daily.csv", try_parse_dates=True
)

# Or save to Excel for sharing
returns_monthly.write_excel(
    "returns_monthly.xlsx", worksheet="Monthly Returns"
)
```

Workflow tip: Download once, save to CSV, and read from the file for all subsequent analysis. This is faster and avoids hitting API rate limits.

Exercise 5: Stock Returns

Try It Yourself (10 minutes)

1. Install `tidyfinance`: `uv add tidyfinance`
2. Download daily prices for 3 stocks of your choice (2020–2024)
3. Compute daily returns using `.pct_change()` and `.over("symbol")`
4. Print `.describe()` for the returns column — which stock is most volatile?
5. Save your returns to a CSV file and read it back
6. **Bonus:** Compute monthly returns using `.group_by()` and `.product()`

Combining DataFrames

Your Data Often Lives in Multiple Tables

In practice, the information you need is rarely in one place:

Table	Contains
Stock prices	Ticker, date, price, volume
Company info	Ticker, sector, employees, founding year
Fund holdings	Fund name, ticker, weight

A join combines columns from two tables by matching on a shared key (like `ticker`).

If you've ever used VLOOKUP in Excel, a join does the same thing — but for entire tables at once.

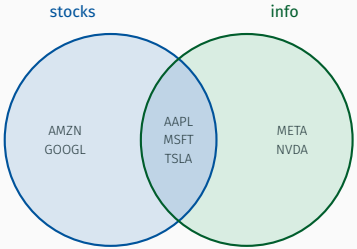
How Joins Work

Imagine two tables sharing a `ticker` column:

- **stocks**: AAPL, MSFT, TSLA, AMZN, GOOGL

Some tickers appear in both (the overlap), some only on one side.

The join type determines which rows survive.



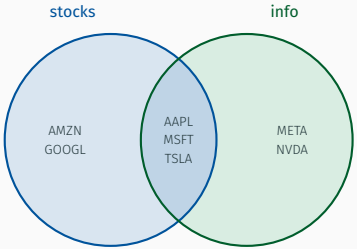
How Joins Work

Imagine two tables sharing a `ticker` column:

- **stocks**: AAPL, MSFT, TSLA, AMZN, GOOGL
- **company_info**: AAPL, MSFT, TSLA, META, NVDA

Some tickers appear in both (the overlap), some only on one side.

The join type determines which rows survive.



Inner Join: Only Matching Rows Survive

```python
company_info = pl.DataFrame({
    "ticker":    ["AAPL", "MSFT", "TSLA", "META", "NVDA"],
    "employees": [164000, 221000, 128000, 67317, 29600],
    "founded":   [1976, 1975, 2003, 2004, 1993],
})

result = stocks.join(company_info, on="ticker", how="inner")
print(result)
```

```
shape: (3, 6)
+--------+--------+--------+--------+-----------+---------+
| ticker | price  | shares | sector | employees | founded |
| ---    | ---    | ---    | ---    | ---       | ---     |
| str    | f64    | i64    | str    | i64       | i64     |
+========+========+========+========+===========+=========+
| AAPL   | 189.84 | 50     | Tech   | 164000    | 1976    |
| MSFT   | 378.91 | 30     | Tech   | 221000    | 1975    |
| TSLA   | 248.42 | 20     | Auto   | 128000    | 2003    |
+--------+--------+--------+--------+-----------+---------+
```

Only the 3 tickers appearing in **both** tables remain. AMZN, GOOGL, META, NVDA are dropped.

Left Join: Keep All Left Rows

```python
result = stocks.join(company_info, on="ticker", how="left")
print(result)
```

```
shape: (5, 6)
+--------+--------+--------+--------+-----------+---------+
| ticker | price  | shares | sector | employees | founded |
| ---    | ---    | ---    | ---    | ---       | ---     |
| str    | f64    | i64    | str    | i64       | i64     |
+========+========+========+========+===========+=========+
| AAPL   | 189.84 | 50     | Tech   | 164000    | 1976    |
| MSFT   | 378.91 | 30     | Tech   | 221000    | 1975    |
| TSLA   | 248.42 | 20     | Auto   | 128000    | 2003    |
| AMZN   | 178.25 | 40     | Retail | null      | null    |
| GOOGL  | 141.8  | 35     | Tech   | null      | null    |
+--------+--------+--------+--------+-----------+---------+
```

All 5 stocks from the left table survive. Where no match exists, Polars fills in `null`.

Anti Join: Find What's Missing

```python
result = stocks.join(company_info, on="ticker", how="anti")
print(result)
```

```
shape: (2, 4)
+--------+--------+--------+--------+
| ticker | price  | shares | sector |
| ---    | ---    | ---    | ---    |
| str    | f64    | i64    | str    |
+========+========+========+========+
| AMZN   | 178.25 | 40     | Retail |
| GOOGL  | 141.8  | 35     | Tech   |
+--------+--------+--------+--------+
```

An anti join returns left-table rows that have **no match** on the right. Useful for finding gaps in your data.

Join Types at a Glance

Type	Rows kept	Use when…
`inner`	Only matching rows from both sides	You need complete records from both tables
`left`	All left rows, matches from right	You want to enrich a primary table
`full`	All rows from both sides	You need the full union of both datasets
`anti`	Left rows with **no** match	Finding gaps or missing data
`semi`	Left rows that have a match	Filtering by existence in another table

In finance, `left` joins are by far the most common — they let you enrich a primary dataset without losing any rows.

Concatenation: Stacking DataFrames

When your data is split across files (e.g. one CSV per month), `pl.concat()` combines them:

```python
jan = pl.DataFrame({"ticker": ["AAPL", "MSFT"], "return_pct": [5.2, 3.1]})
feb = pl.DataFrame({"ticker": ["AAPL", "MSFT"], "return_pct": [-1.4, 2.8]})

result = pl.concat([jan, feb], how="vertical")
print(result)
```

```
shape: (4, 2)
+--------+------------+
| ticker | return_pct |
| ---    | ---        |
| str    | f64        |
+========+============+
| AAPL   | 5.2        |
| MSFT   | 3.1        |
| AAPL   | -1.4       |
| MSFT   | 2.8        |
+--------+------------+
```

`vertical` = stack rows (same columns). `horizontal` = add columns side by side (same rows).

Exercise 6: Joins and Concatenation

Try It Yourself (5 minutes)

1. Create a second DataFrame `sectors` with columns `"ticker"` and `"sector"` for some (but not all) of your tickers
2. Use an **inner join** to combine `portfolio` and `sectors` — how many rows remain?
3. Use a **left join** instead — what happens to tickers without a sector?
4. Use an **anti join** to find which tickers are missing from `sectors`
5. **Bonus:** Use `pl.concat` to stack two DataFrames of returns (one for each month)

Reshaping Data

Wide vs Long: Two Ways to Organise the Same Data

Wide format

Each variable gets its own column.

ticker	Jan	Feb
AAPL	5.2	−1.4
MSFT	3.1	2.8

Compact, human-readable. Good for summary tables.

Long format

One row per observation.

ticker	month	return
AAPL	Jan	5.2
AAPL	Feb	−1.4
MSFT	Jan	3.1
MSFT	Feb	2.8

Tidy, easy to plot and analyse. Good for computation.

Pivot: Reshape from Long to Wide

```python
returns_long = pl.DataFrame({
    "ticker":     ["AAPL", "AAPL", "MSFT", "MSFT"],
    "month":      ["Jan", "Feb", "Jan", "Feb"],
    "return_pct": [5.2, -1.4, 3.1, 2.8],
})

result = returns_long.pivot(
    index="ticker", on="month", values="return_pct"
)
print(result)
```

```
shape: (2, 3)
+--------+-----+------+
| ticker | Jan | Feb  |
| ---    | --- | ---  |
| str    | f64 | f64  |
+========+=====+======+
| AAPL   | 5.2 | -1.4 |
| MSFT   | 3.1 | 2.8  |
+--------+-----+------+
```

Each unique value in `month` becomes a new column. The table gets wider and shorter.

Unpivot: Reshape from Wide to Long

```python
returns_wide = pl.DataFrame({
    "ticker": ["AAPL", "MSFT"],
    "jan":    [5.2, 3.1],
    "feb":    [-1.4, 2.8],
})

result = returns_wide.unpivot(index="ticker", on=["jan", "feb"])
print(result)
```

```
shape: (4, 3)
+--------+----------+-------+
| ticker | variable | value |
| ---    | ---      | ---   |
| str    | str      | f64   |
+========+==========+=======+
| AAPL   | jan      | 5.2   |
| MSFT   | jan      | 3.1   |
| AAPL   | feb      | -1.4  |
| MSFT   | feb      | 2.8   |
+--------+----------+-------+
```

Column names become values in a new `variable` column. The table gets taller and narrower.

When to Use Which Format

	Wide	Long
Good for	Reading, summary tables, correlation matrices	Plotting, `group_by`, regression
Shape	Fewer rows, more columns	More rows, fewer columns
Reshape to	`.pivot()`	`.unpivot()`
Analogies	Excel pivot table output	Tidy data, “one row per observation”

Rule of thumb: If you need to `group_by` or plot, go long. If a human needs to read a table, go wide.

Exercise 7: Reshaping Data

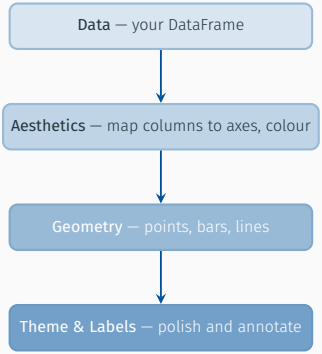
Try It Yourself (5 minutes)

1. Create a long DataFrame with columns `ticker`, `month`, and `return_pct` for 3 tickers across 3 months (9 rows total)
2. Use `.pivot()` to reshape it to wide format (one column per month)
3. Take the wide result and use `.unpivot()` to go back to long — do you get the original back?
4. **Bonus:** Which format would you need to compute the mean return per ticker? Which to show a summary table to your supervisor?

Visualising Data

Build Plots in Layers with `plotnine`

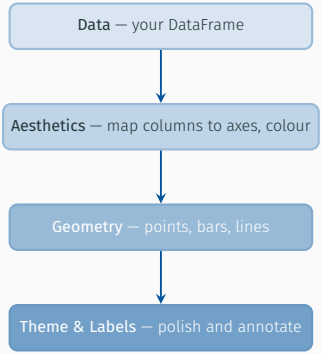
`plotnine` implements the Grammar of Graphics — every plot is built from independent layers:



- Layers are combined with `+`

Build Plots in Layers with `plotnine`

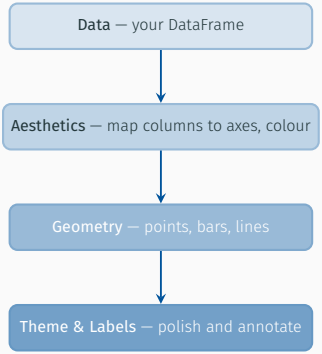
`plotnine` implements the Grammar of Graphics — every plot is built from independent layers:



- Layers are combined with `+`
- Each layer is independent — swap `geom_col()` for `geom_point()` and you get a different chart from the same data

Build Plots in Layers with `plotnine`

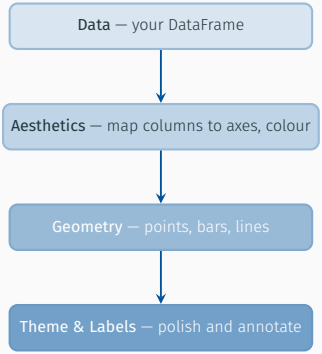
`plotnine` implements the Grammar of Graphics — every plot is built from independent layers:



- Layers are combined with `+`
- Each layer is independent — swap `geom_col()` for `geom_point()` and you get a different chart from the same data
- If you know R's `ggplot2`, this is the same syntax

Build Plots in Layers with `plotnine`

`plotnine` implements the Grammar of Graphics — every plot is built from independent layers:



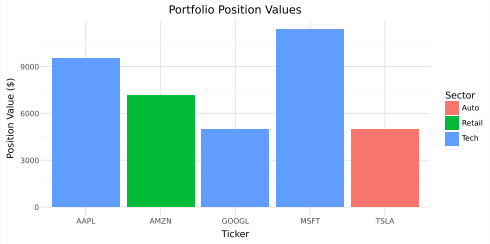
- Layers are combined with `+`
- Each layer is independent — swap `geom_col()` for `geom_point()` and you get a different chart from the same data
- If you know R's `ggplot2`, this is the same syntax
- `plotnine` works with pandas, so we convert: `stocks.to_pandas()`

Plotting Stock Prices Over Time

```python
from plotnine import *

# Convert Polars to pandas for
# plotnine
prices_pd = prices_daily.to_pandas()

(
    ggplot(prices_pd, aes(
        x="date",
        y="adjusted_close",
        color="symbol"))
    + geom_line()
    + labs(title="Stock Prices",
           x="", y="Price ($)",
           color="Symbol")
    + theme_minimal()
)
```

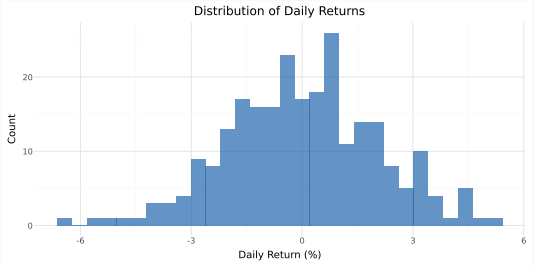


Adding `color="symbol"` in `aes()` automatically draws a separate line for each stock.

Histogram: Distribution of Returns

```python
aapl = (
    returns_daily
    .filter(
        pl.col("symbol") == "AAPL"
    )
    .to_pandas()
)

(
    ggplot(aapl, aes(x="ret"))
    + geom_histogram(
        bins=50,
        fill="#003366",
        alpha=0.7)
    + labs(
        title="AAPL Daily Returns",
        x="Return", y="Count")
    + theme_minimal()
)
```

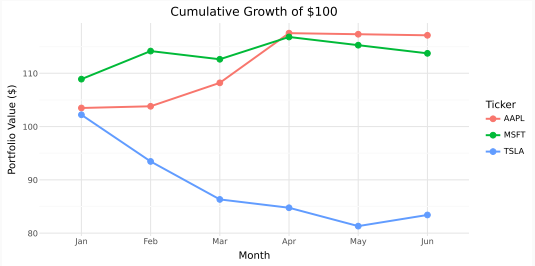


Returns are roughly bell-shaped but with **fat tails** — extreme returns occur more often than a normal distribution predicts.

Cumulative Returns: Growth of $1

```python
cumulative = (
    returns_daily
    .sort("symbol", "date")
    .with_columns(
        growth=
            (1 + pl.col("ret"))
            .cum_prod()
            .over("symbol")
    )
    .to_pandas()
)

(
    ggplot(cumulative, aes(
        x="date", y="growth",
        color="symbol"))
    + geom_line()
    + labs(title="Growth of $1",
           x="", y="Value ($)")
    + theme_minimal()
)
```



`.cum_prod()` computes the running product: $(1 + r_1)(1 + r_2) \cdots (1 + r_t)$, showing how $1 invested on day one grows over time.

The Key `geom_*` Functions

Geom	Plot type	Use when...
`geom_col()`	Bar chart	Comparing categories
`geom_point()`	Scatter plot	Showing relationships between two variables
`geom_line()`	Line chart	Tracking values over time
`geom_histogram()`	Histogram	Showing the distribution of a variable
`geom_smooth()`	Trend line	Adding a fitted line to a scatter plot

The pattern: `ggplot(data, aes(...))` + `geom_*()` + `labs()` + `theme_*()`.
Swap the `geom` to change the chart type. Everything else stays the same.

Exercise 8: Visualising Data

Try It Yourself (5 minutes)
Using the stock returns data from Exercise 5:

1. Plot the adjusted close prices over time for your stocks (use `geom_line()`)
2. Create a histogram of daily returns for one stock (use `geom_histogram()`)
3. Compute cumulative returns with `.cum_prod()` and plot the growth of $1
4. **Bonus:** Map `color="symbol"` to compare multiple stocks on one chart

To save a plot: assign it to a variable and call `p.save("my_plot.pdf")`.

Summary

What You Learned Today

Data structures & I/O

- DataFrames, Series, and data types
- `read_csv` / `write_csv`
- `read_excel` / `write_excel`
- `scan_csv` for lazy reading

Expressions & contexts

- `select`, `with_columns`, **`filter`**, `group_by`
- Arithmetic, comparisons, `when/then/otherwise`
- Casting and expression expansion

Stock returns

- Downloading prices from Yahoo Finance
- Computing daily returns with `pct_change()`
- Window functions with `.over("symbol")`
- Compounding daily → monthly returns

Transformations & visualisation

- Joins, concatenation, pivot/unpivot
- Grammar of Graphics with `plotnine`
- Price charts, return histograms, cumulative growth

Quick Self-Check

Can you answer these in your own words?

1. What is the difference between `read_csv` and `scan_csv`?

Quick Self-Check

Can you answer these in your own words?

1. What is the difference between `read_csv` and `scan_csv`?
2. What is the difference between `select` and `with_columns`?

Quick Self-Check

Can you answer these in your own words?

1. What is the difference between `read_csv` and `scan_csv`?
2. What is the difference between `select` and `with_columns`?
3. Why do we use `adjusted_close` instead of `close` for returns?

Quick Self-Check

Can you answer these in your own words?

1. What is the difference between `read_csv` and `scan_csv`?
2. What is the difference between `select` and `with_columns`?
3. Why do we use `adjusted_close` instead of `close` for returns?
4. What does `.pct_change().over("symbol")` do, and why is `.over()` needed?

Quick Self-Check

Can you answer these in your own words?

1. What is the difference between `read_csv` and `scan_csv`?
2. What is the difference between `select` and `with_columns`?
3. Why do we use `adjusted_close` instead of `close` for returns?
4. What does `.pct_change().over("symbol")` do, and why is `.over()` needed?
5. When do you use a `left` join vs an `inner` join?

Quick Self-Check

Can you answer these in your own words?

1. What is the difference between `read_csv` and `scan_csv`?
2. What is the difference between `select` and `with_columns`?
3. Why do we use `adjusted_close` instead of `close` for returns?
4. What does `.pct_change().over("symbol")` do, and why is `.over()` needed?
5. When do you use a `left` join vs an `inner` join?
6. In `plotnine`, what do `aes()` and `geom_line()` each do?

What's Next

- Next week: panel regression analysis with financial datasets

If something breaks: read the error message, check your data types, try again.
Still stuck? Email me at ce50@st-andrews.ac.uk.

What's Next

- Next week: panel regression analysis with financial datasets
- **Homework:** complete today's exercises — download stock data for your own tickers and explore the returns

If something breaks: read the error message, check your data types, try again.
Still stuck? Email me at ce50@st-andrews.ac.uk.

What's Next

- Next week: panel regression analysis with financial datasets
- **Homework:** complete today's exercises — download stock data for your own tickers and explore the returns
- Resources:

If something breaks: read the error message, check your data types, try again.
Still stuck? Email me at ce50@st-andrews.ac.uk.

What's Next

- Next week: panel regression analysis with financial datasets
- **Homework:** complete today's exercises — download stock data for your own tickers and explore the returns
- Resources:
  - Polars user guide: https://docs.pola.rs/

If something breaks: read the error message, check your data types, try again.
Still stuck? Email me at ce50@st-andrews.ac.uk.

What's Next

- Next week: panel regression analysis with financial datasets
- **Homework:** complete today's exercises — download stock data for your own tickers and explore the returns
- Resources:
  - Polars user guide: https://docs.pola.rs/
  - Polars I/O guide: https://docs.pola.rs/user-guide/io/

If something breaks: read the error message, check your data types, try again.
Still stuck? Email me at ce50@st-andrews.ac.uk.

What's Next

- Next week: panel regression analysis with financial datasets
- **Homework:** complete today's exercises — download stock data for your own tickers and explore the returns
- Resources:
  - Polars user guide: https://docs.pola.rs/
  - Polars I/O guide: https://docs.pola.rs/user-guide/io/
  - Tidy Finance with Python: https://www.tidy-finance.org/python/

If something breaks: read the error message, check your data types, try again.
Still stuck? Email me at ce50@st-andrews.ac.uk.

What's Next

- Next week: panel regression analysis with financial datasets
- **Homework:** complete today's exercises — download stock data for your own tickers and explore the returns
- Resources:
  - Polars user guide: https://docs.pola.rs/
  - Polars I/O guide: https://docs.pola.rs/user-guide/io/
  - Tidy Finance with Python: https://www.tidy-finance.org/python/
  - plotnine gallery: https://plotnine.org/

If something breaks: read the error message, check your data types, try again.
Still stuck? Email me at ce50@st-andrews.ac.uk.