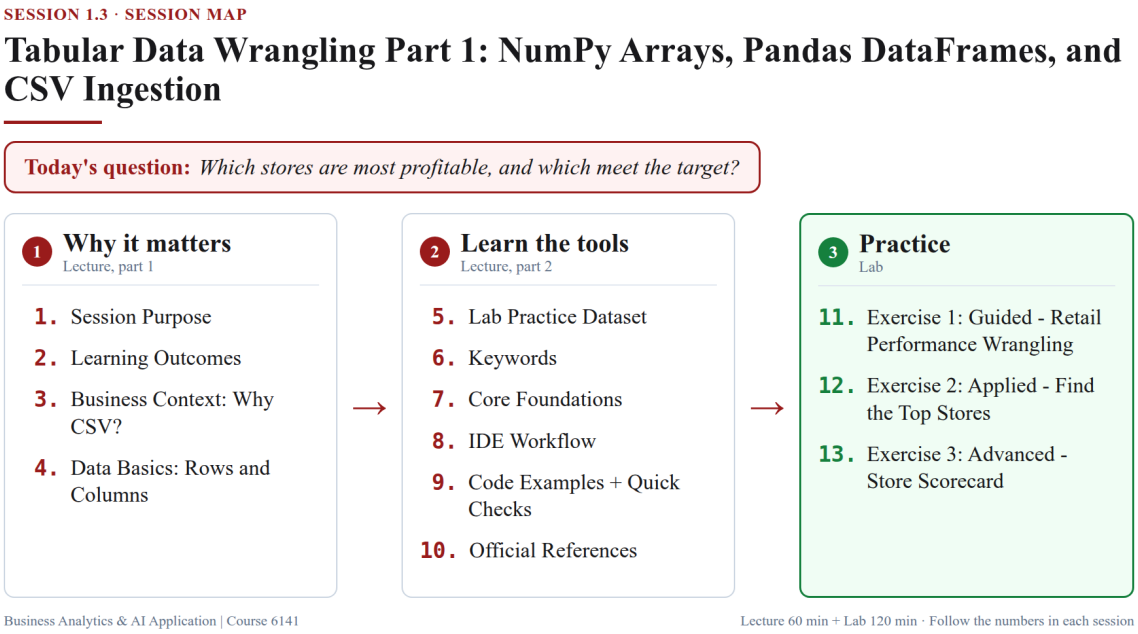
Session 1.3: Tabular Data Wrangling Part 1: NumPy Arrays, Pandas DataFrames, and CSV Ingestion

Course 6141: Business Analytics & AI Application | Reading for Week 3



Schedule: Week 3 | In-person classes (1:2 Delivery Ratio: 60 Min Lecture + 120 Min Lab)

1. Session Purpose

- Business systems (sales registers, accounting, customer databases, Excel) share tables as Comma-Separated Values (CSV) files
- Our case: a retail chain exports monthly results for its 12 stores as a CSV file
- Each row holds one store's sales, costs, visitors, and customer rating
- Your manager asks: which stores are most profitable, and which meet the target? Today, Python answers these questions for every store at once

2. Expected Learning Outcomes (By the end of this session, you will be able to)

1. "Did the store data load correctly?": read and inspect a `DataFrame` (a table in Python, like one Excel sheet)

2. “What is each store’s profit margin?”: create new columns with vectorized math (one formula runs on every row at once)
3. “Which stores meet our target?”: filter rows with `.loc[]` (keep rows that match a condition) and fix a `KeyError` (the column name does not exist)

3. Business Context: Why CSV?

- Where business data comes from: sales registers, the accounting system, the customer database, and Excel reports. Each system can export its tables as one shared file.
- `.xlsx` (Excel file) vs `.csv` (CSV file):

	`.xlsx` Excel file	`.csv` CSV file
Format	Packed format	Plain text
What you need to open it	A spreadsheet program such as Excel	Almost anything: Notepad, Excel, Google Sheets, Python `pandas`, AI assistants
Opened in Notepad	Unreadable symbols (starts with `PK`)	Readable: `Store_ID,Region_Name,Monthly_Sales_USD`

- Trade-off: CSV keeps only values, with no formulas and no formatting.

4. Data Basics: Rows and Columns

- **Row** (read across): one record. Here, all the facts about one store, such as row `1` = everything about `STR-1001`.
- **Column** (read down): one type of information for every store, such as `Monthly_Sales_USD`. In pandas, one column is a `Series`.
- **Cell:** one value where a row and a column meet, such as `185000.0`.
- Size of a table: `df.shape` returns `(12, 6)` = 12 rows, 6 columns. Row labels (the `index`) start at `0`, not 1.

5. Lab Practice Dataset

- File Link: `data/session_01_03_dataset.csv`
- Audited 12-row commercial store dataset used for vector arithmetic and filtering:
  - `Store_ID`: Unique retail location code (e.g., `STR-1001`).
  - `Region_Name`: Geographic division (East, West, North, South).
  - `Monthly_Sales_USD`: Gross store revenue in U.S. Dollars.
  - `Operating_Cost_USD`: Monthly operational store overhead in U.S. Dollars.
  - `Customer_Footfall_Count`: Total monthly foot traffic count.
  - `Customer_Satisfaction_Score`: Customer review rating (1.0 to 5.0 scale).

6. Keywords (Core Programming Concepts)

- `pandas` and `NumPy`:
  - What: free Python libraries for tables (`pandas`) and fast number arrays (`NumPy`)
  - Why: they sort, filter, and calculate tables of any size
- `CSV file` and `pd.read_csv()`:
  - What: a plain-text table where commas separate values; `pd.read_csv()` loads it
  - Why: most business systems export data as CSV
- `DataFrame`:
  - What: a whole table in Python, with row labels and column names
  - Why: it is the main object you clean, calculate, and filter
- `Series`:
  - What: one column of a DataFrame, such as `df["Monthly_Sales_USD"]`
  - Why: most calculations work column by column
- `Vectorized Computation`:
  - What: one formula applied to every row at once, with no loop
  - Why: fast, and nothing to drag down row by row
- `.loc[]` vs `.iloc[]`:
  - What: `.loc[]` selects by label or condition; `.iloc[]` selects by position number
  - Why: pull out exactly the rows a question needs
  - Example: `df.loc[df["Region_Name"] == "West"]` keeps the West rows; `df.iloc[0:2]` keeps rows 0 and 1 (`0:2` starts at 0 and stops **before** 2)

7. Core Programming & Computational Foundations

- **7.1 Columns, not cells: one formula for every row**
  - *The idea:* In Excel, a formula lives in one cell. To calculate the margin of 12 stores, you type the formula once and drag it down, which creates 12 separate copies, one per row. In pandas, a calculation is written once for a whole column, and pandas applies it to every row at the same time. This is called vectorized computation: the formula works on the whole column (a "vector" of values) instead of cell by cell.
  - *See it on real data:* operating margin for the first 3 stores.

	Excel	pandas
What you write	`=(C2-D2)/C2*100` in `E2`, then drag down to `E13`	`df["Operating_Margin_Pct"] = (df["Monthly_Sales_USD"] - df["Operating_Cost_USD"]) / df["Monthly_Sales_USD"] * 100`
How many formulas exist	12, one per row	1, for the whole column
`STR-1000`	`30.99`	`30.985915`
`STR-1001`	`34.05`	`34.054054`
`STR-1002`	`27.55`	`27.551020`

  - *Common mistake (in Excel):* someone types a number over `E5`, for example to "fix" a value for a meeting. That row's formula is gone, the cell still looks normal, and next month's report is wrong without any warning sign that catches the eye. In pandas there is no per-row copy to overwrite: the next run recalculates every row from the data.
  - *More examples:*

Code	Result	What it tells you
`df["Monthly_Sales_USD"] - df["Operating_Cost_USD"]`	`44000.0` for `STR-1000`	operating profit for every store
`df["Monthly_Sales_USD"] / df["Customer_Footfall_Count"]`	`11.45` for `STR-1000`	sales per visitor: what an average visitor spends
`df["Operating_Cost_USD"] / df["Monthly_Sales_USD"] * 100`	`69.01` for `STR-1000`	cost as a share of sales, in percent
`df["Monthly_Sales_USD"] * 1.05`	`149100.0` for `STR-1000`	a "what if sales grew 5%?" scenario
`df["Monthly_Sales_USD"] / 1000`	`142.0` for `STR-1000`	sales in thousands, easier to read in a report

  - *Remember:* write the rule once for the column; pandas applies it to every row.

- **7.2 Inside a DataFrame: index, columns, and values**
  - *The idea:* A DataFrame has three parts, like a labeled spreadsheet:
    - the **index**: one label per row, shown at the left. By default the labels are `0, 1, 2, ...`, counting from `0`, not `1`;
    - the **columns**: one name per column, shown at the top, such as `Store_ID`;
    - the **values**: the data itself, stored as a NumPy array (a fast block of numbers that pandas uses underneath).
  - *See it on real data:* the first 3 rows as pandas prints them.

```
   Store_ID Region_Name  Monthly_Sales_USD  ...
0  STR-1000        East           142000.0  ...
1  STR-1001        West           185000.0  ...
2  STR-1002       North            98000.0  ...
```

The `0`, `1`, `2` on the left are the index, not data. `STR-1001` is in the row labeled `1`, which is the **second** row.

- *One column vs a smaller table:* `df["Store_ID"]` (single brackets) returns one column, a `Series`. `df[["Store_ID", "Region_Name"]]` (double brackets: a list inside the brackets) returns a smaller `DataFrame` with the listed columns.
- *Common mistake:* counting rows from 1. In Python, the first row has position `0`, so "the third row" is position `2`.
- *More examples:*

Code	Result	What it tells you
`type(df["Store_ID"])`	`Series`	single brackets give one column
`df[["Store_ID", "Region_Name"]].shape`	`(12, 2)`	double brackets give a smaller table: 12 rows, 2 columns
`df.columns.tolist()`	`['Store_ID', 'Region_Name', ...]`	the exact column names, to copy instead of typing
`df.index.tolist()`	`[0, 1, 2, ..., 11]`	the row labels start at `0`
`len(df)`	`12`	the number of rows
`df.loc[0, "Store_ID"]`	`'STR-1000'`	one value: row label `0`, column `Store_ID`

- *Remember:* index on the left, column names on top, values in between; counting starts at `0`.

- **7.3 Data types: every column has exactly one type**
  - *The idea:* pandas gives each column one data type (`dtype`), and every value in that column follows it. The type decides what you can do with the column: you can add numbers, but not text.

`dtype`	Meaning	Columns in our data
`float64`	decimal numbers	`Monthly_Sales_USD`, `Operating_Cost_USD`, `Customer_Satisfaction_Score`
`int64`	whole numbers	`Customer_Footfall_Count`
`str`	text (older pandas versions show `object`)	`Store_ID`, `Region_Name`

  - *Why it matters:* if a number is typed with a comma, such as `"1,200"`, pandas cannot read it as a number, so the whole column becomes text. Adding text joins the pieces instead of summing them: a column with `"1,200"` and `"800"` "sums" to `"1,200800"`, not `2000`. In Excel the same value would sit in its cell as text and be quietly skipped by `SUM`.

– *How to catch it:* run `df.dtypes` right after loading. A sales column showing `str` instead of `float64` means some values were stored as text.

– *More examples:*

Code	Result	What it tells you
`df["Monthly_Sales_USD"].sum()`	`1829000.0`	a number column adds up as numbers
`df["Region_Name"].head(3).sum()`	`'EastWestNorth'`	a text column "adds up" by joining the words
`pd.Series(["1,200", "800"]).sum()`	`'1,200800'`	numbers stored as text are joined, not summed
`pd.Series(["1,200", "800"]).str.replace(",", "").astype(float).sum()`	`2000.0`	remove the comma, convert to `float`, and the sum is right
`df["Customer_Footfall_Count"].mean()`	`13075.0`	an `int64` column works for averages too

– *Remember:* check the types first; a number column must show `float64` or `int64`.

- **7.4 Check before you calculate: four inspection tools**

– *The idea:* A professional analyst never calculates on data they have not looked at. Four commands answer the first questions about any new file:

Command	Question it answers	Result for our file
`df.shape`	How big is the table?	`(12, 6)`: 12 rows, 6 columns
`df.head(3)`	What do the first rows look like?	`STR-1000`, `STR-1001`, `STR-1002`
`df.info()`	What are the types, and are values missing?	`12 non-null` in every column: no gaps
`df.describe()`	What are the typical, smallest, and largest values?	sales: mean `152416.67`, min `89000.0`, max `225000.0`

– *Common mistake:* `df.shape` has no brackets, because it is a stored value (an attribute). `df.head()` and `df.info()` do have brackets, because they are actions (methods). Writing `df.shape()` raises `TypeError: 'tuple' object is not callable`.

– *More examples:*

Code	Result	What it tells you
`df["Region_Name"].value_counts()`	`East 3`, `West 3`, `North 3`, `South 3`	how many stores each region has
`df.isnull().sum().sum()`	`0`	the number of empty cells in the whole table

Code	Result	What it tells you
`df["Customer_Satisfaction_Score"].min()`	`3.8`	the lowest rating
`df["Customer_Satisfaction_Score"].max()`	`4.9`	the highest rating
`df["Customer_Satisfaction_Score"].mean()`	`4.35` (rounded)	the average rating

  - *Remember:* shape, head, info, describe: look first, calculate second.

- **7.5 Picking rows: by label with `.loc[]`, by position with `.iloc[]`**
  - *The idea:* There are two ways to point at a row. `.loc[]` uses the **label** in the index, or a condition. `.iloc[]` uses the **position**, counting from `0`. On a freshly loaded file the labels and positions are the same, so the two look alike. They differ as soon as the rows are reordered.
  - *See it on real data:* sort the stores by sales, from lowest to highest. The row order changes, but every row keeps its label.

Request	Meaning	Result
`sorted_df.loc[1, "Store_ID"]`	the row whose label is `1`	`STR-1001`, wherever it now sits
`sorted_df.iloc[0]["Store_ID"]`	whatever row is now first	`STR-1006`, the lowest sales

  - *Filtering with `.loc[]`:* `.loc[]` also accepts a condition, such as `df.loc[df["Region_Name"] == "West"]`, and keeps only the rows where the condition is `True`. This is the form you will use most.
  - *Common mistake:* misspelling a column name. `df["Monthly_Sale_USD"]` (missing `s`) raises `KeyError: 'Monthly_Sale_USD'`: pandas looked for a column with that exact name and found none. Copy column names from `df.columns` instead of typing them.
  - *More examples:*

Code	Result	What it tells you
`df.loc[df["Region_Name"] == "East", "Store_ID"]`	`STR-1000`, `STR-1004`, `STR-1008`	the East stores: a condition plus one column
`df.loc[df["Customer_Satisfaction_Score"] >= 4.5, "Store_ID"]`	`STR-1001`, `STR-1004`, `STR-1005`, `STR-1008`, `STR-1011`	the 5 best-rated stores
`df.iloc[0]["Store_ID"]`	`'STR-1000'`	the first row
`df.iloc[-1]["Store_ID"]`	`'STR-1011'`	`-1` means the last row

Code	Result	What it tells you
`df.iloc[0:3]["Store_ID"]`	`STR-1000`, `STR-1001`, `STR-1002`	rows at positions 0, 1, 2 (stops before 3)
`df.loc[2, "Region_Name"]`	`'North'`	row label 2, column by name
`df.iloc[2, 1]`	`'North'`	the same cell by position: row 2, column 1

  - *Remember:* `.loc[]` = label or condition; `.iloc[]` = position from `0`.

8. Development Workflow in Google Antigravity IDE

`!assets/antigravity_ide_interface.png`

- `[1]` File Explorer: create your script with your name and the session code (for example, `Reacher_S13_demo_1.py`), and check the dataset is in `data/`.
- `[2]` Code Editor: write a few lines, run, repeat. Small steps keep errors small.
- `[3]` AI Agent Chat: on an error, capture the traceback (Python's error report) with `Win + Shift + S` (macOS: `Cmd + Shift + 4`), paste it with `Ctrl + V`, and ask why it failed.
- `[4]` Terminal: run `python` followed by your script name. Return code `0` means the script finished with no errors.

9. Hands-On Coding Micro-Examples

- Every example has three parts: **1. Input** (load the data), **2. Process** (calculate or select), and **3. Output** (print the result).
- Name each file with your own name and the session code, for example `Reacher_S13_demo_1.py` (pattern: `YourName_S13_demo_1.py`).
- *Micro-Example 1 (Load a CSV File and Inspect It):* `YourName_S13_demo_1.py`

```python
# 1. Input: load the tool and the data
import pandas as pd
df = pd.read_csv("data/session_01_03_dataset.csv")

# 2. Process: measure the table and take a preview
dimensions = df.shape
column_types = df.dtypes
preview = df.head(3)

# 3. Output: show the results
print("Dataset Dimensions (Rows, Columns):", dimensions)
print("Column Data Types:\n", column_types)
print(preview)
```

  - *How it works, line by line:*

    - `import pandas as pd` loads the pandas library and gives it the short name `pd`, so every pandas tool can be written as `pd.something`.
    - `pd.read_csv(...)` opens the CSV file, reads the first line as column names, and stores the table in a variable named `df` (short for DataFrame). The path `data/...` starts from the project folder.
    - `df.shape`, `df.dtypes`, and `df.head(3)` measure the table, list each column's type, and copy the first 3 rows. Each result is stored in its own variable, so part 3 can print it.
    - `print()` shows a result in the Terminal. A script that calculates without printing shows nothing at all.

  – *Read the output:*

```
Dataset Dimensions (Rows, Columns): (12, 6)
Column Data Types:
 Store_ID                          str
Region_Name                       str
Monthly_Sales_USD             float64
...
   Store_ID Region_Name  ...  Customer_Footfall_Count
↪  Customer_Satisfaction_Score
0  STR-1000        East  ...                    12400
↪  4.2
```

    - `(12, 6)` means 12 rows (stores) and 6 columns, as expected for this file.
    - The money columns are `float64` (numbers), so you can calculate with them.
    - The `...` in the preview is not missing data: pandas hides middle columns when the table is wider than the Terminal.

- *Quick Check 1 (`.tail()`):*

  – New tool: `.tail(n)` shows the last `n` rows. Why: new records are often added at the bottom.
  – Task: in part **2. Process**, change `preview = df.head(3)` to `preview = df.____(3)` so it shows the last 3 rows.
  – Check: the preview shows index `9`, `10`, `11` (`STR-1009`, `STR-1010`, `STR-1011`).

- *Micro-Example 2 (Calculate New Columns Without a Loop):* `YourName_S13_demo_2.py`

```python
# 1. Input: load the store data
import pandas as pd
df = pd.read_csv("data/session_01_03_dataset.csv")

# 2. Process: profit and margin for every store at once
df["Operating_Profit_USD"] = (
    df["Monthly_Sales_USD"] - df["Operating_Cost_USD"]
)
df["Operating_Margin_Pct"] = (
    df["Operating_Profit_USD"] / df["Monthly_Sales_USD"] * 100.0
)
```

```
# 3. Output: show the first three stores
print(df[["Store_ID", "Monthly_Sales_USD",
          "Operating_Margin_Pct"]].head(3))
```

  - *How it works, line by line:*
    - `df["Monthly_Sales_USD"] - df["Operating_Cost_USD"]` subtracts two columns. pandas lines them up row by row, so each store's cost is subtracted from that same store's sales (section 7.1).
    - `df["Operating_Profit_USD"] = ...` stores the result as a new column. If the name does not exist yet, pandas adds it at the right end of the table.
    - The second calculation uses the new profit column straight away: profit / sales x 100.
    - The round brackets around each calculation only let the formula continue on the next line; they do not change the result.
    - `df[["Store_ID", "Monthly_Sales_USD", "Operating_Margin_Pct"]]` (double brackets) keeps just those 3 columns, and `.head(3)` keeps the first 3 rows.
  - *Read the output:*

```
   Store_ID  Monthly_Sales_USD  Operating_Margin_Pct
0  STR-1000           142000.0             30.985915
1  STR-1001           185000.0             34.054054
2  STR-1002            98000.0             27.551020
```

    - `STR-1000`: `(142000 - 98000) / 142000 x 100 = 30.99%`. Of every $100 of sales, about $31 is left as operating profit.
    - `STR-1002` sells the least and also keeps the smallest share as profit (`27.55%`).
    - The long decimals are correct but hard to read; Quick Check 2 rounds them.

- *Quick Check 2 (`.round()`):*
  - New tool: `.round(n)` rounds every number in a column to `n` decimal places. Why: long decimals such as `30.985915` are hard to read in a report.
  - Task: add one line at the end of part **2.** **Process**: `df["Operating_Margin_Pct"] = df["Operating_Margin_Pct"].____(1)`
  - Check: the margins print as `31.0`, `34.1`, and `27.6`.

- *Micro-Example 3 (Filter Rows and Fill Missing Values):* `YourName_S13_demo_3.py`

```
# 1. Input: load the store data
import pandas as pd
df = pd.read_csv("data/session_01_03_dataset.csv")

# 2. Process: keep West stores with sales of $150,000+
west_high_sales = df.loc[
    (df["Region_Name"] == "West")
    & (df["Monthly_Sales_USD"] >= 150000.0)
]
#    then fill missing satisfaction scores with the median
median_score = df["Customer_Satisfaction_Score"].median()
```

```
df["Customer_Satisfaction_Score"] = (
    df["Customer_Satisfaction_Score"].fillna(median_score)
)

# 3. Output: show the filtered stores and the median
print(west_high_sales[["Store_ID", "Monthly_Sales_USD"]])
print("Median Satisfaction Score Imputed:", median_score)
```

  - *How it works, line by line:*
    - `(df["Region_Name"] == "West")` checks every row and gives `True` for West stores and `False` for the others. Note the double `==`: it compares, while a single `=` stores a value.
    - `(df["Monthly_Sales_USD"] >= 150000.0)` gives `True` for stores selling $150,000 or more.
    - `&` combines the two answers row by row: a row is `True` only if both are `True`. Each condition sits in its own brackets so that Python compares first and combines second.
    - `df.loc[...]` keeps the `True` rows and stores them in `west_high_sales`; the original `df` is not changed.
    - `.median()` finds the middle value of the satisfaction scores: half the stores score higher, half lower. `.fillna(median_score)` puts that value into any empty cell (`NaN`, "Not a Number").
  - *Read the output:*

```
   Store_ID  Monthly_Sales_USD
1  STR-1001           185000.0
5  STR-1005           175000.0
9  STR-1009           160000.0
Median Satisfaction Score Imputed: 4.35
```

    - All 3 West stores sell $150,000 or more, so all 3 pass. With a higher limit, such as $170,000, only 2 would. The index labels `1`, `5`, `9` show where each store sat in the original table: filtering keeps the labels.
    - The median of the 12 scores is `4.35`. This dataset has no empty scores, so `.fillna()` changes nothing today, but the same line protects the script when next month's file has gaps.

- *Quick Check 3 (`len()`):*
  - New tool: `len()` counts items; for a DataFrame, it returns the number of rows. Why: it answers "how many?" questions in one step. Example: `len(df)` returns `12`.
  - Task: add one line to part **3. Output**: `print("Stores found:", ___(west_high_sales))`
  - Check: the Terminal shows `Stores found: 3`.

10. Official Python Documentation & References

- Pandas Official Documentation: 10 Minutes to Pandas.
- Pandas Official Guide: Indexing and Selecting Data (.loc and .iloc).
- NumPy Official Documentation: NumPy Quickstart Tutorial.

11. Exercise 1: Guided Practice - Retail Performance Wrangling

- *Step 1: Open Workspace:* In Google Antigravity IDE, create a new script named `YourName_S13_wrangling.py`, replacing `YourName` with your own name (for example, `Reacher_S13_wrangling.py`). Verify that `data/session_01_03_dataset.csv` is present in Panel `[1]`.
- *Step 2: Load, Calculate, and Summarize:* Type the following Python statements:

```
# 1. Input: load the store data
import pandas as pd
df = pd.read_csv("data/session_01_03_dataset.csv")

# 2. Process: add profit and margin columns, then summarize
df["Operating_Profit_USD"] = (
    df["Monthly_Sales_USD"] - df["Operating_Cost_USD"]
)
df["Operating_Margin_Pct"] = (
    df["Operating_Profit_USD"] / df["Monthly_Sales_USD"] * 100.0
)
summary = df.describe()

# 3. Output: show summary statistics for sales and margin
print("Descriptive Financial Statistics:\n",
      summary[["Monthly_Sales_USD", "Operating_Margin_Pct"]])
```

  - `.describe()` returns summary statistics: count, mean, standard deviation, min, max, and quartiles.

- *Step 3: Compare Excel Mechanics with Pandas:*
  - In Excel, you type `=(C2-D2)/C2*100` in cell `E2` and drag it down, so every row gets its own copy of the formula. If someone types a number over `E5`, that row is wrong, and the mistake is easy to miss.
  - In Pandas, one line of `df["Operating_Margin_Pct"]` code covers every row, so there is no copy to overwrite.
- *Step 4: Execute in Terminal:* Run the script via Panel `[4]` (Terminal) using `python YourName_S13_wrangling.py`. Confirm that the summary statistics are printed.

12. Exercise 2: Applied Practice - Find the Top Stores

- *Coding Challenge:* In `YourName_S13_practice.py`, find the top stores: `Operating_Margin_Pct >= 30.0` and `Customer_Satisfaction_Score >= 4.5`.
- *Task 1 (Targeted Filtering):* Use `.loc[]` to select the qualified stores and output only `Store_ID`, `Region_Name`, and `Operating_Margin_Pct`.
- *Task 2 (Deliberate KeyError & AI Diagnosis):* Deliberately misspell a column name (e.g., `df["Monthly_Sale_USD"]` instead of `"Monthly_Sales_USD"`). Read the `KeyError` (the column name does not exist), capture the traceback with `Win + Shift + S`, paste it into Panel `[3]` (AI Agent Chat) with `Ctrl + V`, and review the AI's explanation.

- *Task 3 (Clean CSV Export):* Correct the column name, then sort with `.sort_values()`; add `ascending=False` for the highest margin first. Save the table with `.to_csv()` to `data/top_performing_stores.csv`.
- *Success Check:* the result has 5 stores, led by `STR-1011` (South, 36.89%), and the script ends with return code `0`.

13. Exercise 3: Advanced Practice - Store Scorecard

- *Goal:* Build a one-page scorecard that sorts all 12 stores into three tiers, so the regional manager can see at a glance which stores to reward and which to watch.
- *What you need:*
  - File: `data/session_01_03_dataset.csv` (12 rows, described in **5. Lab Practice Dataset**).
  - Script name: `YourName_S13_challenge.py`.
  - Tools you already know: `pd.read_csv()`, column math, `.round()`, `&`, `.loc[]`, `.sort_values()`, `len()`, and `.to_csv()`.
- *Requirements (follow them in order):*
  1. Add `Operating_Profit_USD` = `Monthly_Sales_USD` - `Operating_Cost_USD`.
  2. Add `Operating_Margin_Pct` = `Operating_Profit_USD` / `Monthly_Sales_USD` x 100, rounded to 1 decimal place.
  3. Add `Sales_per_Visitor_USD` = `Monthly_Sales_USD` / `Customer_Footfall_Count`, rounded to 2 decimal places. It shows how much an average visitor spends.
  4. Add a `Tier` column using these rules, in this order:
     - Every store starts as `"Standard"`.
     - `"Gold"`: `Operating_Margin_Pct >= 33.0` **and** `Customer_Satisfaction_Score >= 4.5`.
     - `"Watch"`: `Operating_Margin_Pct < 30.0`.
  5. Print how many stores are in each tier, one line per tier, for example `Gold stores: 4`.
  6. Print all 12 stores sorted by `Operating_Margin_Pct`, highest first, showing only `Store_ID`, `Region_Name`, `Operating_Margin_Pct`, `Sales_per_Visitor_USD`, `Customer_Satisfaction_Score`, and `Tier`.
  7. Save the same sorted table to `data/store_scorecard.csv` without row labels.
  8. Answer in a comment at the end of your script: which tier holds the store with the highest `Sales_per_Visitor_USD`, and what does that tell the manager?
- *New tool for this exercise: set a value for selected rows only:*
  - `df["Tier"] = "Standard"` writes one value into every row of a new column.
  - `df.loc[mask, "Tier"] = "Gold"` changes only the rows where `mask` is `True`. `mask` is a condition like the one in Micro-Example 3, with each part wrapped in `( )` and joined with `&`.
- *Hints:*
  - To count a tier, filter first, then use `len()`: `len(df.loc[df["Tier"] == "Gold"])`.
  - Pass a list of column names to `df[[ ... ]]` to keep only the columns you need.
- *Self-Check (expected results):*
  - `Gold stores: 4` (`STR-1011`, `STR-1004`, `STR-1008`, `STR-1001`), `Watch stores: 5`, `Standard stores: 3`.

– The first row of the sorted table is `STR-1011` (South, `36.9`), and the last row is `STR-1006` (North, `23.6`).
– The highest `Sales_per_Visitor_USD` is `STR-1002` (`11.95`), a **Watch** store with a margin of `27.6`. High spending per visitor does not guarantee a high profit margin.