Data Science Fundamentals

A Complete 10-Chapter Data Science & ML Course

Topics covered:

NumPy & pandas, data cleaning & wrangling, statistics for data science

Matplotlib & Seaborn visualization, a full seven-step EDA methodology

Three real worked datasets, applied start to finish

Exercises: 30 hands-on scenarios with worked solutions

Format: A4 · Dark-theme code examples

Course 1 of 6 in the Data Science & ML subject · the prerequisite layer for ml1/nn1/nlp1/llm1

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

What Data Science Actually Is & The Data Science Workflow

Data Science Fundamentals

Chapter 1 · What Data Science Actually Is & The Data Science Workflow

This course assumes real, working Python fluency — `py1` – `py4` already cover variables, control flow, functions, files, JSON, and packaging in depth, and none of that gets re-taught here. This chapter picks up almost exactly where `py1-9`'s own brief `pip` /virtual-environment coverage left off, moving from "how to manage Python packages in general" into the specific stack this course is actually about.

Data Science Is Not a Synonym for Machine Learning

Four terms get used almost interchangeably in casual conversation, and they shouldn't be: **Business Intelligence (BI)** is retrospective reporting on known questions — dashboards showing what already happened. **Data Analytics** is answering a specific business question with data — "did the redesign increase signups?" **Machine Learning** is building models that make predictions or find patterns automatically from data. **Data Science** is the broader discipline that combines statistics, programming, and domain knowledge to extract insight from data — and machine learning is one tool inside that discipline, not a synonym for the whole thing.

> **WHY THIS COURSE DRAWS THE LINE WHERE IT DOES**
>
> A huge amount of real, valuable data science work — cleaning a messy dataset, understanding what it actually contains, finding the pattern that answers the question someone actually asked — happens before any model gets built, and plenty of real data science projects never need a model at all. This course covers that entire "before modeling" layer in depth. Building predictive models is a genuinely separate skill, covered start to finish in `ml1`.

The Data Science Workflow

Five stages, in order, form the backbone this whole subject is organized around:

1. **Collect** — gather data from files, APIs, databases, or the web. Covered only briefly here; a full hands-on project (a real web-scraping tool) is deferred to `dsproj1`, deliberately chosen there as a good first project because it needs no ML at all.

2 **Clean** — handle missing values, duplicates, inconsistent types, and messy strings so the data can actually be trusted. Covered in full in `ds1-4`.

3 **Explore** — understand what the data actually contains: summary statistics, distributions, relationships, visualizations. Covered in `ds1-5` through `ds1-9` — the largest share of this course by chapter count, because it's the largest share of real data science work.

4 **Model** — build a predictive or classification model. Deliberately **out of scope** for this course — this is `ml1`'s entire job, start to finish.

5 **Communicate** — turn findings into something another person can actually use: a chart, a report, a clear conclusion. Threaded through `ds1-7`–`ds1-9` and the capstone rather than given its own dedicated chapter.

Jupyter Notebooks — The Standard Data Science Environment

`py1-1` introduced running Python as a plain script, top to bottom, once. Data science work is exploratory and iterative by nature — you look at a dataset, try something, look at the result, adjust, try again — and a plain script re-run from the top every time that happens is genuinely slow to work with. A **Jupyter notebook** is a document made of individual, independently runnable **cells**: code cells that keep their own output (including inline charts) visible directly beneath them, and text cells for notes, mixed freely in one file. Running one cell at a time, keeping earlier results in memory, and only re-running what actually changed is the standard way real data science work gets done — not a replacement for `py1`'s own script model in general, just the better-fitting tool for this specific, exploratory kind of work.

The Python Data Science Stack — A First Look

LIBRARY	JOB	COVERED IN
NumPy	Fast, array-based numerical computing	ds1-2
pandas	Labeled, tabular data — the DataFrame	ds1-3 through ds1-5
Matplotlib	The foundational plotting library	ds1-7
Seaborn	Statistical plots, built on Matplotlib	ds1-8
scikit-learn	Classical machine learning models	Not this course — ml1

WHY THIS COURSE'S OWN CHAPTER ORDER MATCHES THE TABLE ABOVE

> Every chapter in this course exists to deliver one row of that table in depth. By the time `ds1-9` formally names the EDA methodology, every tool it draws on will already be familiar — nothing in this course's own capstone (`ds1-10`) introduces a new library for the first time.

A Short, Grounding Example

A retail company wants to know why weekend sales dipped last month. A data scientist starts by **collecting** the relevant sales records (a database export, in this case — `ds1-3`'s own CSV-reading skills apply directly). The raw export has missing store IDs and a few duplicated rows from a failed sync — **cleaning** (`ds1-4`) fixes that before anything else happens. **Exploring** the cleaned data (`ds1-5` – `ds1-9`) reveals the dip is concentrated at one specific store, not company-wide — a pattern no amount of staring at raw numbers would have surfaced as quickly as a grouped chart does. No model was ever built; the answer came entirely from disciplined cleaning and exploration. **Communicating** that finding — one clear chart, one clear sentence — closes the loop.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own four-term distinction, why "I want to learn data science" and "I want to learn machine learning" are not the same request, and identify which of the four terms this course itself is scoped around.

View solution

EXERCISE 2

Using this chapter's own five-stage workflow, explain which stage is deliberately marked out of scope for this course, which stage gets the most chapters, and why that allocation matches this chapter's own stated reasoning.

View solution

EXERCISE 3

Explain why a Jupyter notebook's cell-based execution model is a better fit for data science work than py1-1's own plain top-to-bottom script model, without concluding that notebooks make script-based Python obsolete in general.

View solution

CHAPTER 1 QUICK REFERENCE

- Data science ≠ machine learning — ML is one tool inside the broader discipline of data science, not a synonym for it
- The workflow: **Collect** → **Clean** → **Explore** → **Model** → **Communicate** — this course covers everything except Model (that's ml1's job)
- Jupyter notebooks — cell-based, iterative, inline output — the standard exploratory environment, distinct from py1-1's own script model
- Stack roadmap: **NumPy** (ds1-2) → **pandas** (ds1-3–ds1-5) → **Matplotlib** (ds1-7) → **Seaborn** (ds1-8)
- **Next chapter:** NumPy Fundamentals

CHAPTER 2 OF 10

NumPy Fundamentals

Data Science Fundamentals

Chapter 2 · NumPy Fundamentals

`ds1-1` 's own stack roadmap named NumPy as the first library this course actually delivers in depth. Everything in the rest of this course — pandas' own DataFrame (`ds1-3`), the statistics chapter (`ds1-6`), even the plotting libraries (`ds1-7` , `ds1-8`) — is built directly on top of what this chapter introduces.

The ndarray vs. py1-6's Own Python List

`py1-6` covered the Python list: flexible, resizable, and able to hold mixed types in the same container (a string, an integer, and another list, all in one list, is perfectly valid Python). NumPy's core data structure, the **ndarray** (n-dimensional array), makes the opposite trade-off deliberately: every element must be the same fixed type, and — while an array can be resized — its whole point is to represent a fixed-shape block of homogeneous numerical data efficiently.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
zeros = np.zeros(5)          # array([0., 0., 0., 0., 0.])
ones = np.ones((2, 3))       # a 2x3 array of 1.0
ranged = np.arange(0, 10, 2) # array([0, 2, 4, 6, 8])
```

Vectorization — Why This Trade-off Exists

Compare summing every element of a million-item collection, squared, using `py1-6` 's own list-and-loop approach against NumPy's own vectorized equivalent:

```
total = 0
for x in my_list:            # plain Python loop
    total += x ** 2

total = (my_array ** 2).sum()  # NumPy — no explicit loop at all
```

The vectorized version isn't just shorter — it's genuinely, dramatically faster on large data, often by one to two orders of magnitude. The reason is structural, not a matter of Python's own loop syntax being slow in some vague sense: a Python list stores a collection of separate objects scattered in memory, and a plain Python `for` loop has to check each element's type and dispatch the right operation for it, one element at a time, through Python's own interpreter overhead. A NumPy array stores its elements as one single, contiguous, fixed-type block of memory, and a vectorized operation like `** 2` runs as one single, compiled, low-level loop over that block — no per-element type-checking, no per-element interpreter dispatch.

> **THE SAME UNDERLYING KIND OF REASONING AS BUILD-TOOLING1-6**
>
> This is a genuine cousin of `build-tooling1-6`'s own explanation for why Go/Rust-based build tools outrun JavaScript-based ones — compiled, low-level execution avoiding per-item interpreter overhead that an interpreted language pays repeatedly. The specific mechanisms differ (that chapter was about AOT-compiled tooling vs. JIT-compiled JS; this is about a single compiled C loop vs. Python's own per-element interpreter dispatch), but the underlying shape of the argument — avoid paying interpreter overhead once per element by pushing the work down into compiled code — is the same.

Broadcasting

Broadcasting lets NumPy apply an operation between two arrays of different, but *compatible*, shapes — without writing an explicit loop to line them up. The simplest case: adding a single number to an entire array applies that number to every element.

```
arr = np.array([1, 2, 3])
arr + 10        # array([11, 12, 13]) — the scalar is "broadcast" across every element

matrix = np.array([[1, 2, 3], [4, 5, 6]])
row = np.array([10, 20, 30])
matrix + row    # row is broadcast across each row of matrix
```

The compatibility rule, informally: comparing shapes from the trailing dimension backward, each pair of dimensions must either match exactly or one of them must be `1`. A `(2, 3)` array and a `(3,)` array are compatible (the trailing dimensions both read `3`); a `(2, 3)` array and a `(2,)` array are not, without reshaping first.

Indexing, Slicing & Boolean Masking

Basic indexing and slicing work much like `py1-6`'s own list slicing (`arr[1:4]`, `arr[-1]`). NumPy adds a capability plain lists don't have at all: **boolean masking** — indexing an array with a condition that evaluates to an array of `True`/`False` values, keeping only the elements where the condition holds.

```
arr = np.array([3, 7, 2, 9, 4])
arr > 5              # array([False, True, False, True, False])
arr[arr > 5]          # array([7, 9]) — only the matching elements
```

WORTH REMEMBERING AHEAD OF DS1-3

This exact pattern — filtering data down to only the rows that satisfy a condition — is precisely what pandas' own row-filtering syntax (`ds1-3`) is built on top of. Recognizing boolean masking here means `ds1-3` 's own filtering syntax will look like a direct, familiar extension rather than new material.

Basic Linear Algebra

A quick preview, not a full course: `arr.reshape(2, 3)` changes an array's shape without changing its data; `arr.T` transposes a 2D array; `a @ b` (or `np.dot(a, b)`) performs matrix multiplication. These three operations alone are the mathematical backbone of how a neural network actually computes a prediction — a fact `nn1` will build on directly once this course is complete.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own contiguous-memory/compiled-loop explanation, why a NumPy vectorized operation is faster than py1-6's own list-and-loop approach — not just "NumPy is optimized," but the specific structural reason.

View solution

EXERCISE 2

Using this chapter's own broadcasting compatibility rule, explain why a (2, 3) array and a (3,) array can be added directly, while a (2, 3) array and a (2,) array cannot without reshaping.

View solution

EXERCISE 3

Explain what boolean masking is, using this chapter's own example, and explain why this chapter's own warn-box specifically calls this out as worth remembering ahead of ds1-3.

View solution

CHAPTER 2 QUICK REFERENCE

- **ndarray** — fixed-type, contiguous-memory array, vs. py1-6's own flexible, mixed-type list
- **Vectorization** — one compiled loop over contiguous memory, avoiding per-element Python interpreter overhead
- **Broadcasting** — operate across compatible shapes with no explicit loop; trailing dimensions must match or be 1
- **Boolean masking** (`arr[arr > 5]`) — filtering by condition, the direct basis for ds1-3's own pandas filtering
- **reshape / T / @** — a first preview of the linear algebra nn1 will build on directly
- **Next chapter:** Pandas Fundamentals — Series & DataFrame

CHAPTER 3 OF 10

Pandas Fundamentals — Series & DataFrame

Data Science Fundamentals

Chapter 3 · Pandas Fundamentals — Series & DataFrame

`ds1-2` built the ndarray. This chapter delivers the second row of `ds1-1`'s own stack roadmap: pandas, the library this entire course's own idea of "a dataset" is actually built around. Every remaining chapter through `ds1-9` works with the two structures introduced here.

Series — A Labeled ndarray

A pandas **Series** is, structurally, exactly what `ds1-2`'s own ndarray already was, plus one addition: a labeled **index** alongside the values.

```
import pandas as pd

s = pd.Series([120, 340, 95, 410], index=["Mon", "Tue", "Wed", "Thu"])
s["Tue"]      # 340 — access by label
s.values      # array([120, 340, 95, 410]) — the underlying ndarray, unchanged from ds1-2
```

That last line is worth pausing on: `.values` hands back a genuine NumPy ndarray. A Series isn't a new kind of data underneath — it's an ndarray with labels attached on top.

DataFrame — A Table of Aligned Series

A **DataFrame** extends the same idea to two dimensions: a table where each column is its own Series, and every column shares the same row index. `ds1-1`'s own throughline promised this explicitly — a DataFrame is a labeled wrapper around an ndarray, not a separate structure invented from scratch.

```
df = pd.DataFrame({
    "product": ["Latte", "Espresso", "Latte", "Mocha"],
    "quantity": [12, 8, 15, 6],
    "revenue": [54.00, 24.00, 67.50, 30.00],
})
df.values     # a 2D ndarray underneath — same relationship as Series.values above
```

Reading Real Data — CSV and JSON

`py2-7` covered Python's own `json` module — `json.load()` parsing a file into nested dicts and lists by hand, followed by manually walking that structure to pull out the fields you actually want. At dataset scale, pandas replaces that manual walk with a single call:

```
df = pd.read_csv("sales.csv")
df = pd.read_json("sales.json")   # the py2-7 json.load() step, plus the table-construction step, in one call
```

NOT A REPLACEMENT FOR PY2-7 — A HIGHER LAYER BUILT ON TOP OF IT

`pd.read_json()` is doing the exact same underlying parsing `py2-7`'s own `json.load()` does — it's just also handling the "now build a table from this nested structure" step that would otherwise be written by hand. Knowing what `json.load()` actually does underneath makes it much easier to understand why `read_json()` sometimes needs extra arguments (`orient=`, for deeply nested JSON) to know how to flatten a structure that isn't already table-shaped.

.loc vs. .iloc — A Genuinely Common Beginner Confusion

Both select rows and columns. The difference is what they select *by*:

	SELECTS BY	EXAMPLE
.loc	Label (the index value itself)	`df.loc[2]` — the row labeled 2, even after sorting or filtering has reordered rows
.iloc	Integer position (0-based, like a list)	`df.iloc[2]` — the third row physically, regardless of what its label happens to be

WHY THIS DISTINCTION ACTUALLY BITES PEOPLE

For a fresh DataFrame with a default 0, 1, 2, ... index, `.loc[2]` and `.iloc[2]` happen to return the same row — which hides the difference until, later, some rows get filtered or sorted and the index labels no longer line up with position. At that point `.loc[2]` and `.iloc[2]` can silently return two completely different rows. Preferring `.loc` for label-based work and `.iloc` only when position genuinely is what's meant avoids this trap entirely.

Boolean Filtering — ds1-2's Own Mechanism, One Layer Up

```
df[df["quantity"] > 10]   # only rows where quantity exceeds 10
```

This is `ds1-2`'s own boolean masking, applied to whole rows instead of individual array elements — `df["quantity"] > 10` produces a Series of `True`/`False` values (one per row), and indexing the DataFrame with it keeps only the matching rows. Exactly the pattern `ds1-2`'s own warn-box flagged as worth remembering.

A First Look, Column Selection & Derived Columns

```
df.head()      # first 5 rows — a quick look
df.info()      # column types, non-null counts
df.describe()  # summary statistics — previews ds1-6

df["revenue"]                 # select one column (a Series)
df["avg_price"] = df["revenue"] / df["quantity"]   # add a computed column
```

This Course's Own Running Dataset

Picking up `ds1-1`'s own retail example directly: a small daily sales log for a chain of coffee shops. This exact table — messy details included — is what `ds1-4` spends its own chapter cleaning up, so nothing here needs fixing yet.

ORDER_ID	STORE_ID	DATE	PRODUCT	QUANTITY	REVENUE
1001	S1	2026-07-10	Latte	12	54.00
1002	S2	2026-07-10	Espresso	8	24.00
1003	NaN	2026-07-10	Latte	15	67.50
1004	S1	07/11/2026	Mocha	6	30.00
1005	S2	2026-07-11	latte	10	45.00
1005	S2	2026-07-11	latte	10	45.00

Four real, distinct problems, deliberately left visible here rather than fixed on sight: a missing `store_id`, an inconsistent date format on row 1004, an inconsistent product-name capitalization on row 1005, and a fully duplicated row at the end. `ds1-4` addresses each one by name.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own .values examples, why a DataFrame is described as "a labeled wrapper around an ndarray" rather than a genuinely new kind of data structure.

View solution

EXERCISE 2

Using this chapter's own warn-box, explain why .loc[2] and .iloc[2] can return the same row on a fresh DataFrame but different rows later, and explain the general rule for when to prefer one over the other.

View solution

EXERCISE 3

List the four distinct problems this chapter's own sample dataset deliberately leaves unfixed, and explain why leaving them visible here (rather than fixing them immediately) serves this course's own stated teaching approach.

View solution

CHAPTER 3 QUICK REFERENCE

- **Series** — a labeled ndarray · **DataFrame** — a table of aligned Series, both backed by ds1-2's own ndarray underneath (`.values`)
- `pd.read_csv()` / `pd.read_json()` — the py2-7 json.load() step plus table construction, in one call
- **.loc** selects by label, **.iloc** by integer position — they only look interchangeable on a fresh, unsorted index
- Boolean filtering (`df[df["col"] > x]`) — ds1-2's own masking mechanism, applied to whole rows
- This chapter's own sample dataset carries forward, messy details intact, into **ds1-4**
- **Next chapter:** Data Cleaning

CHAPTER 4 OF 10

Data Cleaning

Data Science Fundamentals

Chapter 4 · Data Cleaning

`ds1-3` introduced a small coffee-shop sales table with four real, named problems, left deliberately unfixed. This chapter fixes each one — the same dataset, not a fresh example — so the techniques below are demonstrated on a problem already familiar rather than an abstract new one.

Problem 1: Missing Data — Row 1003's store_id

Detecting and handling missing values

```
df.isna()            # a same-shape DataFrame of True/False, marking every missing cell
df["store_id"].isna().sum()   # how many missing values in just this column

df.dropna(subset=["store_id"])          # option A: drop rows with a missing store_id
df["store_id"].fillna("UNKNOWN")        # option B: fill missing values with a placeholder
```

Neither option is mechanically "correct" — they're a real trade-off. Dropping row 1003 entirely discards a genuine sale that happened, distorting any total revenue figure downstream. Filling it with a placeholder like `"UNKNOWN"` keeps the sale in the total but makes any *per-store* analysis (`ds1-5`'s own `groupby`) silently wrong for that one row. For this dataset, filling with `"UNKNOWN"` is the better trade-off — the revenue total stays accurate, and the ambiguity is at least visible and labeled rather than silently dropped.

Problem 2: The Duplicated Row

Finding and removing exact duplicates

```
df.duplicated()          # True for each row that's an exact repeat of an earlier one
df.drop_duplicates()     # keeps the first occurrence, drops the rest
```

The final row in `ds1-3`'s own table exactly repeats order 1005 — same order ID, same store, same date, same product, same quantity, same revenue. `drop_duplicates()` removes it cleanly, since it isn't a second, legitimate sale — it's the same row appearing twice, most plausibly from a data-export glitch.

Problem 3: The Inconsistent Date Format

Type conversion — why mixed formats block real dates

```
df["date"] = pd.to_datetime(df["date"])
```

Row 1004's own `"07/11/2026"` and every other row's `"2026-07-10"`-style value are both, to pandas, just **strings** until converted — and two differently-formatted strings can't be compared, sorted, or subtracted from each other meaningfully. `pd.to_datetime()` parses each string into an actual datetime value, at which point the original formatting differences disappear entirely — what's stored afterward is a real date, not text that merely looks like one, and every row is now directly comparable regardless of how it was originally typed.

Problem 4: Inconsistent Capitalization

String cleaning

```
df["product"] = df["product"].str.title()   # "latte" -> "Latte"
```

Without this fix, `ds1-5`'s own `groupby("product")` would treat `"latte"` and `"Latte"` as two entirely separate products, silently splitting one product's own totals in two. Pandas' `.str` accessor applies ordinary Python string methods (`py1-5`'s own string-methods material) across an entire column at once, the same vectorized spirit as `ds1-2`'s own numeric operations, just for text.

A First Look at Outlier Detection

A quick, practical rule of thumb, not yet the full statistical grounding (`ds1-6` covers that properly): a value is a plausible outlier if it falls more than **1.5 × the interquartile range (IQR)** below the 25th percentile or above the 75th percentile.

```
q1, q3 = df["revenue"].quantile([0.25, 0.75])
iqr = q3 - q1
outliers = df[(df["revenue"] < q1 - 1.5*iqr) | (df["revenue"] > q3 + 1.5*iqr)]
```

This is a mechanical flag, not an automatic verdict — a flagged value might be a genuine data-entry error, or it might be a real, unusually large sale that's actually the most interesting row in the whole dataset. `ds1-6`'s own statistics chapter explains exactly what a quartile and an IQR are, and why `1.5×` specifically is the conventional threshold.

Before & After

PROBLEM	BEFORE	AFTER
Missing store_id	NaN (row 1003)	"UNKNOWN"
Inconsistent date	Mixed string formats	Uniform datetime type
Inconsistent product casing	"latte" / "Latte"	"Latte" (uniform)
Duplicate row	Order 1005 appears twice	Removed — appears once

CLEANING DECISIONS ARE JUDGMENT CALLS — DOCUMENT THEM

Every fix above involved a real choice, not a mechanical certainty — filling vs. dropping the missing `store_id` is the clearest example. A different analyst, with a different downstream question in mind, might reasonably have chosen to drop that row instead. Real data science practice documents these decisions explicitly (in comments, in a notebook's own text cells) rather than letting them disappear silently into the cleaned dataset — a conclusion drawn later is only as trustworthy as the cleaning choices behind it.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own reasoning, why filling row 1003's missing store_id with "UNKNOWN" was judged the better trade-off here than dropping the row entirely, and describe a different situation where dropping would be the better choice instead.

View solution

EXERCISE 2

Explain, using this chapter's own reasoning about pd.to_datetime(), why "07/11/2026" and "2026-07-10" can't be meaningfully compared before conversion, and what specifically changes once both are converted.

View solution

EXERCISE 3

Explain why this chapter describes the IQR-based outlier rule as "a mechanical flag, not an automatic verdict," and give a concrete example of a flagged value that would be wrong to simply discard.

View solution

CHAPTER 4 QUICK REFERENCE

- **Missing data** — `isna()` to detect, `dropna()` vs. `fillna()` as a real trade-off, not a mechanical choice
- **Duplicates** — `duplicated()` / `drop_duplicates()`
- **Type conversion** — `pd.to_datetime()` turns mismatched date strings into genuinely comparable values
- **String cleaning** — the `.str` accessor applies py1-5's own string methods across a whole column, vectorized
- **Outliers** — the 1.5×IQR rule as a flag, not a verdict; full statistical grounding in `ds1-6`
- Cleaning decisions are judgment calls — document them, don't let them vanish silently
- **Next chapter:** Data Wrangling & Reshaping

CHAPTER 5 OF 10

Data Wrangling & Reshaping

Data Science Fundamentals

Chapter 5 · Data Wrangling & Reshaping

`ds1-4` left the coffee-shop sales table clean: a consistent `store_id`, real datetime values, uniform product names, no duplicates. This chapter combines and reshapes that same cleaned table — and for a reader who already knows SQL (`mysql2`, `mysql3`), most of what follows is genuinely familiar material wearing pandas' own syntax.

groupby — Pandas' Own GROUP BY

`mysql_foundations_07` (`mysql2`'s own Aggregate Functions and GROUP BY chapter) covered grouping rows and computing an aggregate per group in SQL. Pandas' `groupby` does the exact same conceptual job:

```
df.groupby("product")["revenue"].sum()
# SQL equivalent: SELECT product, SUM(revenue) FROM sales GROUP BY product;

df.groupby("product").agg(total_revenue=("revenue", "sum"), orders=("order_id", "count"))
# SQL equivalent: SELECT product, SUM(revenue) AS total_revenue, COUNT(order_id) AS orders
#                 FROM sales GROUP BY product;
```

`.agg()` runs multiple aggregations at once, each labeled — the same shape as writing several aggregate functions in one SQL `SELECT` list.

merge — Pandas' Own JOIN

Introducing a second, small table — a store lookup, mapping `store_id` to a human-readable name and city:

STORE_ID	STORE_NAME	CITY
S1	Sunrise Coffee — Main St	Springfield
S2	Sunrise Coffee — Riverside	Springfield

```
merged = pd.merge(df, stores_df, on="store_id", how="left")
```

```
# SQL equivalent: SELECT * FROM sales LEFT JOIN stores ON sales.store_id = stores.store_id;
```

PANDAS `how=`	SQL EQUIVALENT (MYSQL3)	KEEPS
`"inner"`	INNER JOIN (mysql_advanced_02)	Only rows with a match in both tables
`"left"`	LEFT JOIN (mysql_advanced_02)	Every row from the left table, matched where possible
`"right"`	RIGHT JOIN (mysql_advanced_03)	Every row from the right table, matched where possible
`"outer"`	FULL OUTER JOIN	Every row from both tables, matched where possible

That "UNKNOWN" `store_id` from `ds1-4` — kept, not dropped, precisely because a `"left"` join was used — now has no match in `stores_df`, and its `store_name`/`city` columns come back as `NaN` rather than the row disappearing. This is exactly the missing-data situation `ds1-4` already covered, arriving again from a genuinely different source.

concat — Stacking, Not Matching

`merge` combines two tables based on matching key values. `concat` does something structurally different: it glues tables together along an axis, with no matching logic at all — useful for a case like combining July's sales file with August's, where both files share identical columns and just need to be stacked into one longer table.

```
full_year = pd.concat([july_df, august_df])   # stacked vertically, row-wise
```

Pivot Tables — Reshaping Long to Wide

A reader who's built a pivot table in a spreadsheet already knows this concept — pandas' own version does the same reshaping, in code:

```
pd.pivot_table(merged, index="date", columns="product", values="revenue", aggfunc="sum")
```

This is genuinely just `groupby` plus a reshape, combined into one call: it groups by *two* things at once (date and product), aggregates `revenue` within each combination, and then lays the result out as a real table — one row per date, one column per product — rather than the single flat column of grouped results `groupby` alone produces.

MELT — THE REVERSE DIRECTION

`pd.melt()` does the opposite reshape: it takes a wide table (many columns, one per category) and turns it back into a long, tidy table (one row per observation). Not covered in depth here, but worth knowing it exists as pivot's own counterpart.

Putting It Together

```
merged = pd.merge(df, stores_df, on="store_id", how="left")
summary = merged.groupby(["store_name", "product"]).agg(
    total_revenue=("revenue", "sum"),
    orders=("order_id", "count"),
)
```

One merge, one grouped aggregation — a real per-store, per-product summary table, the exact shape of analysis `ds1-9`'s own EDA chapter will lean on directly to actually spot patterns in a dataset, rather than just describing the mechanics of getting there.

Hands-On Exercises

EXERCISE 1

Using this chapter's own SQL-equivalent comments, translate df.groupby("product")["revenue"].mean() into the equivalent SQL statement, and explain what each part of the pandas call corresponds to in SQL.

View solution

EXERCISE 2

Explain, using this chapter's own reasoning, why the "UNKNOWN" store_id row from ds1-4 ends up with NaN values in store_name and city after the left join, rather than being dropped from the result or causing an error.

View solution

EXERCISE 3

Explain why this chapter describes pivot_table as "genuinely just groupby plus a reshape, combined," using the specific example given (grouping by date and product at once) to justify that description.

View solution

CHAPTER 5 QUICK REFERENCE

- **groupby** + **.agg()** — pandas' own GROUP BY, direct parallel to mysql_foundations_07
- **merge** — pandas' own JOIN; `how="inner"/"left"/"right"/"outer"` map directly onto mysql3's own JOIN types
- **concat** — stacking tables along an axis, no key-matching involved (unlike merge)
- **pivot_table** — groupby across two dimensions at once, reshaped into a wide table · **melt** reverses it
- A left join surfaces ds1-4's own "UNKNOWN" placeholder again — the same missing-data situation, from a new source
- **Next chapter:** Basic Statistics for Data Science

CHAPTER 6 OF 10

Basic Statistics for Data Science

Data Science Fundamentals

Chapter 6 · Basic Statistics for Data Science

`ds1-4` used quartiles and an IQR without fully explaining either, promising this chapter would. This is that chapter — and everything here is deliberately vocabulary-first: `ds1-7` through `ds1-9` all lean on the terms defined below without redefining them.

Mean, Median & Mode — And Why the Choice Matters

Mean is the familiar average — sum divided by count. **Median** is the middle value once everything is sorted. **Mode** is the most frequent value. They usually sit close together — until a dataset has extreme values, at which point they diverge in a way that actually matters.

DS1-4'S OWN CATERING ORDER, REVISITED

`ds1-4`'s own exercise raised a large, genuine catering order — statistically an outlier, but a real sale. That single order pulls the **mean** order size noticeably upward, even though most individual orders are much smaller — the mean is sensitive to extreme values because every value, including the largest, contributes to the sum. The **median** order size barely moves at all, because it only cares about which value sits in the middle position, not how large the largest one is. Neither statistic is "wrong" — a mean is the right choice for projecting total expected revenue; a median is the right choice for describing what a "typical" order actually looks like. Reporting only one of the two, without saying which, is a common way real analysis becomes quietly misleading.

Spread — Delivering on ds1-4's Own IQR Promise

Range is simply max minus min — the crudest measure of spread, and highly sensitive to a single extreme value. **Variance** is the average of each value's squared distance from the mean; **standard deviation** is variance's own square root, put back into the original units so it's directly interpretable.

Quartiles split sorted data into four equal-sized groups: **Q1** (the 25th percentile) is the value below which a quarter of the data falls; **Q2** is the median itself; **Q3** (the 75th percentile) is the value below which three-quarters of the data falls. **IQR** (interquartile range) is simply `Q3 - Q1` — the spread of the *middle* half of the data, deliberately ignoring the most extreme quarter on each end. `ds1-4`'s own `1.5×IQR` threshold is a long-standing statistical convention (originating with the box plot itself) for marking a value as unusually far

outside that middle-half range — not an arbitrary number, but not a law of nature either, just a widely agreed default.

> **PREVIEWING DS1-8**
> Q1, the median, Q3, and the 1.5×IQR whiskers are literally what a **box plot** draws directly — `ds1-8`'s own box-plot section will look like a picture of exactly the numbers defined here, not new material.

Distributions & the Normal Curve

A **distribution** describes the overall shape of how values are spread across their possible range — not any single number, but the whole pattern. The **normal distribution** (the familiar symmetric "bell curve") is the single most important reference shape in statistics, fully described by just two numbers: its mean (the center) and its standard deviation (how wide the bell is). The **empirical rule** gives a fast, useful approximation for normally distributed data: roughly 68% of values fall within one standard deviation of the mean, roughly 95% within two, and roughly 99.7% within three.

Correlation vs. Causation

Correlation measures how strongly two variables move together, typically as a number from `-1` to `1` (Pearson's correlation coefficient): `1` means they move in perfect lockstep upward together, `-1` means one rises exactly as the other falls, `0` means no linear relationship at all. `ds1-7`'s own scatter plots are the standard way to actually see a correlation visually.

> **WHY CORRELATION NEVER PROVES CAUSATION ON ITS OWN**
> Ice cream sales and drowning incidents correlate strongly — both rise in summer. Neither causes the other; a third, unmeasured factor (hot weather, driving both more ice cream purchases and more swimming) is the real driver. This is a **confounding variable**, and it's the standard reason a real, measured correlation can exist between two things with no direct causal link between them at all. Seeing a strong correlation is a genuine reason to investigate further — it is never, on its own, proof of cause and effect.

Probability — Just Enough, Not a Full Course

A **probability** is a number from 0 to 1 describing how likely an event is. Two events are **independent** if one happening tells you nothing about whether the other happens (two separate coin flips); they're **dependent** otherwise (drawing two cards from the same deck without replacement — the first draw changes what's left for the second). This is deliberately the full extent of probability covered here — enough vocabulary to understand later statistical statements, not a dedicated probability course.

Sampling

A **population** is every member of the group actually being studied; a **sample** is the subset actually measured, because measuring an entire population is usually impossible or impractical. A sample is only useful if it's **representative** — if it resembles the population in the ways that matter for the question being asked.

A CONCRETE SAMPLING-BIAS EXAMPLE, TIED TO DS1-1'S OWN SCENARIO

Revisiting `ds1-1`'s own retail example: if a data scientist only sampled weekday transactions when investigating a weekend sales dip, the sample would be structurally incapable of ever revealing the pattern being investigated — not because of a calculation error, but because the sample itself excludes exactly the data the question depends on. This is **sampling bias**: a flawed sampling method producing a misleading conclusion even when every individual calculation performed on it is done correctly.

Hands-On Exercises

EXERCISE 1

Using this chapter's own catering-order example, explain why reporting only the mean order size (without the median) could be misleading, and identify which of the two statistics is the right choice for projecting total expected revenue.

View solution

EXERCISE 2

Explain, using this chapter's own definitions, what IQR actually measures and why the 1.5×IQR threshold from ds1-4 is described here as "a widely agreed default," not an arbitrary number or a law of nature.

View solution

EXERCISE 3

Using this chapter's own ice-cream/drowning example and its own weekday-sampling example, explain the difference between a confounding-variable problem and a sampling-bias problem — are they the same kind of mistake?

View solution

CHAPTER 6 QUICK REFERENCE

- **Mean** — sensitive to outliers · **Median** — robust to them · **Mode** — most frequent value

- **Q1/Q2(median)/Q3** split sorted data into quarters; **IQR = Q3 − Q1**; **1.5×IQR** is a box-plot convention, delivering ds1-4's own deferred promise
- **Distribution** — the overall shape of spread; the **normal curve** and the 68-95-99.7 empirical rule
- **Correlation** (−1 to 1) never proves **causation** alone — watch for confounding variables
- **Probability** basics — independent vs. dependent events, deliberately scoped light
- **Sampling** — population vs. sample, and sampling bias as a structural flaw, not a calculation error
- **Next chapter:** Data Visualization I: Matplotlib Fundamentals

CHAPTER 7 OF 10

Data Visualization I: Matplotlib Fundamentals

Data Science Fundamentals

Chapter 7 · Data Visualization I: Matplotlib Fundamentals

`ds1-6` built a shared statistical vocabulary — mean, distribution, correlation — entirely in words and numbers. This chapter starts turning that vocabulary into pictures. Matplotlib is the foundational plotting library the entire Python data-visualization ecosystem, including `ds1-8`'s own Seaborn, is built directly on top of.

The Figure/Axes Model

Matplotlib's core object hierarchy has two parts worth knowing by name: a **Figure** is the whole canvas — the entire window or image being produced. An **Axes** is one individual plot area within that figure, with its own x-axis, y-axis, and content. A figure can hold one Axes, or several arranged in a grid.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()      # one Figure, one Axes
ax.plot(df["date"], df["revenue"])
ax.set_title("Daily Revenue")
ax.set_xlabel("Date")
ax.set_ylabel("Revenue ($)")
plt.show()
```

WHY THIS COURSE USES FIG, AX = PLT.SUBPLOTS(), NOT JUST PLT.PLOT()

A shorter, older style (`plt.plot(...)` directly, with no explicit figure or axes) also exists and still works — but it implicitly tracks "the current figure" behind the scenes, which gets confusing fast once more than one chart is involved. Explicitly naming `fig` and `ax` scales cleanly to multiple subplots and is the style used consistently through the rest of this course.

Four Chart Types, Four Jobs

1

Line Plot

2

Bar Plot

`ax.plot(x, y)` — trends over an ordered sequence, almost always time. A natural fit for the coffee-shop dataset's own `date` column, now a real datetime thanks to `ds1-4`.

`ax.bar(categories, values)` — comparing discrete categories side by side. The direct visual counterpart to `ds1-5`'s own `groupby` results (revenue per product, revenue per store).

3

Scatter Plot

`ax.scatter(x, y)` — the relationship between two continuous variables. This is `ds1-6`'s own promised payoff: "the standard way to actually see a correlation visually."

4

Histogram

`ax.hist(values, bins=...)` — the actual shape of a distribution, bucketed into ranges. Turns `ds1-6`'s own mean/standard-deviation description of a normal curve into something actually seen, not just described.

Line Plot — Revenue Over Time

```
daily = df.groupby("date")["revenue"].sum()
fig, ax = plt.subplots()
ax.plot(daily.index, daily.values)
ax.set_title("Daily Revenue Over Time")
```

Bar Plot — Revenue by Product

```
by_product = df.groupby("product")["revenue"].sum()
fig, ax = plt.subplots()
ax.bar(by_product.index, by_product.values)
ax.set_title("Total Revenue by Product")
```

Scatter Plot — Quantity vs. Revenue

```
fig, ax = plt.subplots()
ax.scatter(df["quantity"], df["revenue"])
ax.set_xlabel("Quantity")
ax.set_ylabel("Revenue ($)")
```

A visibly upward-sloping cluster of points here would be the visual signature of a strong positive correlation — `ds1-6`'s own Pearson coefficient close to `1`. `ds1-6`'s own confounding-variable warning still applies at

full strength here: a clear pattern in a scatter plot is a reason to investigate, never proof of a direct causal link on its own.

Histogram — The Shape of Order Revenue

```
fig, ax = plt.subplots()
ax.hist(df["revenue"], bins=10)
ax.set_title("Distribution of Order Revenue")
```

This is the chart that would actually make `ds1-6` 's own catering-order outlier visible at a glance — a tall cluster of ordinary-sized orders, with one isolated bar sitting far off to the right.

Choosing the Right Chart Type

QUESTION BEING ASKED	CHART	DELIVERS ON
How does this change over time?	Line	—
How do these categories compare?	Bar	ds1-5's own groupby results
Are these two variables related?	Scatter	ds1-6's own correlation coefficient
What's the overall shape of this variable?	Histogram	ds1-6's own distribution/normal-curve material

AN UNLABELED CHART IS A CHART THAT CAN'T BE TRUSTED

Every example above sets a title and axis labels deliberately, not as decoration. A chart with unlabeled axes technically shows the right shape but leaves a reader unable to verify what's actually being measured, in what units, over what range — exactly the kind of ambiguity `ds1-1` 's own "Communicate" stage of the workflow exists to prevent.

Hands-On Exercises

EXERCISE 1

Explain the difference between a Figure and an Axes in this chapter's own terms, and explain why this course's own fig, ax = plt.subplots() style scales better to multiple charts than the older plt.plot() style.

View solution

EXERCISE 2

Using this chapter's own chart-type table, explain which chart type you'd choose to compare total revenue across the two coffee shop stores, and which chart type you'd choose to check whether order quantity and revenue move together — and why each choice fits ds1-6's own vocabulary.

View solution

EXERCISE 3

Explain why this chapter says a clear upward pattern in a scatter plot is "a reason to investigate, never proof of a direct causal link," tying your answer back to ds1-6's own confounding-variable material.

View solution

CHAPTER 7 QUICK REFERENCE

- **Figure** — the whole canvas · **Axes** — one plot area within it; `fig, ax = plt.subplots()` is this course's own consistent style
- **Line** — trend over time · **Bar** — compare categories (ds1-5's groupby) · **Scatter** — relationship between two variables (ds1-6's correlation) · **Histogram** — shape of a distribution (ds1-6's own normal-curve material)
- A scatter plot's own visible pattern is never proof of causation on its own — ds1-6's confounding-variable warning still applies
- Unlabeled charts undermine ds1-1's own "Communicate" stage of the workflow
- **Next chapter:** Data Visualization II: Seaborn & Statistical Plots

CHAPTER 8 OF 10

Data Visualization II: Seaborn & Statistical Plots

Data Science Fundamentals

Chapter 8 · Data Visualization II: Seaborn & Statistical Plots

`ds1-7` 's own tip-box promised this chapter's material would "look like a picture of exactly the numbers defined here [`ds1-6`], not new material." Seaborn earns that promise directly — every function below returns an ordinary Matplotlib Axes object underneath (`ds1-7` 's own foundation, unchanged), with statistical plot types and DataFrame-aware syntax layered on top.

What Seaborn Actually Adds

Three genuine additions over plain Matplotlib: statistical chart types (box, violin, heatmap, pair plots) that `ds1-7` 's own line/bar/scatter/histogram set doesn't include directly; nicer default styling out of the box; and DataFrame-aware syntax — passing a DataFrame plus column *names* directly, rather than raw arrays pulled out by hand first.

```
import seaborn as sns

sns.boxplot(data=df, y="revenue")   # DataFrame + column name, not a raw array
```

Box Plots — ds1-6's Own Vocabulary, Drawn Directly

This is the direct payoff of `ds1-6` 's own promise. Every visual element of a box plot maps onto a term already defined:

VISUAL ELEMENT	DS1-6 TERM
The box's bottom edge	Q1 (25th percentile)
The line inside the box	Median (Q2)
The box's top edge	Q3 (75th percentile)
The box's own height	IQR (Q3 − Q1)
The whiskers	1.5 × IQR reach
Individual dots beyond the whiskers	Flagged outliers — ds1-4's own IQR rule, drawn

The catering order from `ds1-4` / `ds1-6` would appear here as exactly that: an isolated dot, sitting alone past the upper whisker, visually confirming the same flag the mechanical `1.5×IQR` calculation already raised.

Violin Plots — Adding Shape to the Same Summary

```
sns.violinplot(data=df, y="revenue")
```

A violin plot shows the same quartile information a box plot does, mirrored into a smoothed, rotated shape — effectively `ds1-7`'s own histogram, reshaped and reflected, layered directly over the box plot's own summary statistics. This matters specifically when a distribution's **shape** carries real information a box plot alone hides: two distinct clusters of typical order sizes (a genuinely **bimodal** distribution — small individual orders and separate large group orders, with few orders in between) can produce a perfectly ordinary-looking box plot, since a box plot only ever shows five summary numbers, never the shape connecting them. A violin plot would show that same data as two visible bulges, immediately revealing structure the box plot's own five numbers couldn't.

Heatmaps — Every Pairwise Correlation, One Chart

`ds1-7`'s own scatter plot showed one pair of variables at a time. A correlation **heatmap** extends that to every numeric column at once:

```
correlations = df[["quantity", "revenue"]].corr()
sns.heatmap(correlations, annot=True, cmap="coolwarm")
```

`df.corr()` computes `ds1-6`'s own Pearson coefficient for every pair of numeric columns simultaneously, producing a square matrix. `sns.heatmap()` renders that matrix as color intensity — strong positive correlations in one color, strong negative in another, weak correlations pale — with `annot=True` printing the actual coefficient value inside each cell. With only two numeric columns this collapses to a single interesting cell; the real value appears once a dataset has several numeric columns worth comparing pairwise all at once, exactly the situation `ds1-9`'s own EDA methodology will lean on this for.

Pair Plots — ds1-7's Own Toolkit, Automated

```
sns.pairplot(df[["quantity", "revenue"]])
```

`sns.pairplot()` generates a full grid: a scatter plot for every pair of numeric columns (`ds1-7`'s own scatter plot, run automatically for every combination) plus a histogram for each column against itself along the

diagonal (`ds1-7`'s own histogram, likewise automated). It's genuinely everything `ds1-7` taught individually, generated for an entire dataset's worth of column pairs in one function call rather than one deliberate chart at a time.

THIS IS WHERE DS1-9 ACTUALLY STARTS

A correlation heatmap and a pair plot are, in practice, the first two things run against a genuinely new dataset during real exploratory analysis — a fast, wide first look before deciding which specific relationships deserve a closer, more deliberate chart. `ds1-9` formalizes exactly this instinct into a real, repeatable methodology.

Hands-On Exercises

EXERCISE 1

Using this chapter's own element-by-element table, explain what it means for a box plot to be "ds1-6's own vocabulary, drawn directly," and describe where the ds1-4 catering order would appear on one.

View solution

EXERCISE 2

Explain, using this chapter's own bimodal-distribution example, why a violin plot can reveal structure a box plot alone hides, even when both are computed from identical data.

View solution

EXERCISE 3

Explain why this chapter describes sns.pairplot() as "genuinely everything ds1-7 taught individually," specifically identifying which ds1-7 chart type appears on the diagonal and which appears off the diagonal.

View solution

CHAPTER 8 QUICK REFERENCE

- Seaborn returns ordinary ds1-7 Matplotlib Axes objects underneath — statistical chart types + DataFrame-aware syntax on top
- **Box plot** — Q1/median/Q3/IQR/whiskers drawn directly, delivering ds1-6's own promise
- **Violin plot** — box-plot summary plus ds1-7's histogram shape, mirrored; reveals bimodal structure a box plot alone hides
- **Heatmap** — `df.corr()` (ds1-6's Pearson coefficient) for every column pair at once, rendered as color

- **Pair plot** — ds1-7's scatter plots (off-diagonal) and histograms (diagonal), automated across every column pair
- **Next chapter:** Exploratory Data Analysis (EDA) — A Real Methodology

CHAPTER 9 OF 10

Exploratory Data Analysis (EDA) — A Real Methodology

Data Science Fundamentals

Chapter 9 · Exploratory Data Analysis (EDA) — A Real Methodology

`ds1-3` through `ds1-8` built a toolkit — reading data, cleaning it, combining it, describing it, plotting it — one technique at a time, on one running coffee-shop dataset. This chapter formally names the **method** those tools were all building toward, and deliberately applies it to a fresh dataset the coffee-shop material never touched: a small used-car listings table. The *method*, not a memorized sequence of steps on one specific dataset, is what this chapter needs to transfer — which is exactly why `ds1-10` 's own capstone will apply this same seven-step structure to a *third*, still-different dataset.

MAKE	MODEL	YEAR	MILEAGE	PRICE	FUEL_TYPE
Toyota	Corolla	2019	42,000	16,500	Petrol
Honda	Civic	2020	28,500	18,200	Petrol
Tesla	Model 3	2021	19,000	31,800	Electric
Ford	Focus	2015	88,000	7,200	Diesel
Toyota	Corolla	2022	9,500	21,900	Petrol
Ford	Focus	2016	76,000	7,800	Diesel
Jaguar	E-Type	1968	61,000	85,000	Petrol

The Seven-Step Methodology

1. **Understand the shape.** `df.shape` , `df.info()` , `df.head()` — `ds1-3` 's own first tools. How many rows and columns, what types, any obviously missing values.

2. **Summary statistics.** `df.describe()` — every one of `ds1-6` 's own vocabulary terms (mean, std, Q1, median, Q3) for every numeric column, in one call.

3. **Univariate analysis** — one variable at a time. A histogram (`ds1-7`) of `price` ; `df["fuel_type"].value_counts()` for a categorical breakdown.

4. **Bivariate analysis** — two variables at a time. A scatter plot (`ds1-7`) of `mileage` vs. `price` ; a box plot (`ds1-8`) of `price` grouped by `fuel_type` .
5. **Multivariate analysis** — three or more variables at once. A correlation heatmap and a pair plot (`ds1-8`) across every numeric column simultaneously.
6. **Spot anomalies.** `ds1-4` 's own IQR rule and `ds1-8` 's own box plot, applied here.
7. **Form hypotheses.** The actual point of everything above — turning patterns into specific, testable questions.

Steps 3–5, Worked

```
df["price"].plot(kind="hist")                          # univariate — ds1-7's histogram
df["fuel_type"].value_counts()                          # univariate — categorical breakdown

df.plot(kind="scatter", x="mileage", y="price")          # bivariate — ds1-7's scatter plot
sns.boxplot(data=df, x="fuel_type", y="price")           # bivariate — ds1-8's box plot, grouped

sns.heatmap(df[["year","mileage","price"]].corr(), annot=True)   # multivariate — ds1-8's heatmap
sns.pairplot(df[["year","mileage","price"]])                     # multivariate — ds1-8's pair plot
```

The mileage-vs-price scatter plot here shows something the coffee-shop dataset's own quantity-vs-revenue scatter (`ds1-7`) never did: a clear **negative** correlation — higher mileage, lower price. Same tool, same underlying Pearson mechanism (`ds1-6`), a genuinely different real-world relationship.

Step 6: Spotting Anomalies, Honestly

Running `ds1-4` 's own IQR rule on `price` flags the 1968 Jaguar E-Type immediately — its price sits far above every other listing. Exactly the honest distinction `ds1-4` and `ds1-6` already insisted on: this isn't a data-entry error to delete. A genuine vintage collector's car legitimately commands a price with no relationship at all to the mileage/age/price pattern the rest of the dataset follows — flagging it is correct; discarding it would be a real mistake, and would also quietly remove the row most worth asking a follow-up question about.

Step 7: Forming Hypotheses — The Actual Point of EDA

Everything above describes what the data *is*. This step turns that description into specific, testable questions:

- "Does mileage predict price more strongly than the car's year does?" — a question the heatmap's own two separate correlation values already give a first, rough answer to.
- "Is the Jaguar a genuinely different category of listing (collectible) that should be modeled separately from ordinary used cars?"
- "Does fuel type meaningfully shift price once mileage and year are accounted for?"

EDA IDENTIFIES WHAT'S WORTH MODELING — IT DOESN'T BUILD THE MODEL

None of these questions get answered by anything in this chapter. EDA's own job ends at forming a specific, well-motivated question backed by a real pattern already seen in the data — actually building something that predicts price from mileage and year is `ml1`'s entire job, start to finish. This chapter's own seven steps are the bridge that decides what's worth building in the first place, not the construction itself.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own seven-step list, which step each of the following belongs to: df.describe(), a box plot of price grouped by fuel_type, and a correlation heatmap across year/mileage/price.

View solution

EXERCISE 2

Explain why the mileage-vs-price scatter plot in this chapter shows a negative correlation while ds1-7's own quantity-vs-revenue scatter plot showed a positive one, and explain why this doesn't mean one chapter's own correlation mechanism is different from the other's.

View solution

EXERCISE 3

Using this chapter's own warn-box, explain the difference between what Step 7 (forming hypotheses) actually accomplishes and what ml1 accomplishes, and explain why "does mileage predict price?" is a hypothesis this chapter raises but doesn't answer.

View solution

CHAPTER 9 QUICK REFERENCE — THE SEVEN-STEP EDA METHODOLOGY

- **1. Shape** (ds1-3) → **2. Summary stats** (ds1-6) → **3. Univariate** (ds1-7) → **4. Bivariate** (ds1-7/ds1-8) → **5. Multivariate** (ds1-8) → **6. Anomalies** (ds1-4/ds1-8) → **7. Hypotheses**
- Applied here to a fresh used-car dataset — a genuinely different domain from ds1-3–ds1-8's own coffee-shop table, on purpose
- An outlier flagged in Step 6 (the Jaguar) may be the single most interesting row, not an error to delete
- Step 7 forms testable questions — it deliberately does not answer them; that's ml1's own job
- **Next chapter:** Capstone: A Full EDA Project on a Real Dataset

CHAPTER 10 OF 10

Capstone: A Full EDA Project on a Real Dataset

Data Science Fundamentals

Chapter 10 · Capstone: A Full EDA Project on a Real Dataset

A third dataset, unrelated to the coffee-shop sales table (`ds1-3`–`ds1-8`) or the used-car listings (`ds1-9`): a small employee-attrition table for a fictional company, asking a genuinely different kind of question — not "what predicts a price," but "what's associated with an employee leaving." `ds1-9`'s own seven-step methodology, applied start to finish, unchanged.

EMPLOYEE_ID	DEPARTMENT	AGE	YEARS_AT_COMPANY	SALARY	LEFT_COMPANY
E01	Sales	29	2.5	52,000	Yes
E02	Engineering	34	5.0	81,000	No
E03	Sales	41	8.0	67,000	No
E04	Engineering	26	1.0	74,000	Yes
E05	Support	52	14.0	58,000	No
E06	Sales	31	1.5	49,000	Yes
E07	Engineering	45	9.0	96,000	No
E08	Support	27	0.5	44,000	Yes

Step 1 — Shape

df.shape / df.info() / df.head()

8 rows, 6 columns. One categorical (`department`), one boolean-like categorical (`left_company`), four numeric. No missing values in this small sample — unusual for a real dataset, and worth naming rather than assuming.

Step 2 — Summary Statistics

df.describe()

Salary ranges roughly $44,000–$96,000, mean around $65,000. Years at company ranges from 0.5 to 14 — a wide spread worth investigating directly in Step 3.

Step 3 — Univariate Analysis

Histogram of years_at_company; value_counts() on department and left_company

```
df["years_at_company"].plot(kind="hist")
df["department"].value_counts()
df["left_company"].value_counts()
```

✓ Finding: years_at_company skews heavily toward short tenures — most employees in this sample have been at the company under three years.

Step 4 — Bivariate Analysis

Scatter (years_at_company vs. salary); box plot (salary by left_company)

```
df.plot(kind="scatter", x="years_at_company", y="salary")
sns.boxplot(data=df, x="left_company", y="salary")
```

✓ Finding: a box plot of salary grouped by left_company shows the "Yes" group's salaries sitting visibly lower than the "No" group's — a real, visible bivariate pattern, not yet a conclusion.

Step 5 — Multivariate Analysis

Correlation heatmap and pair plot across age, years_at_company, and salary

```
sns.heatmap(df[["age","years_at_company","salary"]].corr(), annot=True)
sns.pairplot(df[["age","years_at_company","salary"]])
```

✓ Finding: age and years_at_company correlate strongly and positively with each other — unsurprising, since both track roughly the same underlying "how established" idea — while salary correlates more weakly with either alone.

Step 6 — Spotting Anomalies

IQR rule on salary; visual confirmation via box plot

Nothing in this small sample clears the `1.5×IQR` threshold — a genuinely different, and equally valid, outcome from `ds1-9`'s own used-car dataset, where the Jaguar was flagged immediately. A real EDA process sometimes finds nothing anomalous, and that absence is itself a legitimate, worth-recording finding, not a failed step.

Step 7 — Forming Hypotheses

Turning Steps 3–6's own findings into testable questions

- "Does lower salary predict a higher likelihood of an employee leaving, once tenure is accounted for?" — directly motivated by Step 4's own box-plot finding.
- "Is short tenure itself a risk factor for leaving, independent of salary?" — directly motivated by Step 3's own skewed-tenure finding.
- "Does department matter, or is department's own apparent effect actually explained by department-level salary differences instead?" — a genuine, unresolved confounding-variable question, straight out of `ds1-6`'s own material.

Chapter Attribution

CAPSTONE ELEMENT	DRAWN FROM
The seven-step structure itself	ds1-9
df.shape / df.info() / df.head()	ds1-3
df.describe(), the salary/tenure vocabulary	ds1-6
Histogram, scatter plot	ds1-7
Box plot, heatmap, pair plot	ds1-8

CAPSTONE ELEMENT	DRAWN FROM
The IQR anomaly rule, and the honest "flag, not verdict" framing	ds1-4 / ds1-6
The confounding-variable question in Step 7	ds1-6
The Explore/Model boundary itself	ds1-1

Honest Scope Note

WHAT THIS CAPSTONE DELIBERATELY DOESN'T ATTEMPT

- **No ML modeling.** This capstone forms real hypotheses about what predicts attrition — it never builds, trains, or tests a model that actually predicts it. That's `ml1`'s entire job, and this course's own boundary, stated since `ds1-1`.
- **No live web scraping or API-based data collection.** All three datasets in this course (coffee-shop sales, used cars, this chapter's own employee table) were provided directly. A real, hands-on data-collection project — building a tool that actually gathers data from the web — is `dsproj1`'s own first project, chosen specifically because it doesn't need ML at all.
- **No big-data or distributed processing.** Every dataset in this course fits comfortably in memory on an ordinary laptop. Datasets too large for that are a genuinely different, more advanced problem this course doesn't attempt to solve.

Hands-On Exercises

EXERCISE 1

Explain why Step 6 finding no outliers in this capstone's dataset is described as "a legitimate, worth-recording finding, not a failed step," contrasting it with ds1-9's own used-car dataset where an outlier was found immediately.

View solution

EXERCISE 2

Explain why the Step 7 hypothesis about department is described as "a genuine, unresolved confounding-variable question," using ds1-6's own confounding-variable material to justify why department's own apparent effect can't simply be trusted at face value.

View solution

EXERCISE 3

Using this chapter's own scope note, explain why "no ML modeling" and "no live web scraping" are both listed as deliberately out of scope here, and identify which specific future course or project each one is deferred to.

View solution

CHAPTER 10 QUICK REFERENCE — COURSE SUMMARY

- The data science workflow: Collect → Clean → Explore → Model → Communicate (ds1-1) — this course covered everything except Model
- NumPy's ndarray (ds1-2) underlies pandas' Series/DataFrame (ds1-3), which ds1-4 cleans and ds1-5 combines/reshapes
- ds1-6's statistical vocabulary is what ds1-7's Matplotlib charts and ds1-8's Seaborn statistical plots actually visualize
- ds1-9 formalized the seven-step EDA methodology; this capstone applied it, unchanged, to a third, genuinely different dataset
- Next up in the Data Science & ML subject: **ml1** (Machine Learning Fundamentals) — picking up exactly where this course's own Explore/Model boundary leaves off